



UNIVERSIDAD DE GRANADA

Facultad de Ciencias
E.T.S. Ingenierías Informática y de Telecomunicación

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y
MATEMÁTICAS

TRABAJO DE FIN DE GRADO

Fundamentos del Deep Learning y desarrollo de un modelo de análisis de posiciones de ajedrez

Presentado por:
Adrián Rodríguez Montero

Tutores:
Francisco Herrera Triguero
Departamento de Ciencias de la Computación e Inteligencia Artificial

Francisco Javier Melero Rus
Departamento de Lenguajes y Sistemas Informáticos

Curso académico 2022-2023

Fundamentos del Deep Learning y desarrollo de un modelo de análisis de posiciones de ajedrez

Adrián Rodríguez Montero

Adrián Rodríguez Montero *Fundamentos del Deep Learning y desarrollo de un modelo de análisis de posiciones de ajedrez.*

Trabajo de fin de Grado. Curso académico 2022-2023.

**Responsable de
tutorización**

Francisco Herrera Triguero
Departamento de Ciencias de la Computación e Inteligencia Artificial

Francisco Javier Melero Rus
Departamento de Lenguajes y Sistemas Informáticos

Doble Grado en Ingeniería
Informática y Matemáticas

Facultad de Ciencias
E.T.S. Ingenierías
Informática y de
Telecomunicación

Universidad de Granada

DECLARACIÓN DE ORIGINALIDAD

D. Adrián Rodríguez Montero

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2022-2023, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 24 de noviembre de 2022

Fdo: Adrián Rodríguez Montero

A mi familia.

Índice general

Agradecimientos	xv
Breve resumen	xvii
Summary	xix
I. Introducción y objetivos	1
1. Introducción	3
1.1. Contextualización	3
1.2. Descripción del problema	3
1.3. Estructura del trabajo	4
1.4. Bibliografía fundamental	5
2. Objetivos	7
II. Fundamentos del Ajedrez y Ajedrez Computacional	9
3. Fundamentos del Ajedrez	11
3.1. ¿Qué es el ajedrez?	11
3.2. Origen del Ajedrez	12
3.2.1. Leyenda del origen	12
3.2.2. India, Chaturanga	13
3.2.3. China	14
3.3. Expansión y Evolución	15
3.4. Elementos del juego	16
3.4.1. Las piezas	17
3.4.2. El tablero	20
3.4.3. El reloj	20
3.4.4. Reglas generales	21
3.5. Fases del juego	23
3.5.1. Apertura	23
3.5.2. Medio Juego	23
3.5.3. Final	24
3.6. Ritmos de juego	24
3.7. Normas de notación ajedrecística	25
3.7.1. Notación Algebraica	25
3.7.2. Notación de Forsyth-Edwards	27

Índice general

4. La Historia del Ajedrez Computacional	29
4.1. Primeros Pasos	29
4.2. 1950: El Comienzo de la Era	30
4.3. Década de los 60 y los 70: Un Gran Cambio	31
4.4. Años 80: El Reto	32
4.5. 1990-1997: El Control de los Motores	33
4.6. 1997-2006: La Batalla Final	34
4.7. 2006-2017: La Época Dorada	34
4.8. 2017: Una Nueva Revolución	35
4.9. Desde 2019: Inicio de la Era de la Inteligencia Artificial	35
5. Elementos de un Motor de Ajedrez	37
5.1. Generador de movimientos	37
5.2. Codificaciones del tablero de ajedrez	37
5.2.1. Requisitos	38
5.2.2. Tipos basados en arrays	38
5.2.3. Bitboards	40
5.2.4. Otros métodos	44
5.3. Técnicas de búsqueda	44
5.3.1. Árboles	45
5.3.2. Efecto Horizonte	47
5.3.3. Fuerza Bruta	48
5.3.4. Búsqueda Selectiva	48
5.3.5. Búsqueda en Profundidad	49
5.3.6. Optimizaciones y Mejoras	53
5.3.7. Búsqueda en Anchura	56
5.3.8. Búsqueda Primero el Mejor	57
5.3.9. Búsqueda Paralela	59
5.4. Función de Evaluación	60
5.4.1. Balance material	61
5.4.2. Control del tablero, movilidad, espacio y tiempo	61
5.4.3. Estructura de peones	63
5.4.4. Seguridad del rey	64
6. Motores de Ajedrez	67
6.1. Motores de Ajedrez	67
6.1.1. Stockfish	68
6.1.2. Leela Chess Zero	70
6.1.3. Nemorino	71
6.1.4. Ethereal	71
III. Fundamentos Teóricos	73
7. Teorema de No Free Lunch	75
7.1. Preliminares	75
7.1.1. Conceptos básicos sobre Teoría de la Medida	75
7.1.2. Resultados previos	78

7.2.	Conceptos sobre Aprendizaje	80
7.2.1.	Funciones de pérdida generalizadas	82
7.3.	Teorema de No Free Lunch	82
8.	Teorema de Aproximación Universal	87
8.1.	Definiciones y Teoremas útiles del Análisis Funcional	87
8.2.	Teorema de Aproximación Universal para la función de activación Sigmoide y ReLU	89
9.	Aprendizaje Automático	95
9.1.	Aprendizaje Automático	95
9.2.	Algoritmos de aprendizaje	95
9.2.1.	La tarea	96
9.2.2.	La medida de rendimiento	97
9.2.3.	La experiencia	98
9.3.	El problema de la generalización, overfitting y underfitting	98
9.4.	Construcción de un algoritmo de aprendizaje automático	102
10.	Aprendizaje Profundo	103
10.1.	Introducción a las Redes Neuronales	103
10.1.1.	Estructura básica de una red neuronal	104
10.2.	Optimización basada en el Gradiente	109
10.2.1.	El Gradiente Descendente y sus variantes	111
10.2.2.	Algoritmos de optimización del Gradiente Descendente	113
10.3.	Entrenamiento de Redes Neuronales	114
10.3.1.	Forward Propagation	115
10.3.2.	Algoritmo de Backpropagation	116
10.4.	Funciones de Activación	119
10.4.1.	Propiedades de las funciones de activación	119
10.4.2.	Función de activación sigmoidal	120
10.4.3.	Función ReLU	123
10.5.	Técnicas de Regularización	126
10.5.1.	Regularización de la función de coste	127
10.5.2.	Restricciones sobre los parámetros de la red	130
10.5.3.	Introducción de ruido	132
10.5.4.	Early Stopping	132
10.5.5.	Dropout	134
10.5.6.	Batch normalization	134
11.	Redes Neuronales Convolucionales	137
11.1.	La Operación de Convolución	138
11.2.	Tipos de capas	140
11.2.1.	Capas Convolucionales	141
11.2.2.	Capas de Agrupación	144
11.2.3.	Capas Totalmente Conectadas o Densas	146
11.3.	Arquitecturas	146

Índice general

IV. Clasificación de Posiciones de Ajedrez	149
12. Descripción del problema	151
12.1. Descripción del problema	151
12.2. Posición de Ajedrez	152
12.3. Base de Datos	153
13. Estado del Arte	161
13.1. Estado del Arte	161
14. Metodología	167
14.1. Herramientas Software	167
14.2. Métricas	168
14.3. Selección de los Modelos	169
14.4. Modelo Primera Representación (Representación Tridimensional)	171
14.4.1. Primera Representación	171
14.4.2. Modelo	172
14.5. Modelo Segunda Representación (Representación Unidimensional)	173
14.5.1. Segunda Representación	173
14.5.2. Modelo	173
14.6. Modelo Tercera Representación (Representación Bidimensional)	174
14.6.1. Tercera Representación	174
14.6.2. Modelo	175
14.7. Proceso de Entrenamiento	176
15. Resultados Experimentales y Análisis	179
15.1. Resultados	179
15.2. Análisis	199
V. Conclusiones y Trabajo Futuro	203
16. Conclusiones y Trabajo Futuro	205
16.1. Conclusiones	205
16.2. Trabajo Futuro	206
A. Apéndice A	209
A.1. Aplicación Web	209
A.2. Análisis de Requisitos	209
A.2.1. Requisitos Funcionales	209
A.2.2. Requisitos no funcionales	210
A.3. Diseño	210
A.3.1. Diseño de la Base de Datos	210
A.3.2. Bocetos	211
A.4. Implementación	213
A.4.1. Lenguajes de Programación	213
A.4.2. Software	214
A.4.3. Estructura de la Aplicación Web	214

Índice general

A.5. Vistas	215
A.5.1. Vistas Web	216
A.5.2. Vistas Móvil	218
A.6. Foros	222
B. Apéndice B	223
B.1. Algoritmos	223
B.2. Código	224
B.3. Redacción del documento	224
Bibliografía	225

Agradecimientos

La realización de este documento pone fin a una etapa que comenzó hace cinco años. Durante este período han sido muchas las personas que me han apoyado y animado y ahora me gustaría corresponderles como se merecen.

En primer lugar, quiero agradecer a mi familia todo lo que hacen y han hecho por mí, empezando por mis padres que son las personas que me han permitido ser quien soy y estudiar esta carrera lejos de casa. Para mí sois un ejemplo, y espero que el esfuerzo y el sacrificio que habéis realizado haya tenido recomensa. A mi hermana, por estar siempre ahí, apoyándome en todo lo que he necesitado y sacándome una sonrisa en los malos momentos. A mis abuelos, en especial a mi abuela Fina que falleció hace unos meses, gracias por ser la mejor abuela que un nieto puede tener. A mis padrinos, a mis tíos y a mis primos por sus consejos y su cariño. Muchísimas gracias, os quiero mucho.

También quiero agracederle a mis tutores, Francisco Javier Melero Rus y Francisco Herrera Triguero, su dedicación, sus correcciones y sus consejos.

Por último, me gustaría darle las gracias a mis amigos, tanto a los de la infancia como a los que he conocido a lo largo de la carrera, gracias por todo el apoyo que me habéis dado. De entre todos ellos quiero destacar a Carlota por ser una persona muy especial para mí, por estar siempre conmigo en los buenos y en los malos momentos y por preocuparse tanto por mí.

¡Muchas gracias a todos!

Breve resumen

Palabras clave: ajedrez, ajedrez computacional, motor de ajedrez, posición de ajedrez, representación del tablero, inteligencia artificial, aprendizaje automático, aprendizaje profundo, redes neuronales.

El ajedrez es uno de los juegos de mesa más antiguos que existen y también uno de los más populares. Con el paso de los años y gracias a la inteligencia artificial ha crecido enormemente. El principal propósito de este trabajo es analizar posiciones de ajedrez que han provocado que en el siguiente movimiento se produjera una mala jugada, para ello clasificaremos estas posiciones de acuerdo a una serie de etiquetas. Este análisis podrá servir de ayuda a los jugadores a mejorar su nivel de juego.

En primer lugar presentamos el juego del ajedrez, donde repasamos su historia y explicamos las reglas del juego. Posteriormente, nos adentramos en el ajedrez computacional. En esta parte vemos la historia y la evolución que ha tenido destacando los cambios más relevantes de cada época donde se puede ver la importancia de la inteligencia artificial y de las redes neuronales. Tras esto, estudiamos los componentes de un motor de ajedrez y la forma de trabajar que tienen, y además, destacamos algunos de los motores más importantes en la actualidad.

En la segunda parte presentamos los fundamentos teóricos del trabajo. En particular, estudiamos los fundamentos de la teoría del aprendizaje donde explicamos que no existe un algoritmo de aprendizaje universal gracias al teorema de No Free Lunch. Asimismo, presentamos los fundamentos del caso particular de aprendizaje profundo, junto con los dos modelos principales necesarios para nuestro trabajo, las redes neuronales prealimentadas y las redes neuronales convolucionales. Además, demostramos el teorema de aproximación universal para redes neuronales y descubrimos que cualquier función real entre compactos puede ser aproximada tan cerca como queramos por una de estas redes.

En la parte de clasificación de posiciones de ajedrez presentamos el problema que debemos abordar para analizar posiciones de ajedrez. Hemos diseñado una base de datos de posiciones de ajedrez que cumplen el requisito de que el siguiente movimiento que se produjo fue una mala jugada. Desarrollamos una aplicación web para que la comunidad de ajedrez nos ayudara a etiquetar las posiciones de ajedrez a través de motivos posicionales, pero al contar con poca participación tuvimos que optar por crear una serie de algoritmos que nos permiten extraer las características de una posición y con estas hemos podido etiquetar las posiciones.

Hemos desarrollado un software basado en aprendizaje profundo mediante redes neuronales que aprendió a clasificar posiciones de ajedrez gracias a una serie de etiquetas. Los modelos que hemos empleado utilizan diferentes representaciones del tablero. La primera de ellas consta de siete tableros de tamaño 8×8 y fue extraída de la literatura, la segunda está formada por un vector unidimensional y la tercera se compone de un único tablero de tamaño 8×8 . Por último, hemos expuesto mediante un ejemplo de tres mil posiciones de ajedrez jugadas todas ellas por un mismo jugador con las piezas blancas el beneficio que puede suponer este trabajo en el mundo del ajedrez.

Summary

Keywords: chess, computational chess, chess engine, chess position, chessboard representation, artificial intelligence, machine learning, deep learning, neural networks.

Artificial intelligence is one of the great revolutions of recent times. If we take a look at our daily lives, we probably notice that many manifestations of artificial intelligence have been among us for decades. This technology performs a lot of tasks and intelligent work that we generally do not perceive, but that we need in our day-to-day life.

This area of knowledge has influenced the way chess games are now played at the highest technical level and the popularization of this sport discipline in the world. Today, most grandmasters and chess players use powerful chess engines for the analysis and preparation of their games.

This paper aims to analyze chess positions where in the following move is considered to be bad. For that purpose, we will classify a series of positions based on a set of labels, in total 38. To perform the classification we need to develop a deep learning model that allows us to classify the positions according to their characteristics and subsequently extract from each label those classes that are most repeated. We will also perform a demonstration of its usefulness, in which we will show the patterns we have found in a set of three thousand positions where a bad move occurred in the next move, all played by the same player with white pieces.

The document has been divided into five parts. In the first part we will introduce the work and the proposed objectives. In the second part we will study the main concepts of the game of chess and computational chess. In the third part we will explain the fundamentals of the learning problem. In addition, we will study the basics of the particular case of deep learning, as well as the main models needed for our work, the feedforward neural networks and the convolutional neural networks. In the fourth part we will present the problem we want to address, the analysis of chess positions. In the fifth part we will present our conclusions and possible future work. Additionally, in the appendices we can find the web application and the algorithms developed for the realization of this work.

In the part dedicated to chess we will begin by presenting its history and evolution. Afterwards, we will explain the basic rules of the game as well as its elements and phases. We will also focus on presenting the rules of chess notation where we highlight the FEN notation, as it is the one we have used to develop the web application.

The following three chapters focus on computational chess. We will begin with a chapter dedicated to the history of computational chess, we will see the most relevant periods and milestones of each one and we will highlight the importance of artificial intelligence, in particular, neural networks. In the next chapter we will study the elements of a chess engine: the move generator, the search function and the evaluation function. Once we have studied these components, we will explain the concept of a chess engine and show some of the most important ones nowadays.

Summary

The next part is composed of five chapters where we will present the mathematical part of this work. We will start by reviewing concepts of measurement theory and introduce the concept of learning algorithm in order to present the No Free Lunch theorem. This theorem lets us know that there is no universal learning algorithm that outperforms all other algorithms in all learning tasks. Another important theorem that we have studied is the universal approximation theorem for neural networks, both for sigmoid activation functions and ReLU functions. This theorem states that a single intermediate layer is sufficient to approximate, with arbitrary accuracy, any function with a finite number of discontinuities; and the activation functions of the hidden neurons are nonlinear. Subsequently, we will focus on learning algorithms, where we will expose the generalization problem and explain how a learning algorithm is constructed.

Next, we will introduce the field of machine learning known as deep learning, where we will see the basic structure of a neuron and how it works. We will explain gradient-based optimization and its variants, and we will see the training process used by neural networks. We will also highlight the role of activation functions and explain the importance of regularization techniques in learning algorithms. We will end this part by explaining the fundamentals of convolutional neural networks where we present the convolution operation, the types of layers used and the most commonly used architectures.

In the fourth part of this paper we will present the proposed chess position classification problem. We have designed a database with the characteristics we need. The database is composed of chess positions obtained from games where the players had more than 2200 Elo points. These positions occurred between move twenty and move fifty where in the next move a bad move occurred. From all the positions that fulfill these characteristics, we selected a total of fifty thousand and these will make up our database. Now we must label the positions in order to classify them. The initial idea was to create a web application where the chess community could help us in this task but the result we obtained were not as expected, since we did not have the necessary number of answers to train the learning models. Therefore, we chose to implement a series of algorithms that simulate what we wanted to achieve with the application. In addition to labeling the positions it also provided us with different ways to represent a position on a chessboard. These chessboard representations will be the ones used by the models as input data. In total we have 38 labels divided into two blocks, four of them are common to both sides and the remaining ones are equally divided on each side. We have developed three different models one for each of the representations we have used. The first representation (3D) has been extracted from the literature and is made up by seven boards of size 8×8 , six of them are for the pieces and an additional one to know which side is to move. The second representation is composed by a one-dimensional vector where the seven described boards appear one after the other. This representation was used as an experiment. The last representation is made up by a square list, that is, a two-dimensional 8×8 matrix where the white pieces are represented by positive numbers, the black ones by negative numbers and the empty square by 0. These models were selected using the hold-out method where we divided the dataset into three sets, one to perform training (training dataset), one for validation (validation dataset) and one to test the model with previously unseen data (test dataset).

The results achieved were quite satisfactory for some labels, but for others the results we obtained were not as good as we expected, since some of the labels had very unbalanced classes. The best chessboard representation was the first one, the three-dimensional represen-

tation, even though it did not obtain the best results for certain labels. With the best model of each label we put into practice the usefulness of our work by analyzing three thousand games played by the same player with white pieces. In this way, we were able to extract the patterns that were most repeated in the positions where he made a bad move.

Parte I.

Introducción y objetivos

1. Introducción

1.1. Contextualización

La inteligencia artificial (IA), iniciada después de la Segunda Guerra Mundial, es uno de los campos del conocimiento humano que más ha crecido durante los últimos años. La IA abarca una gran variedad de subcampos, que van desde áreas de propósito general, como el aprendizaje y la percepción, a otras más específicas como la demostración de teoremas matemáticos, la escritura de poesía, el diagnóstico de enfermedades y el juego del ajedrez que es el tema que nos concierne.

Este dominio sintetiza y automatiza tareas que en principio son intelectuales y es, por lo tanto, potencialmente relevante para cualquier ámbito de la actividad intelectual humana. Intenta replicar y desarrollar la inteligencia y sus procesos implícitos a través de computadoras.

El ajedrez ha inspirado el progreso de la inteligencia artificial durante décadas y los desarrollos de la inteligencia artificial para el ajedrez han avanzado más allá del juego, cambiando la forma en la que conviven máquinas y humanos.

La inteligencia artificial es utilizada en el mundo del ajedrez sobre todo para mejorar el nivel de juego de los jugadores y este es el propósito principal de nuestro trabajo, conseguir que los jugadores sean capaces de mejorar aprendiendo de sus errores. El tema a tratar se centra en el análisis de posiciones de ajedrez, en concreto, la clasificación de posiciones de ajedrez gracias a una serie de etiquetas.

Algunos estudios previos realizaron experimentos con jugadores expertos sin la ayuda de la inteligencia artificial. En estos experimentos se mostraban una serie de posiciones y los jugadores debían clasificarlas de acuerdo a ciertos patrones que encontraran en ellas. Estas posiciones eran etiquetadas y agrupadas pero cada jugador tenía sus propios criterios.

Para resolver el problema que vamos a tratar nos apoyaremos en una rama de la inteligencia artificial que es conocida como aprendizaje profundo y haremos uso de las conocidas redes neuronales. El objetivo de nuestro trabajo consiste en clasificar una serie de posiciones en base a unas etiquetas comunes, permitiendo que los jugadores sean capaces de saber reconocer los patrones que aparecen en las posiciones donde cometen una mala jugada. De esta forma, los jugadores podrán centrarse en un conjunto de posiciones con unas mismas características, facilitando así su aprendizaje.

1.2. Descripción del problema

En este trabajo pretendemos analizar posiciones de ajedrez donde en el siguiente movimiento se produjo una mala jugada. Para ello, clasificaremos un conjunto de posiciones en base a un

1. Introducción

grupo de 38 etiquetas. Estas etiquetas son motivos posicionales de ajedrez como pueden ser el número de columnas abiertas, torres dobladas, ciertas estructuras de peones, etc. Para llevar a cabo la clasificación hemos desarrollado una serie de modelos de aprendizaje profundo. También realizaremos una demostración de su utilidad a través de un ejemplo.

En total, contaremos con cincuenta mil posiciones extraídas de la base de datos de *Lichess* [55], las cuales filtraremos según unos criterios que explicaremos en el capítulo 12. Una vez obtenidas las posiciones filtradas necesitamos etiquetarlas, para ello haremos uso de una serie de algoritmos (ver en el apéndice B) que hemos implementado.

Para entrenar los modelos hemos empleado tres formas distintas de representar una posición de ajedrez. La primera de ellas ha sido extraída de la literatura y está formada por siete tableros de tamaño 8×8 , seis tableros para las piezas y uno para saber el turno de movimiento, esta representación es la que hemos llamado tridimensional (matrices tridimensionales $8 \times 8 \times 7$). La segunda representación está formada por un vector unidimensional formado por los siete tableros que acabamos de comentar, uno a continuación del otro, no tiene en cuenta la localidad espacial; esta es la representación unidimensional. La última representación es la representación bidimensional y está formada por un único tablero de tamaño 8×8 donde las piezas y las casillas vacías se codifican con un número. Estas representaciones se detallan en el capítulo 14.

Por cada representación del tablero hemos desarrollado un modelo distinto. Cada uno de ellos ha sido entrenado 38 veces, una vez por etiqueta. Los modelos propuestos que emplean la primera y la tercera representación se basan en redes neuronales convoluciones, mientras que el modelo de la segunda representación está formado únicamente por capas totalmente conectadas.

Estos modelos serán comparados para saber cuál de ellos proporciona mejores resultados. Una vez tengamos el mejor modelo para cada etiqueta mostraremos la utilidad de nuestro trabajo con un conjunto de tres mil posiciones, todas ellas fueron jugadas por un mismo jugador con las piezas blancas. Gracias a nuestros modelos destacaremos los motivos posicionales que más aparecen en las posiciones analizadas donde el jugador comete una mala jugada en el siguiente movimiento. De esta manera, el jugador podrá conocer aspectos de las posiciones donde comete una mala jugada y, por consiguiente, mejorará su nivel de juego.

1.3. Estructura del trabajo

El presente trabajo se encuentra estructurado en 5 partes que, a su vez, se componen de una serie capítulos:

- La primera parte está compuesta por dos capítulos, 1 y 2. En el primero de ellos que es en el que nos encontramos se presenta una breve introducción del trabajo desarrollado y en el segundo capítulo se enumeran los objetivos que debe cumplir.
- La segunda parte se centra en el juego del ajedrez. Esta parte consta de cuatro capítulos, en el primero de ellos, el capítulo 3, se explica la historia y evolución del ajedrez, algunas notaciones importantes y las reglas básicas. Los capítulos 4, 5 y 6 se centran en el ajedrez computacional, en los cuales se detalla la historia del ajedrez computacional, los componentes que tiene un motor de ajedrez, qué se conoce por motor de ajedrez y se describen algunos de los motores más importantes actualmente. Además, se explica

el avance que las redes neuronales han producido en los motores de ajedrez.

- La tercera parte está dedicada al estudio del aprendizaje automático, en particular, al aprendizaje profundo y las redes neuronales artificiales. Además, se estudian dos importantes teoremas, uno sobre aprendizaje y otro acerca de las redes neuronales. En el capítulo 7 se explican conceptos sobre aprendizaje y se presenta el teorema de No Free Lunch de optimización. El capítulo 8 se centra en el teorema de aproximación universal. En el capítulo 9 se explica el concepto de aprendizaje automático y se profundiza en los algoritmos de aprendizaje. En el capítulo 10 se define formalmente el modelo que constituyen las redes neuronales prealimentadas y su procedimiento de entrenamiento. Además, se presentan las funciones de activación y las técnicas de regularización. Esta parte concluye con el capítulo 11 donde se describen las redes neuronales convolucionales y los tipos de capas más comunes que incluyen.
- En la cuarta parte se presenta el principal problema que tratamos de resolver, el cual se describe en el capítulo 12. En el capítulo 13 se recogen los trabajos relacionados y los avances que se han producido en este campo. En el capítulo 14 se detallan los aspectos más técnicos, como el software desarrollado, la métrica empleada, las representaciones consideradas, los modelos propuestos y el entrenamiento realizado. Finalmente, en el capítulo 15 se presentan los resultados alcanzados, un análisis sobre estos y su utilidad.
- La última parte de este trabajo se compone únicamente del capítulo 16 donde se expondrán las conclusiones obtenidas a lo largo del trabajo y las posibles vías a seguir en un futuro.

Además, el presente documento cuenta con dos apéndices. En el primero de ellos, el apéndice A, se presenta la aplicación web que hemos realizado. El propósito de la aplicación es que la comunidad de ajedrez nos ayude a etiquetar las posiciones, es decir, nos eche una mano en señalar los motivos posicionales que aparecen en cada posición. En el apéndice B se describen los algoritmos que hemos necesitado para etiquetar las posiciones, el repositorio donde se puede encontrar el código que se ha desarrollado para la realización del presente documento y la plantilla que se ha utilizado.

1.4. Bibliografía fundamental

Entre la gran variedad de libros y artículos que hemos consultado destacamos aquellos que consideramos que han sido más relevantes para la elaboración del presente documento:

- *Learning from data* [1], escrito por Yaser S. Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin, este libro ha sido la principal referencia usada para la elaboración del capítulo 11.
- *Redes Neuronales & Deep Learning* [4], escrito por Fernando Berzal, este libro ha sido empleado para los capítulos 9 y 10.
- *Approximation by Superpositions of a Sigmoidal function* [21], este artículo se ha empleado para la elaboración del capítulo 8.
- *Deep Learning* [46], este libro escrito por Ian Goodfellow, Yoshua Bengio y Aaron Courville se ha utilizado en la elaboración de los capítulos 9, 10 y 11.

1. Introducción

- *An Overview Of Artificial Neural Networks for Mathematicians* [47], artículo escrito por Leonardo Ferreira Guilhoto el cual se ha empleado para el desarrollo del capítulo 8.
- *Foundations of Machine Learning* [63], libro cuyos autores son Mehryar Mohri, Afshin Rostamizadeh y Ameet Talwalkar que ha sido utilizado en la elaboración del capítulo 10.
- *Understanding Machine Learning* [75], escrito por Shai Shalev-Shwartz y Shai. Ben-David. Este libro ha sido fundamental para la realización de los capítulos 7 y 10.

Para la segunda parte, la parte de ajedrez, hemos hecho uso de algunos artículos, repositorios y sobre todo recursos web entre los que destaca *Chess Programming Wiki* [24].

2. Objetivos

Los principales objetivos planteados en este trabajo son los siguientes:

1. *Ajedrez.* En esta parte, el objetivo fundamental es entender la manera en la que trabajan los motores de ajedrez. Para conseguir este propósito necesitamos:
 - a) Explicar los aspectos más importantes del ajedrez.
 - b) Comprender el funcionamiento de un motor de ajedrez.
 - c) Conocer la importancia que tienen las redes neuronales en los motores de ajedrez.
2. *Matemáticas.* El propósito principal es describir los fundamentos del aprendizaje profundo. Para ello, debemos:
 - a) Estudiar los fundamentos de la teoría del aprendizaje junto con algún resultado relevante.
 - b) Comprender y analizar el funcionamiento de las redes neuronales y, en particular, de las redes neuronales convolucionales.
3. *Informática.* El principal objetivo es implementar un software basado en aprendizaje profundo para clasificar posiciones de ajedrez, el cual nos permite identificar patrones que se repiten antes de que se produzca una mala jugada. Para este propósito, tenemos que conseguir:
 - a) Diseñar una base de datos conformada por posiciones de ajedrez con las características que necesitamos para realizar nuestro estudio.
 - b) Desarrollar una aplicación web para que la comunidad de ajedrez nos ayude en la clasificación de posiciones de ajedrez.
 - c) Codificar y etiquetar las posiciones de ajedrez de la base de datos mediante diferentes representaciones.
 - d) Elaborar modelos de aprendizaje profundo para la clasificación de posiciones de ajedrez y realizar una comparación entre ellos dependiendo de la representación del tablero que hemos empleado.

Estos objetivos han sido cubiertos a lo largo del desarrollo del presente trabajo.

Los objetivos relacionados con el ajedrez se han alcanzado en la segunda parte del trabajo.

- El objetivo 1a) se ha alcanzado en el capítulo 3.
- El objetivo 1b) se ha cubierto en los capítulos 5 y 6. En los cuales se estudian las componentes de un motor de ajedrez y se explican diferentes tipos de motores.
- El objetivo 1c) ha sido alcanzado en los capítulos 4, 5 y 6.

2. Objetivos

Los objetivos de la parte de matemáticas han sido alcanzados en la tercera parte del trabajo.

- Se ha completado el objetivo 2a) en los capítulos 7 y 9 donde se explica el concepto de algoritmo de aprendizaje, cómo se construye un algoritmo de aprendizaje y el problema de la generalización. Además, en el capítulo 7 explicamos que no existe un algoritmo de aprendizaje universal.
- El objetivo 2b) se ha cubierto en los capítulos 8, 10 y 11. En el capítulo 8 se explica un importante teorema para redes neuronales, el teorema de aproximación universal. En el capítulo 10 se explican los fundamentos de las redes neuronales y, en particular, de las redes neuronales prealimentadas y en el capítulo 11 se presentan las redes neuronales convolucionales.

Los objetivos de la parte de informática han sido cubiertos fundamentalmente en la cuarta parte del trabajo.

- El objetivo 3a) se ha cubierto en el capítulo 12 y en la primera sección del apéndice B.
- El objetivo 3b) se presenta en el primer apéndice del presente trabajo A donde se explica en qué consiste y el propósito de nuestra aplicación web.
- El objetivo 3c) se ha alcanzado en los capítulos 12, 14 y en el apéndice B.
- El objetivo 3d) ha sido alcanzado en los capítulos 12, 14 y 15 donde se explica el problema que se pretende resolver, los modelos que se han desarrollado para ello y se realiza una comparativa por etiqueta entre los diferentes modelos.

Parte II.

Fundamentos del Ajedrez y Ajedrez Computacional

En esta parte se expondrán los fundamentos del ajedrez y se presentará el ajedrez computacional. Se explicará en qué consiste el ajedrez, su origen, expansión y evolución, los elementos y fases del juego, los ritmos de juego y las normas de notación más empleadas. En el segundo capítulo se realizará un recorrido por las etapas que ha vivido el ajedrez computacional. En el tercer capítulo se describirán los componentes de un motor de ajedrez. En el último capítulo de esta parte se analizarán algunos de los motores más famosos y potentes actualmente y se concerá la importancia de las redes neuronales en ellos.

3. Fundamentos del Ajedrez

En este capítulo se presentan los fundamentos del ajedrez, se explica qué es el ajedrez, cuál es su origen y cómo ha evolucionado, los elementos y las fases del juego; así como los ritmos de juego que existen. Por último, se muestran algunas normas de notación ajedrecística entre las cuales destaca la notación FEN que ha sido empleada en el desarrollo de la aplicación web (apéndice A).

3.1. ¿Qué es el ajedrez?

El ajedrez es uno de los juegos de mesa más antiguos que existen y también uno de los más populares, es un juego de *suma cero*¹, finito y de *información completa*². El ajedrez es un juego de estrategia que se desarrolla sobre un tablero en el cual se enfrentan dos contrincantes, uno comanda el bando blanco y otro el bando negro. Cada jugador cuenta al inicio con dieciséis piezas que pueden desplazarse respetando ciertas reglas por un tablero de 64 casillas también conocidas como escaques.

El tablero sobre el que se juega está cuadriculado en 8x8 casillas alternadas en dos colores, blanco y negro, que constituyen las sesenta y cuatro posibles posiciones de las piezas para el desarrollo del juego. Al comienzo del juego cada jugador dispone de dieciséis piezas y estas son, un rey, una dama, dos torres, dos caballos, dos alfiles y ocho peones. Comienza a jugar quien lleve las piezas blancas, lo que le concede una ventaja pequeña pero esencial en los niveles altos de competición al jugador que comande este bando, por eso se sortea antes de comenzar la partida el bando con el que juega cada jugador. A partir de entonces ambos jugadores se turnan para mover alguna de sus piezas.

Cada jugador intentará obtener ciertas ventajas en la posición, siendo el objetivo del juego conseguir derrocar al rey del oponente. Para ello es necesario amenazar la casilla que ocupa el rey con alguna de las piezas propias sin que el otro jugador pueda proteger a su rey, ya sea interponiendo una pieza entre su rey y la pieza rival que lo amenaza, mover su rey a una casilla libre o capturar la pieza que lo amenaza. Si esto se produce estamos en una situación de jaque mate y concluye la partida. Otras posibilidades de victoria son que el rival abandone o que se quede sin tiempo en el reloj. Hay otro resultado posible en el ajedrez además de la victoria o la derrota que es el empate, o también llamado tablas, que se produce por común acuerdo, cuando a ninguno le quedan suficientes piezas para infiligrar jaque mate, si se repite tres veces la misma posición de todas las piezas en el tablero o cuando un jugador no puede realizar en su turno ningún movimiento reglamentario pero el rey no se encuentra en jaque, esto son tablas por ahogado. Otra posibilidad de tablas es la regla de los 50 movimientos, si en los últimos cincuenta movimientos consecutivos de cada bando no se produce captura de

¹En teoría de juegos no cooperativos, un juego de suma cero es aquel en el que las metas de los competidores son contrarias, la ganancia o pérdida de un participante se equilibra con exactitud con las pérdidas o ganancias de los otros participantes.

²En teoría de juegos, un juego de información completa es aquel en el que no hay información oculta, se conoce la situación de todos los jugadores.

3. Fundamentos del Ajedrez

pieza o movimiento de peón la partida termina en tablas si uno de los jugadores las reclama.

Hoy en día el ajedrez es considerado por el Comité Olímpico Internacional un deporte y las competiciones internacionales son reguladas por la *FIDE*³. 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.

3.2. Origen del Ajedrez

El origen del ajedrez [38] no es del todo claro y sigue siendo una cuestión de debate entre los historiadores. Hay una variedad de leyendas, historias y conjeturas empezando desde la disputa sobre el lugar y terminando con la disputa de cuándo empezó la historia del ajedrez que siguen sin resolverse.

Todos llegan a la conclusión de que la invención del ajedrez no se atribuye a una persona, puesto que el juego es demasiado complejo para que una sola mente humana haya podido ser capaz de crear todas sus reglas. En primer lugar vamos a presentar una de las leyendas más famosas que existen sobre el origen del ajedrez.

3.2.1. Leyenda del origen

Cuenta la leyenda que hace mucho tiempo reinaba en cierta parte de la antigua India un rey llamado Sheram. En una de las batallas en las que participó su ejército perdió a su hijo y toda la felicidad. Sus más cercanos consejeros y ministros se esforzaban por animarlo pero todo era en vano.

Un día apareció en su corte un sabio llamado Sissa que le presentó un juego de guerra que lo animaría. Este juego se jugaba sobre un tablero sobre un hermoso tablero de madera con sesenta y cuatro casillas y treinta y dos figuras. Tras explicarle las reglas el rey comenzó a jugar y se sintió maravillado.

Un día, en una de sus partidas, vio cómo la posición de las piezas representaba la batalla en la que su hijo perdió la vida. El sabio le indicó que en esa posición la victoria se conseguiría realizando un sacrificio con uno de los visires (conocidos actualmente como alfiles), y le comentó: “A veces el sacrificio de una pieza importante es necesario para conseguir la victoria final”. El rey comprendió el sacrificio que realizó su hijo y agradecido de que por fin alguien hubiera conseguido distraerlo y abrirle los ojos le ofreció a Sissa cualquier cosa que este quisiera. Tras mucho insistir Sissa le pidió a cambio de su juego un grano de trigo en la primera casilla del juego, dos en la segunda, cuatro en la tercera y así sucesivamente.

En principio, al rey le pareció una ofensa pedir tan mísera recompensa y le pidió a sus ayudantes que calcularan el número total de granos de trigo y se los dieran a Sissa. Después de calcular el número de granos que pedía el sabio durante horas, los ayudantes se acercaron y le comunicaron al rey que no había suficiente trigo para pagar la deuda. La cantidad de granos de trigo equivalía a:

³Acrónimo de la Federación Internacional de Ajedrez, una organización internacional que conecta las diversas federaciones nacionales de ajedrez.

$$T_{64} = 1 + 2 + 4 + \dots + 9.223.372.036.854.775.808 = \sum_{i=0}^{63} 2^i = 2^{64} - 1$$

donde T_{64} corresponde al número total de granos que pidió Sissa. Esta cantidad ascendía a un total de 18.446.744.073.709.551.615 granos de trigo. El rey se quedó de piedra pero el sabio satisfecho por haber conseguido que el rey volviera a estar feliz y por la lección matemática que había dado renunció al presente.

3.2.2. India, Chaturanga

La teoría más extendida es que el ajedrez tiene su origen en la India, concretamente en el Valle del Indo durante el Imperio Gupta, y data alrededor del siglo VI d.C. aunque la referencia más antigua se encuentra en el *Mahábhárata*⁴, escrito alrededor del 500 a.C. Esta teoría se apoya tanto en los primeros registros literarios persas como en el análisis de la etimología de las palabras utilizadas en el juego y su coevolución con el ajedrez.

La mayoría de los expertos defienden esta teoría y concuerdan en que el ancestro más antiguo del ajedrez es el Chaturanga como consecuencia del trabajo de Harold James Ruthven Murray que en su obra *A History of chess* [64] defendía que el ajedrez tiene su origen en la India. Alex Kraaijeweld que realizó en el año 2000 un estudio en el que analizó 40 variantes antiguas y modernas del juego llegó a la misma conclusión, que el ajedrez procedía de un antiguo juego llamado Chaturanga.

En sus orígenes, el juego se conoció como Chaturanga, o juego del ejército, cuyo nombre significa cuatro divisiones en referencia a las cuatro piezas que simbolizan las unidades del ejército indio: elefantes, carros de guerra, caballería e infantería.

El Chaturanga se jugaba en un tablero 8x8 llamado ashtāpada donde las casillas eran del mismo color y algunas de ellas se encontraban marcadas, aunque el significado de estas marcas se desconoce.

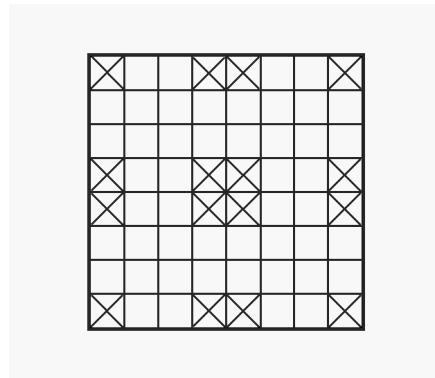


Figura 3.1.: Tablero del Chaturanga. [15]

Existían dos variantes del juego, una para cuatro jugadores y una para dos. En sus inicios el juego era jugado por cuatro personas que se divían en los bandos negro y amarillo que eran

⁴Extenso texto épico-mitológico de la India.

3. Fundamentos del Ajedrez

aliados y luchaban contra los bandos verde y rojo, que también jugaban de compañeros. La salida correspondía al jugador que comandaba las piezas verdes.

Las piezas eran:

- Un *Rajá*, nombre que se le daba a la pieza del rey y que se movía un paso en cualquier dirección.
- Un *Elefante*, que andaba en todas direcciones y tantas casillas como quisiera el jugador.
- Un *Caballo*, que se movía como el nuestro
- Un *Barco*, similar a nuestro alfil.
- Cuatro *Peones*, que van hacia delante paso a paso.

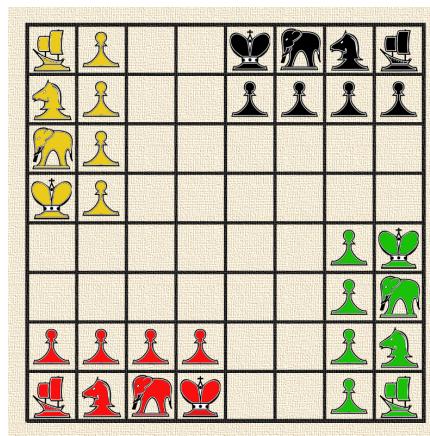


Figura 3.2.: Disposición inicial de una partida del juego Chaturanga para cuatro jugadores.
[33]

En la variante de dos jugadores cada bando contaba con un rey, un general o visir (antecesor de la dama), ocho infantes, dos elefantes (los dos alfiles), dos caballos y dos carros (las actuales torres), siguiendo el esquema del ajedrez actual. El chaturanga según Van der Linde se jugaba con dados y las reglas exactas del juego se desconocen pero se creen que son las mismas que las del juego persa *shatranj*⁵, sucesor del Chaturanga.

3.2.3. China

Otra idea que se baraja es que el ajedrez se inventara en China. La leyenda dice que el ajedrez fue inventado por el comandante Hán Xin en el año 200 a.C. que inventó un juego para representar una batalla particular.

Tras la batalla, el juego desapareció durante varios siglos y reapareció en el siglo VII d.C. con nuevas reglas. El juego se hizo popular bajo el nombre de *Xiang Qi*, que significa el Juego del Elefante. Este juego era muy distinto al ajedrez que conocemos hoy en día, ya que contaba con otras piezas, otro tablero e incluso otras reglas.

⁵También conocido como shatranji es una forma antigua de ajedrez, a partir del cual se ha desarrollado paulatinamente el ajedrez moderno.

Según este origen, el Xiang Qi viajó de China a India y posteriormente a Persia donde fue lentamente modificado en el ajedrez con las piezas y el tablero de sesenta y cuatro casillas que conocemos actualmente.

3.3. Expansión y Evolución

El Chaturanga se expandió desde la India a través de las rutas comerciales como la llamada Ruta de la Seda donde llegó a Persia, y desde allí al Imperio Bizantino, extendiéndose posteriormente por toda Asia. Al entrar en contacto con la civilización persa se transformó en el Chatrang. En el Chatrang los bandos aliados se fundieron en un solo ejército y el rey sobrante se convirtió en el visir o primer ministro (que a la postre sería la futura dama) y que movía una casilla en diagonal. Apareció el tablero bicolor, desapareció el uso de los dados y tanto los elefantes (alfiles) como los carros (torres) cambiaron su ubicación inicial con respecto a la del Chaturanga.

Más tarde, el Chatrang persa llegó al mundo árabe, quienes lo adoptaron bajo el nombre de shatranj. Los árabes estudiaron y analizaron en profundidad los mecanismos del juego, desarrollaron el sistema de notación algebraica y escribieron numerosos tratados sobre ajedrez.

El juego fue introducido en Occidente por los musulmanes. Llegó a Europa alrededor del siglo X, a través de la península ibérica donde llegó a ser muy popular, a finales del siglo XI alcanzó todo el continente europeo. El visir pasó a llamarse Alferza y cuando un peón coronaba solo podía cambiarse por esta pieza. El enroque no existía todavía y las piezas en lo que respecta a sus movimientos no sufrieron modificación alguna.

Durante la Edad Media se escribieron diversas obras sobre el ajedrez, sobre todo por parte de los judíos donde se establecían sus normas y reglas. Con respecto a Europa, España e Italia eran los países donde más se practicaba y se jugaba con las normas árabes.

En el año 1283, el rey Alfonso X el Sabio escribió el *Libro de los juegos* o también llamado *Libro del ajedrez, dados y tablas* [42], la obra más destacada sobre el ajedrez de la Edad Media. El manuscrito original se encuentra en la biblioteca del Monasterio de El Escorial, en Madrid. En esta época el ajedrez adquiere el aspecto medieval de hoy en día, con el rey y la dama (aunque todavía se llamaba Alferza), los obispos (posteriormente alfils), los caballeros y las torres. En lo que respecta al movimiento, el movimiento de la dama y el alfil eran muy diferentes a los actuales, ya que la dama solo podía moverse a una casilla adyacente en diagonal, hacia delante o hacia atrás y el alfil podía saltar. El enroque seguía sin aparecer.

A finales del siglo XV se realizaron importantes cambios en las reglas del ajedrez. Estos cambios se produjeron en España, se cree que en Valencia o en Salamanca aunque no está del todo claro. Las modificaciones efectuadas buscaban otorgar movilidad a las piezas para agilizar las aperturas y además potenciar la importancia de las mismas. Los peones pueden avanzar dos casillas desde la posición original, el alfil pasó a desplazarse como en el ajedrez moderno, de manera oblicua a lo largo de las casillas del color en que se encontraba. La pieza que sufrió más modificaciones fue la dama que pasó a ser la pieza más poderosa del tablero pudiendo moverse como la torre y los dos alfils.

En el año 1575 se celebró en El Escorial (Madrid, España) lo que los historiadores conocen como el primer campeonato mundial de ajedrez. El organizador de dicho evento fue Felipe

3. Fundamentos del Ajedrez

II que invitó al extremeño Ruy López, creador de la apertura española, al granadino Alfonso Cerón y a dos de los mejores ajedrecistas italianos, Leonardo da Cutri *Il Puttino* y Paolo Boi *el Siracusano*. El desenlace del toreno fue un anticipo del ocaso imperial español, pues Ruy López, hasta entonces indiscutido, cayó derrotado en la final frente a Leonardo da Cutri quien fue el vencedor del torneo.

Entre los siglos XVII y XIX el ajedrez se consolidó como el juego predilecto de los intelectuales. A comienzos del siglo XVIII, el panorama ajedrecístico estaba dominado por Francia e Inglaterra. Surgieron grandes jugadores como François André Danican, más conocido como Philidor que es considerado el primer gran teórico de la materia y es autor de la obra *Los peones son el alma del ajedrez*, que reconoce el carácter fundamental de los peones y establece las primeras bases posicionales del juego. El ajedrez fue cogiendo peso y se convirtió en uno de los pasatiempos favoritos de la aristocracia en Europa y solía estar muy presente en las cortes reales.

A partir de la segunda mitad del siglo XIX, los maestros comenzaron a acudir a los primeros torneos. En el año 1862 se organiza en Londres un torneo donde existe por primera vez en la historia la limitación de tiempo de juego mediante un reloj de arena. En el año 1886 Wilhelm Steinitz se proclamó primer campeón mundial de la historia del ajedrez al ganar en Estados Unidos al polaco Johannes Zukertort.

El gran maestro Wilhelm Steinitz está considerado como el primer ajedrecista que entendió la verdadera dimensión del juego y el que abrió el camino hacia su estudio sistemático. Los rasgos distintivos de su juego eran el uso del rey como pieza activa, la defensa y una habilidad extraordinaria en el uso de los peones. La discusión acerca de la efectividad o no de su método permitió seguir desarrollando el juego a principios del siglo XX. Los siguientes grandes ajedrecistas seguirían su estela y algunos de los grandes maestros que le sucedieron en el trono de campeón del mundo fueron el alemán Emanuel Lasker siendo el segundo campeón mundial, posteriormente el cubano José Raúl Capablanca, y así una lista de carismáticos campeones, en total veinte, algunos de los cuales son: Boris Spassky, Bobby Fischer, Anatoly Karpov, Garry Kaspárov considerado el mejor jugador de ajedrez de todos los tiempos por muchos y el actual campeón del mundo Magnus Carlsen.

En el siglo XX, con el surgimiento de las bases de datos, los motores de ajedrez y los diversos métodos para llevar a cabo una preparación estratégica confortable y eficiente produjeron una revolución en el mundo del ajedrez. En el año 1997, el superordenador Deep Blue [10], construido por IBM, derrotaba al entonces campeón mundial Garry Kaspárov. Esta victoria dio paso a la nueva era, la era tecnológica y digitalizada en el mundo del ajedrez. Hoy en día existen diferentes motores de ajedrez, algunos de los cuales son: Stockfish, Ethereal, Leela Chess Zero, Nemorino de los cuales hablaremos en el siguiente capítulo. También existen multitud de portales y plataformas de ajedrez, juegos en línea y diversas modalidades en ellos para que cada jugador pueda evolucionar y mejorar en su juego.

3.4. Elementos del juego

En esta sección vamos a exponer los elementos necesarios para poder jugar al ajedrez ([3], [24]), son necesarias las treinta y dos piezas, dieciséis por bando, el tablero de ajedrez y conocer las reglas del juego. Opcionalmente puede utilizarse un reloj de ajedrez, siendo imprescindible en competiciones.

3.4.1. Las piezas

Para distinguir un bando de otro, las piezas de cada jugador son de distinto color, uno dirige las piezas claras, llamadas blancas, y el otro las de color oscuro, las negras. Cada jugador tiene dieciséis piezas de seis tipos distintos. Los tipos de piezas de los que dispone cada jugador son: un rey, una dama, dos torres, dos alfiles, dos caballos y ocho peones.

Cada pieza se mueve en el tablero de una forma particular:

- El *Rey* puede moverse en cualquier dirección ya sea vertical, horizontal o en diagonal avanzando o retrocediendo una sola casilla (excepto en el enroque, en el cual se mueve 2 o 3 casillas).

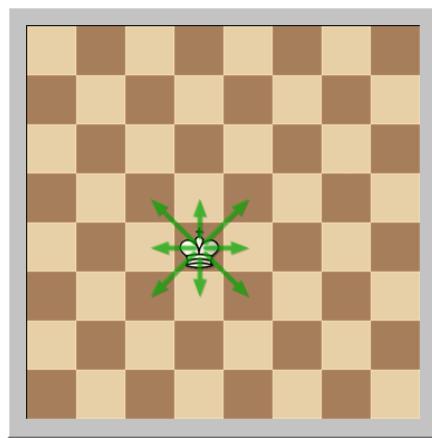


Figura 3.3.: Posibles movimientos del Rey. [2]

- La *Dama* también puede moverse en cualquier dirección ya sea avanzando o retrocediendo en el tablero el número de casillas que desee hasta topar con el borde del tablero o con otra pieza.

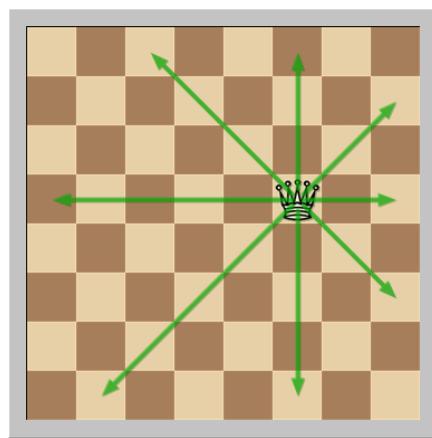


Figura 3.4.: Posibles movimientos de la Dama. [2]

3. Fundamentos del Ajedrez

- La *Torre* solamente puede moverse en dirección horizontal o vertical no en diagonal, hasta que se encuentra con el borde del tablero o con otra pieza.

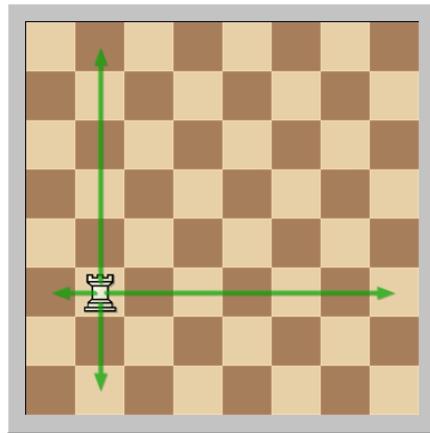


Figura 3.5.: Posibles movimientos de la Torre. [2]

- El *Alfil* solo se puede mover en dirección diagonal tantas casillas como desee hasta topar con el borde o con otra pieza.

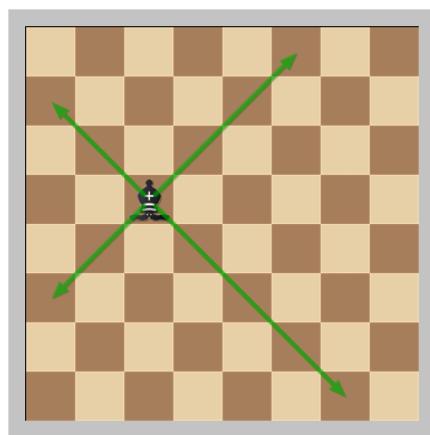


Figura 3.6.: Posibles movimientos del Alfil. [2]

3.4. Elementos del juego

- El *Caballo* se puede mover avanzando dos casillas en vertical y una en horizontal o viceversa, realizando un movimiento en “L” y es la única pieza que puede saltar por encima del resto de piezas.

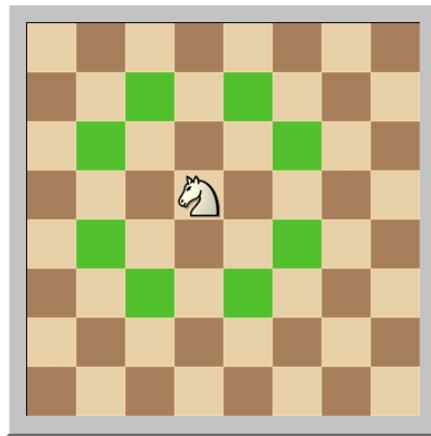


Figura 3.7.: Posibles movimientos del Caballo. [2]

- El *Peón* puede moverse una o dos casillas en su primer movimiento, una vez haya sido adelantado solo puede avanzar una casilla. A diferencia del resto de piezas no puede retroceder y tampoco puede capturar a piezas contrarias que se encuentran en la misma dirección, el peón puede capturar a las piezas que se encuentran a una casilla en diagonal respecto a él, no a las que estén delante o al lado excepto en la captura al paso.

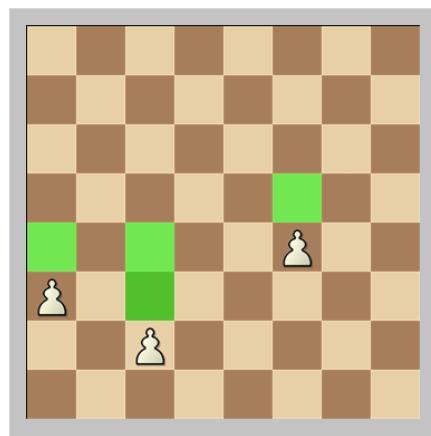


Figura 3.8.: Posibles movimientos del Peón. [2]

Dependiendo del contexto, la palabra pieza puede adoptar tres significados, puede referirse a cualquiera de las treinta y dos figuras, solamente al rey, la dama, la torre, el alfil y el caballo para diferenciarlos de los peones o solo a una pieza menor, el alfil o el caballo.

Debido a que el bando que comanda las piezas blancas posee la ventaja de realizar el primer movimiento, el color de las piezas se asigna por sorteo. En los torneos se intenta que cada

3. Fundamentos del Ajedrez

jugador juegue el mismo número de partidas con las piezas blancas que con las negras, esto se lleva a cabo mediante la elaboración de un *fixture*⁶ con la alternancia de colores en cada ronda, o que el torneo tenga un número par de rondas.

El modelo de piezas que se utiliza en competiciones recibe el nombre de modelo *Staunton*, fue diseñado en 1849 por *Nathaniel Cook* y fue llamado así por *Howard Staunton* campeón inglés del siglo XIX.

3.4.2. El tablero

El tablero de ajedrez es un cuadrado subdividido en sesenta y cuatro casillas o escaques cuadradas e iguales (8×8), alternadas de color claro u oscuro. Cada jugador se sitúa frente a su oponente y el tablero debe estar colocado de manera que cada jugador tenga una casilla blanca en su esquina inferior derecha.

Los elementos básicos del tablero son:

- *Fila*: cada una de las ocho líneas de ocho casillas que se forman alineando estas horizontalmente respecto a los jugadores. Son nombradas con números del 1 al 8, siendo la primera fila la más cercana al jugador que lleva las piezas blancas.
- *Columna*: cada una de las ocho líneas de ocho casillas que se forman alineando estas de forma vertical respecto a los jugadores. Son nombradas con letras minúsculas partiendo desde la letra *a* hasta la letra *h*, siendo la primera columna la que se encuentra más a la izquierda con respecto al bando blanco.
- *Diagonal*: cada una de las dieciséis líneas que se forman agrupando las casillas diagonalmente. Las dos diagonales mayores tienen ocho casillas.
- *Centro*: los cuatro escaques centrales, *d4, e4, d5, e5*. A veces se incluyen en el centro del tablero los doce escaques que rodean a esos cuatro.
- *Esquinas*: cada una de las cuatro casillas ubicadas en las esquinas del tablero.
- *Bordes*: las dos filas y las dos columnas que se sitúan en los extremos del tablero, es decir, las filas 1 y 8 y las dos columnas *a* y *h*.

Un tablero de ajedrez puede contener los números y las letras para identificar las filas, las columnas y las casillas, ya que la notación oficial es la notación algebraica y es la que se emplea de manera más frecuente a la hora de registrar el desarrollo de partidas de ajedrez.

3.4.3. El reloj

El reloj de ajedrez consta de un doble cronómetro que mide el tiempo del que dispone cada jugador para realizar sus movimientos. El reloj del jugador que tiene el turno está en marcha y el de su oponente se encuentra detenido, cuando el jugador finaliza el turno detiene su reloj y pone en marcha el reloj del contrario.

Los relojes de ajedrez analógicos tienen un funcionamiento mecánico y contienen en su parte posterior una tuerca la cual debe moverse en un sentido para darle cuerda al reloj antes de comenzar la partida. Este tipo de relojes también poseen un elemento llamado bandera, el

⁶Calendario de encuentros para un torneo

cual es sostenido por el minutero durante los últimos tres minutos del tiempo de juego de cada jugador. Si se excede ese tiempo, el minutero deja caer la bandera y el jugador al que se le haya caído la bandera pierde automáticamente la partida si no ha completado las jugadas establecidas o excede el límite de tiempo establecido.

En la actualidad, los relojes de ajedrez más utilizados son los digitales. Estos están basados en un funcionamiento electrónico, mediante pilas, permitiendo configurar diferentes ritmos de juego, como el sistema *Fischer* (incrementa el tiempo del reloj en varios segundos por cada movimiento) o el sistema *Bronstein* (igual que el anterior pero sin sobrepasar el tiempo inicial asignado); así como determinar con exactitud qué jugador ha agotado su tiempo primero.

3.4.4. Reglas generales

3.4.4.1. Disposición inicial de las piezas

El tablero como mencionamos previamente debe estar de forma que la casilla en la esquina inferior derecha de ambos jugadores sea blanca. Las piezas en ajedrez tienen una ubicación simétrica que pasamos a explicar.

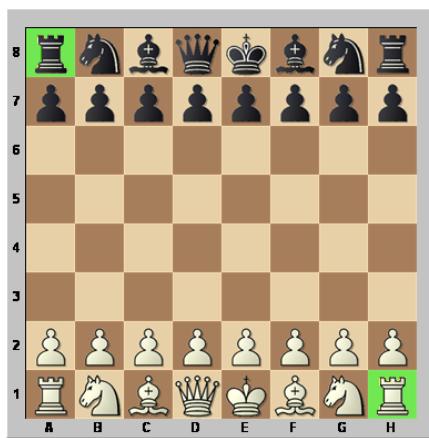


Figura 3.9.: Posición inicial de una partida de ajedrez. [2]

- Cada bando comienza la partida con ocho peones. Los peones blancos se sitúan en la segunda fila y los peones negros en la séptima fila.
- Cada jugador empieza la partida con dos torres en su poder, una en el flanco de rey y otra en el flanco de dama. Al inicio, las torres se sitúan en las esquinas del tablero, las torres blancas en la primera fila, en las casillas a1 y h1 y las torres negras en la octava fila, en a8 y h8.
- Al comienzo de la partida cada jugador dispone de dos caballos, uno situado en el flanco de rey y otro en el flanco de dama, situados al lado de las torres. En la posición inicial, los caballos blancos ocupan las casillas b1 y g1 y los negros las casillas b8 y g8.
- Al inicio de la partida cada jugador dispone de dos alfiles, uno de casillas blancas y otro de casillas. En la posición inicial, los alfiles blancos se sitúan en c1 y f1, mientras que los negros en c8 y f8.

3. Fundamentos del Ajedrez

- Antes de comenzar la partida cada jugador cuenta con una dama. La dama blanca se posiciona en $d1$ y la dama negra en la casilla $d8$.
- El rey, la pieza más importante del ajedrez se coloca al inicio de la partida en la casilla $e1$ si es el rey blanco y el rey del bando negro se coloca en $e8$.

3.4.4.2. Consideraciones generales

- El jugador que lleva las piezas blancas es el primero en mover, lo que supone una pequeña ventaja para este bando.
- Los jugadores mueven alternadamente una de sus piezas, a excepción del caso del enroque en el que se mueven dos piezas, el rey y una de las torres.
- Si una casilla se encuentra ocupada por una pieza no se podrá ocupar por otra del mismo bando. En cambio, sí es posible ocupar un escaque ocupado por una pieza adversaria mediante la captura de esa pieza. Esto se denomina capturar la pieza contraria y la pieza de esa casilla se retira y se sustituye por la pieza que capture.
- Cada pieza tiene una manera particular de moverse que hemos explicado previamente en la subsección 3.4.1. Existen movimientos especiales en ajedrez los cuales son, la captura al paso del peón y el enroque que puede ser corto o largo, además de la coronación.
 - *Enroque:* El rey y una torre se pueden mover en un movimiento simultáneo. El rey se mueve dos casillas hacia la torre y esta se coloca en la casilla adyacente al rey por el lado contrario. Para poder realizar este movimiento son necesarias unas serie de condiciones: tanto el rey como la torre no se han movido en toda la partida, todas las casillas entre el rey y la torre deben estar vacías, el rey no se encuentra en jaque y el rey debe moverse a una casilla en la que no puede ser atacado por ninguna pieza contraria.
 - *Captura al paso:* este movimiento solo puede realizarse con peones y no es obligatorio. La situación se produce cuando un peón es avanzado dos escaques desde su casilla inicial y queda en la casilla al lado de un peón contrario. Este último puede capturar el peón en sentido horizontal, a derecha e izquierda, tal como lo haría diagonalmente. La situación resultante es la misma si el peón hubiera movido una sola casilla y el otro lo hubiera capturado.
 - *Coronación:* cuando un jugador consigue que uno de sus peones llegue a la última fila puede pedir cambiar el peón por la pieza que más le convenga, a excepción de otro peón o el rey.
- Las únicas piezas que pueden saltar son los caballos o también las torres pero solamente en el caso excepcional del enroque que saltan por encima del rey. Por saltar se entiende que una pieza pase por encima de otra pieza.
- El rey adversario nunca se puede capturar. Si al realizar un movimiento el rey contrario pasa a estar amenazado, es decir, en riesgo de captura, se dice que está en jaque. En esta situación el jugador debe realizar cualquier movimiento que saque al rey del jaque. Si el rey está en jaque y no existe un movimiento legal que permita sacar al rey del jaque llegamos a una situación de jaque mate y finaliza la partida. El jugador que tenga

el rey amenazado de captura es el que pierde la partida.

3.5. Fases del juego

Cualquier persona que se inicia en el ajedrez comienza a mover sus piezas sin ningún sentido aparente. En el ajedrez se definen una serie de etapas durante la partida. Para poder mejorar es fundamental conocer cuáles son las diferentes partes del juego y como enfrentarse a cada una de ellas.

Una partida de ajedrez se puede dividir en tres fases bien diferenciadas aunque una vez se profundiza es posible que estas fases se subdividan en otras subfases. Las tres fases generales en las que se divide una partida de ajedrez son: apertura, medio juego y final, cada una con unas características particulares.

3.5.1. Apertura

La apertura es la primera fase del juego. Esta fase consiste en sacar y preparar nuestra piezas sin dejar debilidades en nuestra posición para empezar un ataque al bando contrario. Esta fase del juego está muy estudiada y existen numerosas aperturas como por ejemplo la apertura española, la apertura italiana, la apertura francesa, etc, y estas tienen multitud de variantes.

Vamos a ver ahora cuáles son los principios del desarrollo:

- 1. Sacar los peones centrales para luchar por el centro.
- 2. Desarrollar las piezas menores, estas son los caballos y los alfiles.
- 3. Enrocarse para poner bajo seguridad al rey.
- 4. Dar un paso con la dama a la fila dos si jugamos con las piezas blancas o a la fila siete si jugamos con negras para conectar las torres.
- 5. Llevar las torres al centro.

Además de estos cinco pasos esenciales para la apertura debemos tener en cuenta que no debemos mover la misma pieza más de una vez en la apertura, tampoco debemos sacar la dama muy pronto. También debemos completar nuestro desarrollo de los cinco puntos anteriores antes de lanzar un ataque y, en general, se considera preferible dar prioridad al desarrollo de las piezas menores en vez de a los peones a excepción de los dos peones centrales.

3.5.2. Medio Juego

Una vez hemos realizado los pasos anteriores, hemos completado los principios del desarrollo y pasamos al medio juego. Actualmente, se tarda poco tiempo en llegar a esta fase de la partida ya que existe una gran cantidad de teoría sobre la apertura. Una vez nos hemos adentrado en esta fase es momento de parar, analizar la posición actual y pensar bien las jugadas a realizar. Durante esta fase del juego tendremos que pensar una estrategia para conseguir

3. Fundamentos del Ajedrez

algún tipo de ventaja, ya sea posicional o en cuanto a material y para ello definiremos una táctica. Nuestras prioridades serán:

- Controlar el centro.
- Controlar las columnas o diagonales abiertas o semiabiertas.
- Obtener ventaja en el desarrollo.
- Obtener ventaja de espacio.
- Tener mejor estructura de peones.
- Tener el rey mejor protegido.
- Intentar meter piezas en la mitad del tablero del oponente.

3.5.3. Final

El final es la última fase de la partida y es la etapa posterior al medio juego. Esta fase de la partida es la determinante, ya que es donde se decide el ganador. En esta fase es primordial la técnica. Con la posición conseguida y el material conservado durante la fase anterior deberemos lanzar un ataque contra el rey enemigo con el fin de destruir su defensa y conseguir ganar la partida. En esta fase algún bando dará jaque mate y obtendrá la victoria, se llegarán a posiciones de tablas o ahogado, o algún jugador abandonará la partida.

3.6. Ritmos de juego

Hacia finales del siglo XIX comienza a haber un problema en el devenir de las partidas de ajedrez, ya que había jugadores que tardaban demasiado en reflexionar sus jugadas y así es como nacen los controles de tiempo en ajedrez. En torneos oficiales el tiempo del que dispone cada jugador depende del tipo de torneo y lo establece la organización del torneo. Un jugador puede disponer de cierto tiempo para toda la partida, o para alcanzar un número de jugadas establecidas de antemano; también puede recibir o no un incremento de tiempo por cada jugada. Cuando un jugador agota el tiempo establecido o no es capaz de realizar el número de jugadas asignadas pierde la partida.

Las modalidades de las partidas de ajedrez según su duración pueden ser:

- *Ritmo clásico*: este es el ritmo de juego más empleado a nivel magistral el cual también se denomina ritmo normal o lento. Cada jugador tiene un tiempo superior a los 60 minutos para reflexionar sobre sus jugadas en toda la partida. En torneos de élite el control de tiempo se asigna para un cierto número de jugadas más un tiempo para el final de la partida y se deben anotar los movimientos realizados en la partida. Algunos de los formatos son:
 - 2 horas y media para los primeros 40 movimientos seguido de 20 movimientos en 1 hora, y luego 30 minutos de muerte súbita, por lo que una partida puede durar hasta 7 horas.
 - 120 minutos por lado para los primeros 40 movimientos, 60 minutos para los siguientes 20 movimientos y 15 minutos para el resto del juego, con un incremento

de 30 segundos por movimiento a partir del movimiento 61.

- 2 horas para 45 o 50 jugadas más 1 hora para 25 o 30 jugadas.
- 90 minutos para los primeros 40 movimientos seguidos de 30 minutos para el resto de la partida y 30 segundos de aumento por movimiento.
- *Ajedrez rápido o activo*: cada jugador dispone de un tiempo fijo de entre 10 y 60 minutos. En este tipo de partidas ni en los ritmos más rápidos a este, los jugadores no están obligados a anotar las jugadas.
- *Ajedrez blitz o relámpago*: las partidas tienen una duración de menos de 10 minutos. Las normas son las mismas que en el ajedrez activo.
- *Ajedrez bala*: todas las jugadas deben efectuarse en un tiempo fijo, menos de 3 minutos para cada jugador. Comúnmente, cada jugador dispone de un solo minuto para toda la partida. Debido a su naturaleza muy rápida, este modo se practica sobre todo en línea.

En España a las partidas que se juegan sin reloj se las conoce popularmente como partidas amistosas o partidas de café.

3.7. Normas de notación ajedrecística

Con el objetivo de registrar las partidas con propósitos documentales existen varios sistemas de notación de partidas de ajedrez ([49], [16]). El sistema más utilizado y el recomendado por la *FIDE* es el sistema algebraico. También existen otros métodos como la notación descriptiva aunque actualmente está en desuso, la notación numérica para los juegos por correspondencia o el sistema de Forsyth-Edwards (FEN) empleado para registrar posiciones fijas del juego.

3.7.1. Notación Algebraica

La notación algebraica por su sencillez es la que actualmente más se utiliza. Vamos a explicar en esta sección las normas generales de esta notación, para ello empezaremos por explicar la codificación del tablero.

- Las filas del tablero se nombran con los números del 1 al 8. Las piezas blancas ocupan al comienzo las dos primeras filas y las negras las dos últimas.
- las columnas del tablero se nombran con las letras minúsculas desde la *a* hasta la *h* comenzando por la izquierda del jugador que porta las blancas.
- Las casillas reciben el nombre de la columna y la fila correspondientes, por ejemplo: *b5*, *e4*, *d7*, etc.
- Cada pieza es identificada mediante su letra inicial en mayúscula menos el peón que no requiere letra asignada. Dependiendo del idioma varía la letra asociada a las piezas.

3. Fundamentos del Ajedrez

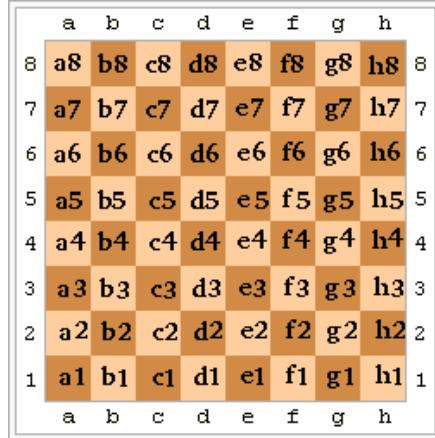


Figura 3.10.: Tablero de ajedrez con notación algebraica. [88]

Las normas generales de la notación algebraica en lo que respecta a los movimientos son:

- Una jugada se escribe con la inicial de la pieza y la casilla a la que se mueve, con la excepción del peón que solo se menciona la casilla a la cual se mueve. Por ejemplo, *Ce4* representa la jugada de un caballo que se mueve a la casilla *e4*, *d5* simboliza la jugada de un peón de la columna *d* a la quinta fila.
- La captura de una pieza se indica con una x entre la letra inicial de la pieza o la columna del peón y la casilla donde estaba la pieza que ha sido capturada. Como ejemplo *Txb5* significa que una torre capture una pieza de *b5* y *dxe6* indica que el peón de la columna *d* capture en *e6*.
- Puede ocurrir que dos piezas iguales se puedan mover a la misma casilla, en caso de que esto suceda se indica la columna de la pieza que se mueve, si ambas piezas están en la misma columna se indica la fila. Por ejemplo, *Ced5* significa que el caballo de la columna *e* es el que se mueve a la casilla *d5*.
- El jaque se anota con un signo + y el jaque mate con # o con ++.
- El enroque corto se indica con 0-0 y el enroque largo como 0-0-0.
- Una captura al paso se indica con a.p..

Algunas abreviaturas empleadas en el análisis de partidas son:

- !: Buena jugada.
- !!: Jugada brillante.
- ?: Mala jugada.
- ???: Muy mala jugada.
- !?: Jugada interesante.
- ?!: Jugada dudosa.

3.7.2. Notación de Forsyth-Edwards

La notación de Forsyth-Edwards más conocida como notación FEN no es más que un sistema de notación estándar utilizado para describir o registrar posiciones dentro de un tablero de ajedrez. Fue propuesto en 1883. Además de situar las piezas sobre el tablero aporta información adicional sobre la posición. Es una notación muy utilizada para anotar las posiciones iniciales de problemas y estudios de ajedrez, ya que la mayoría de los programas de ajedrez la soportan, no ocupa mucho espacio, es escalable para muchos programas y es muy empleada por personas invidentes.

Las normas básicas del sistema FEN son:

- Las piezas se nombran por su inicial que varía dependiendo del idioma. Las piezas blancas se anotan en mayúsculas, K (rey), Q (dama), R (torre), B (alfil), N (caballo), P (peón). Las mismas se utilizan para las piezas negras pero con letra minúscula.
- El tablero se lee de izquierda a derecha y de arriba a abajo, se empieza por la casilla a8.
- En cada fila anotaremos cada pieza por orden comenzando por la izquierda. En caso de encontrarnos casillas vacías pondremos el número de ellas que haya seguidas, una casilla vacía equivale a un 1, dos casillas vacías es un 2, etc.
- Una fila que esté completamente vacía se anota con el número 8.
- Cada fila se separa del resto mediante una barra (/).

Vamos a mostrar a continuación una posición con el correspondiente código FEN el cual vamos a analizar desglosándolo en seis campos diferentes.

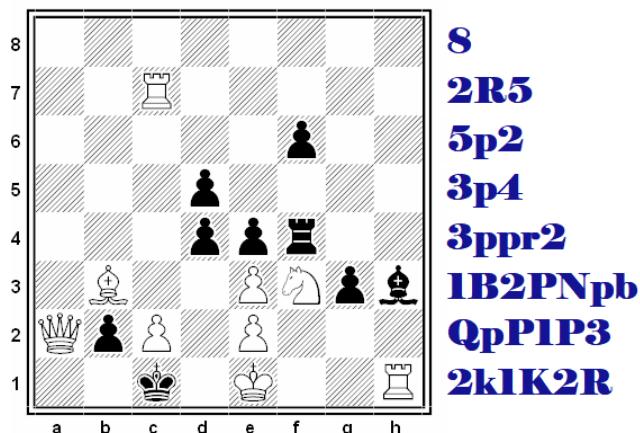


Figura 3.11.: FEN: 8/2R5/5p2/3p4/3ppr2/1B2PNpb/QpP1P3/2k1K2R w K – 0 10. [90]

- 1. Posición de piezas: 8/2R5/5p2/3p4/3ppr2/1B2PNpb/QpP1P3/2k1K2R.
- 2. Bando que tiene el turno: *w*.
- 3. Enroques permitidos: *K*.
- 4. Casilla de captura al paso: *–*.

3. Fundamentos del Ajedrez

- 5. Número de medio jugadas que han sido realizadas desde la última captura o movimiento de peón: 0.
- 6. Número de la siguiente jugada: 1.

El primer campo describe la posición de las piezas en cada fila, en nuestro ejemplo podemos apreciar que la última fila está vacía por eso aparece en primer lugar un 8, la segunda fila tiene dos casillas vacías la torre blanca y cinco casillas vacías y así iríamos analizando las ocho filas.

El segundo campo identifica el color de las piezas a las que les toca mover, w indica que mueven las blancas y b que mueven las negras. En nuestro ejemplo el turno actual lo tienen las piezas blancas.

El tercer campo se emplea para saber si hay posibilidad de realizar algún tipo de enroque. Los códigos que se emplean para ello son:

- K: las piezas blancas pueden enrocarse en el flanco de rey.
- Q: las piezas blancas pueden enrocarse en el flanco de dama.
- k: las piezas negras pueden enrocarse en el flanco de rey.
- q: las piezas negras pueden enrocarse en el flanco de dama.

Por lo que si todos los enroques estuvieran permitidos el tercer campo quedaría de la siguiente manera KQkq. En este caso solo se puede enrocar el rey blanco en el flanco de rey.

El cuarto campo indica sobre la casilla que es posible realizar una captura al paso. En caso de no ser posible realizar una captura al paso se indica como en este caso con un guión.

El quinto campo se emplea para indicar el número de medio jugadas realizadas desde la última captura o movimiento de peón. Así es posible determinar si la partida ha terminado en tablas en base a la regla de los 50 movimientos explicada anteriormente. En este caso no se ha realizado ninguna medio jugada.

El sexto y último campo identifica el número de movimiento de la siguiente jugada que se vaya a realizar. Toma el valor 1 para la posición inicial y se incrementa en uno en cada movimiento de las negras. En este ejemplo la siguiente jugada se corresponde con el movimiento número 10 de las blancas.

4. La Historia del Ajedrez Computacional

En este capítulo se explica la historia del ajedrez computacional y cómo ha evolucionado. En primer lugar, se expone la idea que tenía el ser humano de imitar su propio comportamiento. Posteriormente se describen algunas de las primeras máquinas que se construyeron que supuestamente jugaban al ajedrez. Tras esto, se recorre cada época destacando los avances más significativos hasta llegar a la época actual.

Si observamos toda la historia del ajedrez, descrita en el capítulo anterior, podemos apreciar que el ajedrez ha pasado por etapas evolutivas importantes. Sin embargo, nada ha revolucionado tanto el ajedrez como la llegada de los motores de ajedrez por ordenador. Hoy en día, mucha gente está familiarizada con algunos motores como Stockfish, Leela Chess Zero, Nemorino, etc. Y dado que incluso en un teléfono móvil los programas de ajedrez son más potentes que grandes maestros, ese es el impulso para que los jugadores los utilicen como una herramienta para aprender y desarrollar su juego.

Todos percibimos los motores de ajedrez como algo común y familiar, muchos los utilizan a diario, pero la mayoría desconoce los problemas a los que se enfrentaron las generaciones anteriores al crear estos motores. ¿Quién inventó el primer ordenador de ajedrez? ¿Cuánto tiempo tardó la inteligencia artificial en ser más poderosa que los humanos? Y muchas otras cuestiones son las que vamos a tratar de responder examinando la historia de los motores de ajedrez [45].

4.1. Primeros Pasos

El ser humano desde la más remota antigüedad ha intentado reproducir mediante objetos la apariencia física y los movimientos de animales o personas. Cuando la técnica comenzó a ser lo suficientemente precisa surgieron los autómatas complejos. Además de los artefactos que imitaban estas características también se intentaban copiar características intelectuales de los seres humanos. Así, en pleno siglo XVIII se construyó una máquina que aparentemente jugaba y ganaba al ajedrez.

En 1769 Wolfgang von Kempelen, un consejero de la corte de Austria e inventor, construyó un autómata que era capaz de jugar al ajedrez, El Turco. Estaba conformado por una figura humana, con túnica y turbante, sentada frente a un escritorio sobre el que había un tablero de ajedrez. En 1770 se presentó la máquina en la corte de la emperatriz María Teresa I de Austria. Debido al gran éxito Wolfgang von Kempelen fue solicitado desde numerosos lugares de Europa y el autómata viajó por el mundo e incluso llegó a jugar una partida contra Napoleón Bonaparte. Sin embargo, Wolfgang estaba más ocupado en otros inventos, desmontó el autómata y lo abandonó en un rincón de palacio.

El Turco en realidad no era un autómata, se trataba de un fraude. El mueble que servía

4. La Historia del Ajedrez Computacional

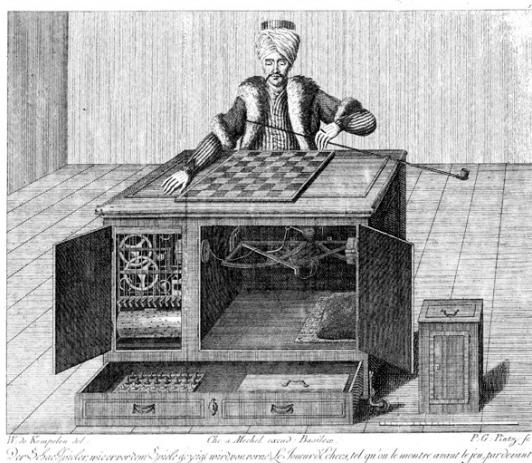


Figura 4.1.: Grabado de El Turco. [87]

de mesa contenía un jugador humano en su interior. Posteriormente aparecieron nuevos engaños como Ajeeb en 1868 o Mephisto en el año 1878 y no fue hasta el año 1912 cuando aparece el primer autómata que realmente podía enfrentarse a un humano, fue obra del ingeniero Leonardo Torres-Quevedo y se llamó El Ajedrecista.



Figura 4.2.: El Ajedrecista, interior del autómata. [87]

Después de estos sucesos el ajedrez mecánico cayó en el olvido y no retomaría el interés hasta la década de los 50 con la aparición de la computadora.

4.2. 1950: El Comienzo de la Era

Durante la Segunda Guerra Mundial, se dedicaron muchos esfuerzos en reclutar a los mejores científicos y a reunirlos en equipos con el objetivo de ganar la guerra. Por ello, las mentes más brillantes del siglo XX dedicaron sus años de guerra a desarrollar nuevas tecnologías y armas que ayudaran a derrotar al enemigo.

A pesar de ello, la guerra dio lugar a un número asombroso de avances. Los científicos sentaron las bases de una serie de nuevas direcciones y en lo que respecta al ajedrez computacional dos nombres Alan Turing y Claude Shannon desempeñaron un papel decisivo en su desarrollo.

Shanon es conocido como el padre de la teoría moderna de la información, y Turing es el padre del ordenador moderno. Sus contribuciones al campo de la informática son más que conocidas, pero también son los padres de los motores de los ordenadores de ajedrez.

El período de tiempo que comprende los años 1949 y 1959 es considerado el nacimiento del ajedrez computacional [24]. Fue en el año 1949 cuando el matemático Claude Shannon escribe el artículo *Programming a Computer for Playing Chess* [76] donde sienta las bases de los principios básicos para hacer un programa de ajedrez, estos principios siguen siendo usados en los programas actuales de ajedrez. También escribió las posibles estrategias de búsqueda, las cuales describimos en el siguiente capítulo.

Un año más tarde, el matemático Alan Turing [85] inventó un algoritmo mediante el cual se podía enseñar a una máquina a jugar al ajedrez. Cabe destacar que Turing programó Turochamp en papel, sin acceso a un ordenador real. Intentó probarlo en el Ferranti Mark I, el primer ordenador comercializado, y fracasó. Finalmente lo probó a mano en una partida amistosa en 1951.

Cincuenta y un años más tarde, el campeón del mundo Garry Kaspárov puso a prueba a Turochamp. El programa no tuvo ninguna posibilidad de vencer, pero todo el mundo reconoció, entre ellos los mejores jugadores de todos los tiempos, que el algoritmo era un logro alcanzado por Turing.

No está claro si Shannon y Turing trabajaron coordinados, de forma independiente o se inspiraron el uno en el otro, se desconoce el contenido de sus conversaciones. Sin embargo, como acabábamos de reflejar, aunque no colaboraran, sus aportaciones fueron enormes. Todas las generaciones posteriores de desarrolladores de ajedrez computacional se apoyaron en estos dos genios.

En el año 1951, Diertrich Prinz, un amigo de Alan Turing, consiguió realizar un algoritmo en el Ferranti Mark I, y crear un programa capaz de resolver el mate en dos movimientos. En 1956, un grupo de científicos dirigido por Stan Ulam creó un programa capaz de jugar al ajedrez, Maniac I. El programa trabajaba en un tablero 6x6 y sin alfíes, y necesitaba unos 12 minutos a una profundidad 4 para realizar movimientos. El programa jugó 3 partidas y ganó la última siendo la primera vez que un humano perdía frente a una computadora en un juego de habilidad intelectual. En 1957, el ingeniero de IBM Alex Bernstein creó el primer programa automatizado capaz de jugar una partida completa de ajedrez.

4.3. Década de los 60 y los 70: Un Gran Cambio

Los primeros motores de ajedrez por ordenador eran bastante débiles y primitivos, pero durante estas dos décadas su poder creció rápidamente.

En las décadas de 1960 y 1970, los algoritmos de los motores de ajedrez mejoraron mucho. La base la puso el genio John von Neumann, que desarrolló el algoritmo minimax 5.3.5.1, que se adaptaba perfectamente al ajedrez, ya que el objetivo principal de este algoritmo era

4. La Historia del Ajedrez Computacional

reducir el árbol de búsqueda de movimientos. En las siguientes décadas, y como veremos en el próximo capítulo, minimax se mejoró con una heurística avanzada y una profundización iterativa que aumentó gradualmente la profundidad de búsqueda de minimax.

La limitación que tenían Turing y otros pioneros se encontraba en la potencia de cálculo. Sin embargo, en estas décadas, la velocidad del hardware creció exponencialmente. Esto hizo posible la aplicación de algoritmos avanzados, los tiempos de búsqueda de minimax se redujeron considerablemente.

En los años sesenta y setenta, el ajedrez por ordenador adquirió notoriedad. El primer campeonato entre ordenadores de ajedrez se celebró en Nueva York, en el año 1970. Cuatro años más tarde se celebró en Estocolmo el primer Campeonato Mundial de Ajedrez por Ordenador. Con la aparición de la escena surgieron empresas especializadas que comenzaron a desarrollar software y hardware de ajedrez a medida. Esto permitió que los motores de ajedrez fueran aún mejores. Estos motores, ya eran lo suficientemente buenos como para vencer a un aficionado, pero todavía no podían competir contra un maestro.

En 1967, MacHack VI fue el primer ordenador que derrotó a un oponente humano, el cual tenía una fuerza de unos 1300 puntos Elo. En 1976 se produjo un salto significativo, el motor de ajedrez Chess 4.5 ganó el Grupo B del torneo Paul Mason en el norte de California. Un año más tarde, también ganó el Abierto de Minnesota con un rendimiento de 2271 puntos Elo y derrotó a un jugador con una calificación de 1969.

A finales de la década de los 70, un ordenador de ajedrez podía competir contra humanos, pero no podía jugar al nivel de los maestros. Esto cambió en la década de 1980.

4.4. Años 80: El Reto

A principios de esta década, la programación de motores de ajedrez se había convertido en un negocio lucrativo. Los ordenadores personales se generalizan en los hogares, aumenta drásticamente el interés por los programas informáticos, incluidos los motores de ajedrez. Se conoce que las ventas de ordenadores a principios de esta década superaron los 100 millones de dólares, luego no es de extrañar que estos siguieran mejorando.

Edward Fredkin, profesor de informática de la Universidad Carneige Mellon, presentó el premio Fredkin. Ofrecía premios en metálico por diversos logros en el mundo de la programación de ajedrez (cinco mil dólares para el primer motor que alcanzara el nivel de maestro, diez mil para el primero que alcanzara el nivel de gran maestro y cien mil dólares para el que consiguiera vencer al campeón del mundo). Esto provocó que comenzara una carrera para desarrollar el mejor motor de ajedrez.

En 1983, Belle [19], un ordenador desarrollado por Joe Condon (hardware) y Ken Thompson (software) que tenía como único objetivo jugar al ajedrez, fue la primera máquina en alcanzar el nivel de maestro.

Al final de 1980 un grupo de estudiantes crearon Deep Thought [48], el primer motor en alcanzar el nivel de gran maestro que lo logró a finales de la década de los ochenta. Este fue un programa de ajedrez que fue el prototipo de Deep Blue, el cual fue construido por un equipo de IBM y estaba compuesto por hardware especializado para jugar al ajedrez y podía calcular unas 200 millones de posiciones por segundo. En 1988 compartió el primer puesto

con el gran maestro Tony Miles en el Abierto de Estados Unidos y un año más tarde derrotó con contundencia al maestro internacional David Levy.

La derrota de la humanidad frente a las máquinas parecía inevitable. Por tanto, se decidió recurrir al campeón del mundo de aquella época y posiblemente el mejor jugador de todos los tiempos Garry Kaspárov. En el año 1985 se celebró en Hamburgo una sesión de partidas donde se enfrentaba contra 32 máquinas de ajedrez, y las derrotó a todas. En 1989, Kaspárov se enfrentó a Deep Thought [67] en un partido a dos partidas y consiguió derrotar a la máquina. El desenlace de este tipo de partidas cambiaría en la década de los 90.

4.5. 1990-1997: El Control de los Motores

En la década de 1990 se produjeron muchos más enfrentamientos entre humanos y ordenadores de ajedrez. Los mejores motores de ajedrez comenzaron a competir en torneos de ajedrez de élite, con un éxito desigual.

En 1989, los investigadores en Deep Thought fueron contratados por IBM. Comenzaron a desarrollar una versión más potente del motor. El nombre del proyecto pasó a llamarse Deep Blue [10]. En el año 1996 se llevó a cabo una segunda ronda del hombre contra la máquina donde se enfrentaron Deep Blue y Kaspárov. Deep Blue sorprendió a Kaspárov en la primera partida y se convirtió en el primer motor de ajedrez en derrotar al campeón del mundo en una partida clásica. Sin embargo, Garry se recompuso y ganó el partido por 4-2, retrasando lo inevitable.

Durante los siguientes años, los programadores de Deep Blue siguieron mejorando la potencia del motor de ajedrez. Incluso, contrataron al gran maestro Joel Benjamin como consultor, para que les ayudara a elaborar un libro de aperturas. Tras una cuidadosa preparación, desafiaron a Kaspárov de nuevo para la revancha y todo el mundo sabe lo que ocurrió.



Figura 4.3.: Kaspárov y Deep Blue durante la revancha en 1997. [8]

En este famoso encuentro, Kaspárov ganó la primera partida, pero en la segunda partida hubo una jugada que avergonzó al campeón del mundo, la jugada 44. Esto hizo tambalear su confianza, y en esta segunda partida perdió una seria oportunidad de hacer tablas. Deep Blue lo derrotó en la sexta partida con un sacrificio de caballo al principio de la apertura. Así, Deep Blue venció a Kaspárov y consiguió el premio Fredkin.

4. La Historia del Ajedrez Computacional

Tras el encuentro hubieron muchas elucubraciones, pero 1997 marcó el fin del dominio humano sobre los ordenadores de ajedrez.

4.6. 1997-2006: La Batalla Final

Tras la victoria de Deep Blue, todo el mundo esperaba que la brecha entre humanos y máquinas aumentara aún más. Sin embargo, durante los seis años siguientes, hasta aproximadamente 2003, la humanidad siguió resistiendo.

Los encuentros entre máquinas y personas se fueron sucediendo. En 2002, el nuevo campeón del mundo, Vladimir Krámnik, hizo tablas con el motor Deep Fritz. En 2003 Kasparov empató dos partidas contra Deep Junior 7 y X3d Fritz. Sin embargo, en 2004 comenzó a producirse lo que se venía avisando, Hydra derrotó a tres grandes maestros de forma aplastante

En los años 2004 y 2005 se celebraron en Bilbao, España, dos torneos entre los principales grandes maestros de ajedrez y computadoras de ajedrez donde las máquinas salieron victoriosas en ambos torneos. En 2006 Deep Fritz derrotó al campeón mundial Vladimir Krámnik por 4-2 lo que supuso la victoria definitiva por parte de las máquinas.

4.7. 2006-2017: La Época Dorada

Todos los días, los motores impregnaban todos los ámbitos del ajedrez, desde la retransmisión y el análisis hasta el juego. Se han convertido en una herramienta indispensable para cualquier ajedrecista. Cuando aparecieron los smartphones, se desarrollaron inmediatamente versiones de motores de ajedrez para Android e iOS.

El nivel de los Campeonatos Mundiales de Ajedrez por Ordenador (WCCC) sigue creciendo. La polémica ha surgido por el prestigio y el dinero, donde el caso más notorio es la descalificación del motor Rybka, cuatro veces campeón del mundo, por plagio del código.

Además del Campeonato Mundial de Ajedrez [15], en 2010 se introdujo una nueva competición, la Top Chess Engines Competition (TCEC). A diferencia del WCCC, el TCEC ofrece partidas más largas, que se juegan con equipos de alta calidad, lo que se traduce en una mayor calidad del ajedrez.

Ahora los humanos ya no son capaces de derrotar a las máquinas en juegos como el ajedrez, ni siquiera jugando la máquina con algún tipo de hándicap. Los enfrentamientos entre humanos y máquinas han demostrado que la brecha se está ampliando.

En 2015 el motor de ajedrez Komodo jugó 6 partidas con hándicap contra Movsesyan, un antiguo jugador top 10, y lo derrotó fácilmente. En 2016, Hikaru Nakamura, uno de los mejores jugadores de ajedrez del mundo, también salió derrotado al enfrentarse a Komodo.

Hasta finales de 2017 no ocurrió nada revolucionario. Su poder cambió de forma constante, pero lentamente. Nadie esperaba grandes cambios, pero apareció AlphaZero.

4.8. 2017: Una Nueva Revolución

El 5 de septiembre de 2017, un grupo de científicos de Google AI Deep Mindshattered consiguió descifrar el mundo del ajedrez. En su artículo *Mastering chess and shogi by self-play with a general reinforcement learning algorithm* [78], describen el desarrollo de un nuevo motor de ajedrez, AlphaZero, basado en un enfoque totalmente nuevo. En lugar de la búsqueda alfa-beta 5.3.5.3 y la función de aproximación lineal para la estimación de la posición que utilizan los motores tradicionales, AlphaZero utiliza una función de aproximación no lineal basada en una red neuronal profunda y el método de Monte-Carlo. Como consecuencia, AlphaZero podría aprender por sí mismo.

El equipo probó la fuerza del nuevo motor en un encuentro de 100 partidas contra el motor de ajedrez clásico más potente disponible en ese momento, Stockfish 8. AlphaZero salió victorioso de manera airosa pero lo más impresionante fue el estilo de juego, más intuitivo y humano que el de los motores tradicionales.

La reacción ante este acontecimiento fue variada. Muchos ajedrecistas se quedaron atónitos ante una derrota tan abultada de Stockfish, pero por otro lado, había mucho escepticismo.

El 7 de diciembre de 2018, el equipo de Deep Mind publicó otro artículo [79]. En él presentaron los resultados de otro encuentro entre AlphaZero y Stockfish. De mil partidas, AlphaZero ganó 155 y perdió solo 6. Por tanto, AlphaZero había tenido un impacto significativo en el mundo del ajedrez y provocó una nueva revolución.

4.9. Desde 2019: Inicio de la Era de la Inteligencia Artificial

Tras la aparición de AlphaZero, un motor basado en redes neuronales, el ajedrez cambió para siempre. Por desgracia, nadie podía comprar AlphaZero o concederle una licencia, por lo que surgió Leela Chess Zero, un motor de red neuronal de código abierto.

En 2019, Leela es capaz de vencer la final frente al motor de ajedrez tradicional más potente, Stockfish, dando inicio a una nueva era, la era de la Inteligencia Artificial.

Aunque Stockfish conservó el primer puesto en la lista de motores de ajedrez en términos de clasificación, los motores de redes neuronales comenzaron a acercarse cada vez más. Así, en septiembre de 2020, se lanzó Stockfish 12, y se anunció que Stockfish se había hecho cargo del proyecto Stockfish + NNUE ¹. Así, la fuerza bruta de Stockfish se ha visto reforzada por la capacidad de estimación de la red neuronal. Actualmente, Stockfish 15 es el motor mejor clasificado según la Computer Chess Rating List (CCRL).

¹NNUE significa red neuronal eficientemente actualizable.

5. Elementos de un Motor de Ajedrez

Los programas informáticos de ajedrez ya sean implementados en hardware o software emplean un paradigma diferente al de los seres humanos para elegir sus movimientos, usan métodos heurísticos para construir, buscar y evaluar árboles que representan secuencias de movimientos desde la posición actual e intentan ejecutar la mejor secuencia para salir victoriosos. Estos árboles suelen ser muy grandes, teniendo miles o millones de nodos. En este capítulo se presentan y se analizan los tres componentes esenciales de un motor de ajedrez [56]. El primero es el generador de movimientos que se encarga de generar estados sucesores y para ello requiere de una representación del tablero real de manera que la computadora sea capaz de entenderlo (codificación del tablero). El segundo es el módulo de búsqueda que es el que explora los estados sucesores a través del árbol de juego, y el tercero es el módulo de la función de evaluación que asigna un valor numérico a una posición para medir cómo de buena es.

5.1. Generador de movimientos

La generación de jugadas es una parte básica del motor de ajedrez. Si queremos que el programa juegue, es necesario que el motor sea capaz de generar una lista de jugadas con todos los posibles movimientos candidatos a ser jugados. La implementación depende en gran medida de la representación del tablero de ajedrez, y puede generalizarse en dos tipos, la generación de jugadas pseudolegales y legales.

- *Generación de movimientos pseudolegales:* las piezas obedecen a sus reglas normales de movimiento pero no se comprueba de antemano si dejarán al rey en jaque.
- *Generación de movimientos legales:* solo se generan jugadas legales, aquellas jugadas pseudolegales que no dejan a su propio rey en jaque, lo que significa que hay que dedicar un tiempo extra para asegurarse de que el rey no va a quedar en jaque después de cada jugada. La principal dificultad son *las clavadas*¹.

Pasamos a ver ahora algunas de las diferentes representaciones del tablero que existen.

5.2. Codificaciones del tablero de ajedrez

La representación del tablero en el ajedrez por computadora es fundamental para todos los aspectos de un programa de ajedrez, incluida la representación de movimientos, la función de evaluación y la realización y eliminación de movimientos, es decir, la búsqueda de movimientos, así como el mantenimiento del estado del juego durante la partida. En el ajedrez

¹Una clavada en ajedrez es una situación en la cual una pieza no puede moverse sin exponer a otra pieza de su color y de mayor valor a ser capturada.

5. Elementos de un Motor de Ajedrez

por computadora, la representación del tablero de ajedrez es una estructura de datos que representa la posición en el tablero y el estado del juego asociado. Existen muchas estructuras de datos, llamadas colectivamente como representación del tablero. Los programas de ajedrez emplean a menudo más de una representación del tablero en diferentes momentos para obtener una mayor eficiencia. Los factores principales a la hora de elegir una representación de la placa son la eficiencia de ejecución y la huella de memoria. Otras consideraciones son el esfuerzo necesario para codificar, probar y depurar la aplicación.

Los primeros programas usaban listas de piezas y listas cuadradas, ambas están basadas en matrices. La mayoría de las implementaciones modernas emplean un enfoque de matriz de bits más elaborado pero más eficiente llamados bitboards o tableros de bits que mapean bits de una palabra de 64 bits o palabra doble a cuadrados de la tarjeta.

5.2.1. Requisitos

Una descripción completa de una posición de ajedrez debe contener los siguientes elementos:

- Ubicación de cada pieza en el tablero.
- Bando al que le toca mover.
- Estado de la regla de los 50 movimientos.
- Estado del enroque de ambos jugadores.
- Si es posible realizar alguna captura al paso y de qué pieza se trata.

Normalmente, la representación del tablero no incluye la regla de empate por triple repetición de posiciones. Para determinar esta regla, es necesario mantener un historial completo del juego desde la última acción irreversible (captura, movimiento de peón o enroque) y por lo general, se realiza un seguimiento en estructuras de datos separadas.

El estado del tablero también puede contener información secundaria como para los cuadrados que contienen piezas, las casillas que son atacadas o protegidas por cada pieza, las piezas clavadas, etc.

El estado del tablero se encuentra asociado a cada nodo del árbol de juego, lo que representa una posición a la que se llega mediante un movimiento; este movimiento puede que se haya jugado sobre el tablero o puede haber sido generado como parte de la búsqueda del programa.

5.2.2. Tipos basados en arrays

5.2.2.1. Listas de piezas

Algunos de los primeros programas de ajedrez de la historia trabajaban con cantidades extremadamente limitadas de memoria, mantenían listas de serie (matrices) de las piezas en un orden de búsqueda conveniente, ya que alojar las 64 casillas del tablero en memoria suponía mucho. Asociada con cada pieza estaba su ubicación en el tablero, además de otra información como casillas que representan sus movimientos legales. Había varias listas, una para las piezas blancas y otra para las negras y estas se dividían en su mayoría en piezas y

peones. Esta fue una representación compacta, ya que la mayoría de las casillas del tablero están desocupadas, pero son ineficaces puesto que adquirir información sobre la relación de las piezas con el tablero o entre sí era demasiado tedioso. Actualmente, en muchos de los programas todavía se utilizan las listas de piezas junto con una estructura de representación de tablero separada, para dar acceso en serie a las piezas sin tener que buscar en el tablero.

5.2.2.2. Lista cuadrada

Para representar un tablero una de las formas más sencillas que hay es crear una matriz bidimensional 8×8 o lo que es lo mismo, una matriz unidimensional de 64 elementos. Cada elemento de la matriz identifica qué pieza ocupaba la casilla dada o indica si la casilla está vacía. Una posible codificación es considerar el número 0 como vacío, los números positivos como piezas blancas y los negativos como negras. Por ejemplo:

- 0: Vacío.
- 1: Peón blanco.
- 2: Caballo blanco.
- 3: Alfil blanco.
- 4: Torre blanca.
- 5: Dama blanca.
- 6: Rey blanco
- -1: Peón negro.
- -2: Caballo negro.
- -3: Alfil negro.
- -4: Torre negra.
- -5: Dama negra.
- -6: Rey negro

Un ejemplo de esta representación se puede encontrar en el capítulo 14, concretamente en la figura 14.2. Este tipo tiene un problema en la generación de movimientos. Cada movimiento debe verificarse para comprobar si está en el tablero y esto ralentiza el proceso significativamente. Una solución es usar matrices 12×12 con los bordes exteriores llenos con el valor 99. Durante la generación de movimientos la operación que verifica si hay una pieza en la casilla de destino también comprobará si esa casilla de destino está fuera del tablero. Podemos lograr un mejor uso de la memoria con una matriz 10×12 que proporciona las mismas funcionalidades que la matriz 12×12 . Algunos motores de ajedrez emplean matrices de 16×16 para mejorar la velocidad de conversión de las filas y las columnas y también permiten trucos especiales de codificación.

5.2.2.3. Método 0x88

Este método aprovecha que las dimensiones del tablero de ajedrez son una potencia par de dos, en concreto, 8 al cuadrado. El método 0x88 usa una matriz unidimensional de tamaño

5. Elementos de un Motor de Ajedrez

128, numerada de 0 a 127 en vez de una matriz de tamaño 64. Se trata de dos tableros uno al lado del otro donde el tablero de la izquierda es el tablero real y el tablero de la derecha representa los movimientos ilegales. Cuando se generan movimientos en el tablero real se puede comprobar que la casilla se encuentra en el tablero usando el operador AND del número de la casilla con formato hexadecimal 0x88, en binario 10001000. Si el resultado es distinto de cero entonces la casilla está fuera del tablero principal por lo que la casilla es ilegal. Esta operación bit a bit requiere menos recursos informáticos que las comparaciones de enteros por lo que los cálculos de movimientos ilegales son más rápidos. Además, la diferencia entre las coordenadas de dos casillas determina de forma única si se encuentran en la misma fila, columna o diagonal.

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 5.1.: Representación del método 0x88. [23]

5.2.3. Bitboards

Un tablero de bits llamado comúnmente bitboard es una estructura de datos de matriz de bits especializada que se utiliza en sistemas informáticos cuando juegan a juegos de mesa como es el caso del ajedrez. Cada bit corresponde a un espacio o pieza del tablero lo que permite operaciones paralelas bit a bit para establecer o consultar el estado del juego o determinar movimientos o jugadas en el transcurso de la partida. Los bits en el propio tablero de bits se relacionan entre sí según las reglas del juego y normalmente forman una posición de juego cuando se agrupan. Los bitboards son sobre todo efectivos cuando los bits asociados de varios estados relacionados en el tablero encajan en una sola palabra o palabra doble de la arquitectura de la CPU, de manera que los operadores bit a bit como AND y OR pueden usarse para construir o consultar estados del juego.

Centrándonos en el ajedrez, esta representación es más eficiente y más elaborada que las estructuras basadas en arreglos. La representación más común y más simple de la configuración de piezas en un tablero de ajedrez es como una lista o matriz de piezas en un orden de búsqueda conveniente que asigna cada pieza a su ubicación del tablero. La recopilación de las casillas atacadas por cada pieza necesita una enumeración en serie de dichas casillas para una pieza. Los mapas se actualizan por cada movimiento lo que requiere una búsqueda lineal o dos si se capturó alguna pieza a través de la lista de piezas. Se mantienen listas separadas para piezas blancas y negras y, en general, para peones blancos y negros.

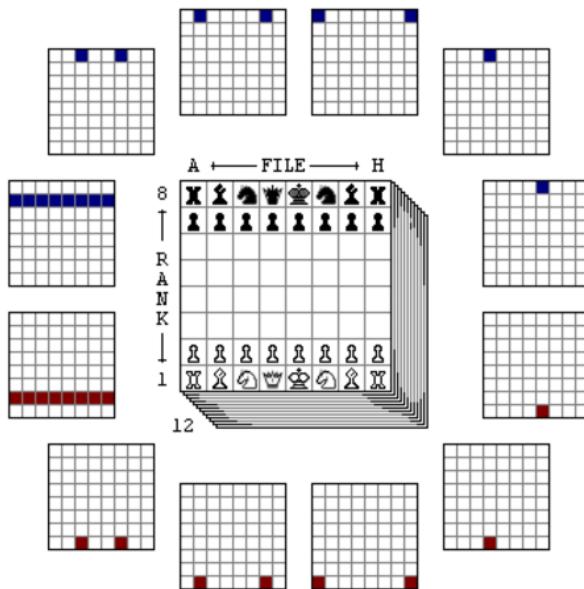


Figura 5.2.: Representación de una posición mediante bitboards. [24]

5.2.3.1. Estándar

En las representaciones de tableros de bits, cada bit de una palabra de 64 bits o palabra doble en arquitecturas de 32 bits se asocia con una casilla del tablero de ajedrez. El mapeo de bits que se puede utilizar puede ser cualquiera, pero por convención los bits se asocian con cuadrados de izquierda a derecha y de abajo hacia arriba. El bit 0 se corresponde con la casilla que en notación algebraica es *a1*, el bit número 7 es la casilla *h1*, el bit 56 es *a8* y el último, el bit 63, se corresponde con la casilla *h8*.

Muchas configuraciones diferentes del tablero están representadas por sus propios tableros de bits, incluidas las ubicaciones de los reyes, de los peones de ambos bandos, tableros de bits para cada una de las otras piezas o combinaciones de piezas como todas las piezas blancas. Hay dos tableros de bits que son universales, uno por casilla para todas las piezas que atacan esa casilla y otro inverso para todas las casillas atacadas por una pieza por cada casilla que contiene una pieza. Los tableros de bits también pueden ser constantes como uno que represente la primera fila y está formado por un bit en las posiciones 0-7. Existen otros bitboards locales o de transición como el que controla todos los espacios adyacentes al rey atacados por piezas opuestas.

Los tableros de bits estándar tienen algunos inconvenientes siendo uno de los más importantes el agrupar los vectores de ataque de las piezas deslizantes (alfil, torre y dama), ya que tienen un número indefinido de espacios de ataque en función de otros espacios ocupados. Esto requiere varias secuencias largas de máscaras, turnos y complementos por pieza.

5. Elementos de un Motor de Ajedrez

5.2.3.2. Tableros de bits rotados

Debido al inconveniente mencionado anteriormente surgen los tableros de bits rotados. Estos son estructuras de datos de bitboards complementarias que permiten la tabularización de los vectores de ataque de piezas deslizantes, utilizan copias rotadas de un tablero de bits para colocar espacios (bits) en una fila o diagonal en bits adyacentes análogos a los bits que representan una fila. Se trata de dos tableros de bits, uno para los vectores de ataque de archivos de las torres y otro para los vectores de ataque diagonales y antidiagonal de los alfiles. Así, tan solo necesitamos realizar una búsqueda en una tabla en vez de necesitar largas secuencias de operaciones bit a bit.

Los tableros de bits rotan la configuración del tablero en 45 grados, 90 grados y/o 315 grados. Con este tablero de bits resulta sencillo determinar los ataques de torre en una fila, tan solo usando una tabla indexada por el cuadrado ocupado y las posiciones ocupadas en esa fila. Para los ataques de torre en la columna tan solo necesitamos girar el tablero de bits 90 grados y así se pueden examinar de la misma manera. Ahora es fácil comprobar las diagonales al añadir tableros de bits rotados 45 y 315 grados. La dama se puede ver como una combinación de alfil y torre. Ahora bien, rotar un tablero de bits no es una transformación elegante y puede necesitar decenas de instrucciones.

5.2.3.3. Hash directo

Los vectores de ataque de las filas y columnas de las torres y los vectores de ataque diagonales y antidiagonal de los alfiles se pueden enmascarar por separado y usar para ello índices en una tabla hash de vectores de ataque precalculados basada en los bits de ocupación de la parte enmascarada. El vector de ataque completo de una pieza se obtiene como la unión de cada uno de los vectores unidireccionales indexados de la tabla hash. El número de entradas de la tabla hash no es muy alto pero se requieren dos cálculos de función hash y dos búsquedas por pieza.

5.2.3.4. Tableros de bits mágicos

Los tableros de bits rotados son una buena idea pero hemos visto que no son muy eficientes ni muy prácticos. Podríamos tener para cada casilla una tabla que tomara todo el tablero de bits de ocupación y devolviera las posibles jugadas. Por desgracia, esto es inviable, aunque se puede reducir a algo manejable.

La primera observación clave es la misma que la que tuvimos en cuenta para los tableros de bits rotados, si tenemos una torre en $d4$, solo importan las piezas que se encuentran en la cuarta fila y las que estén en la columna d . El resto de bits los podemos enmascarar a cero, disminuyendo de manera drástica el número de entradas posibles con las que tratar. La segunda observación clave es que, para muchas entradas, la salida será idéntica. Por último, para el caso de la torre en $d4$, si hay piezas en $f4$ y $d6$, las piezas que estén por ejemplo en $h4$ y $d7$ no importan, porque nuestra torre está bloqueada por las piezas de $f4$ y $d6$. El resultado es el mismo si están ahí como si no. Estas ideas se combinan en la idea que se conoce actualmente como tableros de bits mágicos.

Los tableros de bits mágicos conocidos como magic bitboards son una extrapolación de la

compensación espacio-temporal de la búsqueda directa en una tabla hash de los vectores de ataque. Estos utilizan una transmutación del vector de ataque completo como índice en la tabla hash. El nombre de magia o mágicos se refiere simplemente a la generación y uso de una función *hash perfecta*² junto con diversos trucos para reducir el tamaño potencial de la tabla hash que tendría que almacenarse en memoria. Gracias a la rápida multiplicación de 64 bits y a las rápidas y enormes memorias caché de los procesadores recientes se pueden llevar a cabo los dos ataques de línea de alfil o torre en una sola ejecución. Estos tableros se han convertido en un estándar de facto de los motores de tablero de bits modernos.

Una observación a tener en cuenta es que los cuadrados exteriores, es decir, la primera fila y la última así como la primera columna y la última son irrelevantes para la ocupación del vector de ataque, la pieza ataca esos cuadrados o no dependiendo de otros bloqueos de piezas independientemente de la ocupación, por lo que pueden eliminarse de la consideración quedando así 36 casillas, es decir, un tablero de 6x6.

La técnica de generación de la tabla de bits de movimiento mágico consta de cuatro pasos clave:

- Enmascarar los bits de ocupación relevantes para formar una clave. Por ejemplo, si hay una torre en a1 los bits de ocupación relevantes son de a2 a a7 y de b1 a g1.
- Multiplicar la clave por un *número mágico*³ para obtener un mapeo de índices, así podemos encontrar diferentes combinaciones de colisiones hash. Este número mágico se puede generar por fuerza bruta de ensayo y error de manera bastante fácil, aunque no es del todo seguro que el número mágico sea el mejor posible.
- Desplazar a la derecha el mapeo del índice en 64-n bits para crear un índice siendo n el número de bits en el índice. Cuanto mejor sea el número mágico menos bits serán necesarios en el índice.
- Utilizar el índice para referenciar una base de datos de movimientos preinicializada.

En general, un mapeo uno a uno de N bits ocupados dispersos a N bits consecutivos no siempre es posible debido a los desbordamientos. Un mapeo perfecto de N bits dispersos a N bits consecutivos probablemente no sea mínimo para la mayoría de los cuadrados. Requiere, uno o dos huecos dentro de los N bits consecutivos para evitar colisiones, disparando el tamaño de la tabla.

Pero el propósito es hacer un hash perfecto de los conjuntos de ataque en lugar de los bits ocupados consecutivos. El número de conjuntos de ataque distintos es mucho menor que las ocupaciones relevantes. Por tanto, si tenemos la ayuda de colisiones constructivas, alguna conjectura inicial sobre cómo mapear los bits, y/o prueba y error, siempre es posible usar exactamente N bits. Es posible reducir el número de bits si uno se esfuerza en maximizar las colisiones constructivas. Hay diferentes tipos de implementaciones como fancy magic bitboards, plain magic bitboards o black magic bitboards entre otras.

²Una función hash perfecta en términos informáticos es una función hash que para un conjunto S asigna distintos elementos en S a un conjunto de números enteros sin colisiones. En términos matemáticos, es lo que se conoce como función inyectiva.

³Un número mágico en informática se refiere a unos caracteres alfanuméricos que identifican de manera codificada un archivo, generalmente se ubican al comenzar dicho archivo.

5. Elementos de un Motor de Ajedrez

5.2.4. Otros métodos

5.2.4.1. Codificación Huffman

Como hemos visto, las posiciones del tablero de ajedrez pueden ser representadas como patrones de bits. La codificación Huffman almacena los elementos más comunes del tablero, como son las casillas vacías y los peones, con menos bits que un elemento más común. La codificación viene dada por:

- Casilla vacía: 0
- Peón: $10c$
- Alfil: $1100c$
- Caballo: $1101c$
- Torre: $1110c$
- Dama: $11110c$
- Rey: $11111c$

La letra c es el bit que representa el color de la pieza, si la pieza es blanca se sustituye la c por un 1 y si la pieza es negra por un 0. Además, se necesitan otros bits adicionales. Se requieren 6 bits para la regla de los 50 movimientos, para marcar la columna de posible captura al paso se emplean 4 bits. Para saber a qué jugador le toca mover se utiliza otro bit más y para conocer los derechos de enroque 4 bits adicionales.

Esta codificación requiere más uso de la CPU que otras representaciones que minimizan el trabajo del procesador y de los ciclos de memoria. El tamaño final de la representación hace que esta sea ideal para almacenar posiciones a largo plazo.

5.2.4.2. Representación Compacta de Tablero de Ajedrez (CCR)

Este método usa como concepto clave un array de 8 elementos para representar todo el tablero. Cada entero de 32 bits sin signo en la matriz representa una sola fila. Las piezas se codifican en 4 bits, por lo que cada entero de rango contiene 8 casillas, en total una fila. El código de ocupación de una casilla se puede marcar fuera de un registro y añadir al contador del programa para indexar una tabla de salto, ramificándose directamente al código y así poder generar movimientos para el tipo de pieza de esa casilla. No se requieren controles del borde del tablero y no es posible realizar movimientos fuera del tablero, lo que aumenta la velocidad de generación de movimientos.

5.3. Técnicas de búsqueda

Debido a que encontrar o adivinar una buena jugada en una posición de ajedrez es difícil de lograr estáticamente, los programas de ajedrez se basan en algún tipo de búsqueda [60] para poder jugar razonablemente, consideran los movimientos como un *árbol de juego*⁴. La

⁴En teoría de juegos, un árbol de juego no es más que un grafo dirigido de tipo árbol cuyos nodos representan posiciones del juego y cuyas aristas representan movimientos de los jugadores.

búsqueda implica mirar hacia delante en diferentes secuencias de movimientos y evaluar las posiciones después de hacer los movimientos. Cada movimiento individual se llama hebra o ply y la evaluación termina cuando se llega a una hoja final que es evaluada. Formalmente, la búsqueda de un juego de tablero de suma cero para dos jugadores con información perfecta implica recorrer y minimizar una estructura de datos en forma de árbol mediante varios algoritmos de búsqueda. La mejor forma de discriminar entre estos es mirar las consecuencias de cada movimiento, es decir, la serie de movimientos futuros. Se asumirá que nuestro rival siempre realiza el mejor movimiento posible para asegurar el realizar la menor cantidad de errores posibles.

Aunque difieren en su implementación, casi todos los motores de ajedrez existentes hoy en día implementan en gran medida los mismos algoritmos. Se basan en la idea del algoritmo minimax de profundidad fija adaptado al problema del ajedrez por Shannon en 1950, quien clasificó las búsquedas en dos tipos:

- *Tipo A*: una búsqueda de fuerza bruta que examina todas las variaciones hasta una profundidad determinada.
- *Tipo B*: una búsqueda selectiva que se centra en las ramas importantes con la esperanza de reducir las ramificaciones del árbol de búsqueda.

El Tipo B fue el más popular hasta los años 70, a partir de entonces, los programas de Tipo A tuvieron suficiente poder de procesamiento y algoritmos de fuerza bruta más eficientes y se hicieron más fuertes. Actualmente, la mayoría de los programas se acercan más al Tipo A pero tienen algunas características del Tipo B mencionadas en la subsección de búsqueda selectiva [5.3.4](#).

En esta sección repasaremos lo que son los árboles junto con el concepto de árbol de búsqueda, nos centraremos en el que nos interesa que es el del ajedrez y presentaremos el efecto horizonte. Posteriormente analizaremos la búsqueda por fuerza bruta y la búsqueda selectiva. Por último, nos adentraremos en las técnicas de búsqueda sobre árboles, así como en las optimizaciones que emplean los motores de ajedrez.

5.3.1. Árboles

Antes de entrar en los árboles de búsqueda vamos a ver un breve resumen sobre las estructuras de datos de tipo árbol. Los árboles son las estructuras de datos más utilizadas y una de las más complejas, son un tipo abstracto de datos que imitan la estructura jerárquica de un árbol.

Un árbol puede definirse de forma recursiva como una colección de nodos donde cada nodo es una estructura de datos con un valor, junto con una lista de referencias a los nodos hijos, con la condición de que no haya referencias duplicadas ni que ningún nodo apunte a la raíz. También se puede definir como un árbol ordenado donde cada nodo tiene un valor asignado.

Veamos alguna de la terminología más empleada en árboles:

- *Nodos*: cada elemento que contiene un árbol.
- *Nodo Raíz*: nodo superior del árbol.
- *Nodo Padre*: nodo que tiene un hijo.

5. Elementos de un Motor de Ajedrez

- *Nodo Hijo*: nodo conectado directamente con otro cuando se aleja de la raíz.
- *Nodos Hermanos*: conjunto de nodos que comparten a un mismo padre dentro de la estructura.
- *Nodo Hoja*: nodo sin hijos, se encuentran en los extremos de la estructura.
- *Nodo Interno*: nodo que no es la raíz y que además tiene algún hijo.
- *Nodo Ancestro*: nodo accesible por ascenso repetido de hijo a padre.
- *Nodo Descendiente*: nodo accesible por descenso repetido de padre a hijo.
- *Rama*: una ruta del nodo raíz a cualquier otro nodo.
- *Orden*: número máximo de hijos que puede tener un nodo.
- *Grado*: número de subárboles de un nodo, es decir, número mayor de hijos que tiene alguno de los nodos del árbol, limitado por el orden.
- *Brazo*: conexión entre dos nodos.
- *Camino*: secuencia de nodos y brazos conectados con un nodo descendiente.
- *Nivel*: cada generación dentro del árbol. El nivel de un nodo se define por $1 +$ el número de brazos entre el nodo y el nodo raíz, el nivel de la raíz es 1.
- *Altura de un nodo*: número de brazos en el camino más largo entre ese nodo y una hoja.
- *Altura de un árbol*: altura de su nodo raíz, es decir, número máximo de niveles del árbol.
- *Profundidad*: número de brazos desde la raíz del árbol hasta un nodo.

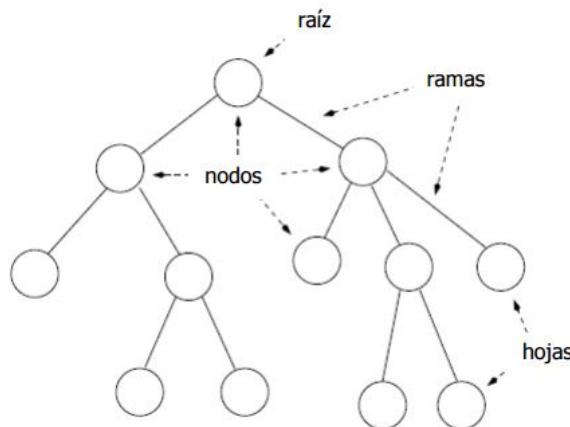


Figura 5.3.: Representación gráfica de un árbol. [35]

Los recorridos o búsquedas sobre árboles son algoritmos que nos permiten recorrer un árbol en un orden específico, nos permiten encontrar un nodo en el árbol o buscar una posición determinada.

Ahora ya podemos introducir el concepto de árbol de búsqueda. El árbol de búsqueda, como parte del árbol de juego, es una estructura de datos dinámica y jerárquica en forma de árbol, un grafo dirigido de nodos también llamados vértices o estados que están conectados

mediante aristas (brazos) o transiciones de estado y se utiliza para localizar claves específicas dentro de un conjunto. Para que un árbol funcione como un árbol de búsqueda, la clave de cada nodo debe ser mayor que cualquier clave de los subárboles de la izquierda y menor que cualquier clave de los subárboles de la derecha. La ventaja de los árboles de búsqueda es que el tiempo de búsqueda es eficiente pero para ello el árbol debe estar bien equilibrado, es decir, las hojas del árbol deben tener profundidades comparables. Existen varias estructuras de árboles de búsqueda como los árboles de búsqueda binaria, B-tree, (a,b)-tree, etc.

Ahora pasamos a describir el árbol de búsqueda para el juego del ajedrez. Los nodos representan posiciones de ajedrez y las aristas dirigidas, las ramas, representan movimientos posibles en cada posición. La raíz del árbol de búsqueda es la posición que queremos evaluar para encontrar la mejor jugada. Las hojas son nodos terminales o nodos en los que se les ha asignado un valor heurístico, donde se logra la profundidad deseada desde la raíz. El objetivo de la búsqueda es encontrar un camino desde la raíz del árbol hasta un nodo terminal que tenga la mayor valoración posible.

Al comienzo del juego las piezas blancas disponen de 20 movimientos posibles. Una vez han movido las blancas, las negras tienen 20 opciones distintas para responder la elección de las blancas. Una vez efectuados estos movimientos se abre un abanico de unas 400 posiciones distintas que pueden surgir. Tras 2 jugadas, el número de posiciones posibles crece de manera exponencial y si seguimos tendríamos un árbol de ajedrez con una cifra astronómica pero finita. En resumen, el número de partidas diferentes que pueden desarrollarse es una cifra superior a todos los átomos del universo.

Si fuera posible examinar el árbol por completo, buscando todas las líneas de juego y el resultado que se obtiene sería posible determinar cuál es el mejor movimiento inicial. Sin embargo, en la práctica el árbol es demasiado extenso y no es posible realizar este cometido. También resulta imposible determinar el mejor movimiento buscando en el árbol de variantes a partir de una avanzada posición de medio juego. La mejor estrategia a seguir es buscar en un sector limitado del árbol de variantes, esperando obtener suficiente información como para elegir correctamente el mejor movimiento a realizar.

Los programas de ajedrez en las líneas de juego cruciales como jaques y capturas tienen mayor profundidad de búsqueda que en otro tipo de jugadas. Los diferentes algoritmos se encargan de decidir dado un árbol de variantes qué movimiento realizar frente a una posición dada.

5.3.2. Efecto Horizonte

El efecto horizonte también llamado problema del horizonte es un problema que se genera y se aplica habitualmente a la inteligencia artificial y a la computación cuando se requieren tomar decisiones en ciertos juegos. En muchos juegos como en el ajedrez el árbol de búsqueda es inmenso y es imposible buscar en todo el árbol. Este problema se produce por el estudio a profundidad fija. Por tanto, para una computadora que solo busca en x capas, existe la posibilidad de que seleccione una jugada perjudicial pero el efecto que produzca no sea visible, ya que no busca en la profundidad del error, es decir, no busca más allá de su horizonte.

Cuando evaluamos un árbol de juego grande empleando ciertas técnicas que veremos a continuación la profundidad de búsqueda es limitada por razones de viabilidad. Sin em-

5. Elementos de un Motor de Ajedrez

bargo, la evaluación de un nodo intermedio del árbol de búsqueda puede dar un resultado engañoso. Cuando existe un camino significativo justo sobre el horizonte de la profundidad de búsqueda, el programa ha caído en el efecto horizonte.

En 1973 Hans Berliner definió este fenómeno como: *La creación de desvíos que demoran sin resolver una consecuencia inevitable; o que hacen parecer alcanzable una meta que no se puede lograr.*

El efecto horizonte se puede mitigar ampliando el algoritmo de búsqueda con una búsqueda en reposo o quiescence como veremos más adelante. Esto permite que el algoritmo pueda ver más allá de su horizonte en busca de una serie de movimientos de gran importancia para el estado del juego como son las capturas en ajedrez.

Como ejemplo de efecto horizonte supongamos que el programa busca en el árbol de juego hasta la capa 3 y desde la posición actual determina que la mejor opción es capturar una torre del oponente. Sin embargo, esto produce que en una serie de combinaciones un peón del oponente corone y se convierta en dama quedando en desventaja. Se ha producido el temido efecto horizonte, el programa evaluó una posición intermedia en una secuencia táctica y obtuvo una valoración errónea.

5.3.3. Fuerza Bruta

La búsqueda por fuerza bruta es una técnica que consiste en enumerar sistemáticamente todas las posibles soluciones a un problema con el objetivo de comprobar si alguna de ellas es la solución. Los algoritmos de fuerza bruta son conceptualmente sencillos pero suelen sufrir un crecimiento exponencial del espacio de búsqueda.

5.3.4. Búsqueda Selectiva

El objetivo de este método consiste en buscar mediante un criterio fijado ramas interesantes, reduciendo así el factor de anchura presente en cada posición del árbol de búsqueda. Basados en consideraciones de tiempo, los programas buscan encontrar movimientos relevantes de una posición con el fin de filtrar las alternativas consideradas inferiores. Normalmente se emplea en ramas forzadas cuya anchura del árbol sea limitada.

Hay varios tipos:

- *Extensiones*: se estudian nodos a más profundidad que a la profundidad máxima y representan casos forzados. Se busca encontrar mejores jugadas más rápidamente o resolver el efecto horizonte. Para el caso del ajedrez se usan extensiones que se producen por jaque, por amenazas, por peones, por la seguridad del rey, etc
- *Podas*: nombre que recibe cualquier heurística que elimina completamente ciertas ramas del árbol de búsqueda, asumiendo que no tienen relación con el árbol de búsqueda. Algunas técnicas de poda se basan en una búsqueda especializada, pero reducida, otras se basan solo en las propiedades estáticas de los movimientos. Existe la poda hacia atrás como la poda alfa-beta que ahora veremos y la poda hacia adelante que aunque puede reducir de manera considerable el árbol de búsqueda también puede introducir errores de búsqueda. En el caso del ajedrez se emplean varios tipos de podas como la poda de movimiento nulo, la poda Delta, la poda de inutilidad, etc.

- *Reducciones*: es una clase de heurística de búsqueda que disminuye la profundidad a la que se busca en una determinada rama del árbol, también conocida como extensión negativa. Para el caso del ajedrez hay varias como las reducciones de movimiento nulo, RankCut, reducciones por fallo (FHR), etc.

5.3.5. Búsqueda en Profundidad

La búsqueda en profundidad es un algoritmo de *búsqueda no informada*⁵ que se emplea para recorrer nodos de las estructuras de datos tipo árbol, como los árboles de búsqueda. Comienza en la raíz y explora todo lo posible a lo largo de cada rama antes de retroceder. Su funcionamiento consiste en expandir cada uno de los nodos que va localizando de manera recurrente, desde el nodo inicial hacia el nodo hijo. Cuando ya ha visitado todos los nodos en ese camino regresa al nodo predecesor, de manera que repite el mismo procedimiento con cada uno de los vecinos del nodo, es decir, examina los nodos hijo antes que los hermanos. Puede implementarse con recursividad utilizando una pila de nodos.

La búsqueda en profundidad se emplea cuando queremos comprobar si una solución entre varias posibles cumple ciertos requisitos. Los requisitos de memoria son moderados, ya que solo se mantiene en memoria un camino desde la raíz hasta una hoja, que crece proporcionalmente con la profundidad de la búsqueda.

Para verlo más claro vamos a ilustrar con un ejemplo el procedimiento donde el recorrido se realiza en orden numérico de forma consecutiva.

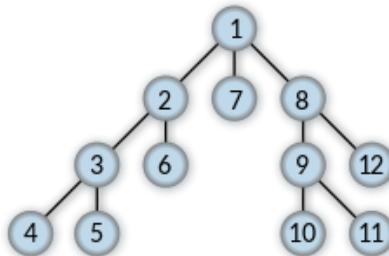


Figura 5.4.: Búsqueda en profundidad. [87]

Hay varios tipos de algoritmos que emplean la búsqueda en profundidad pero en este caso nos centraremos en aquellos que emplean los programas de ajedrez en sus diferentes técnicas de búsqueda, los más importantes son el algoritmo minimax y la poda alfa-beta con sus respectivas optimizaciones, también realizaremos un breve apunte sobre negamax.

5.3.5.1. Minimax

El algoritmo minimax es un algoritmo que se utiliza para determinar la puntuación en un juego de suma cero después de un cierto número de movimientos. Se encarga de decidir dado

⁵Los métodos de búsqueda no informados o ciegos son estrategias de búsqueda donde se evalúa el siguiente estado sin conocer a priori si este es mejor o peor que el anterior.

5. Elementos de un Motor de Ajedrez

un árbol de variantes qué movimiento realizar frente a una posición dada que se encuentra en el nodo raíz. El algoritmo comienza calculando el valor numérico o puntuación para la posición final de cada variante. Estas posiciones se denominan posiciones terminales y su puntuación es calculada por una función de evaluación. La función de evaluación se encarga de medir cuán buena es la posición para el primer jugador. Si la puntuación es positiva significa que la computadora tiene ventaja y si por el contrario la puntuación es negativa es el rival quien tiene ventaja. El algoritmo entiende que el objetivo del primer jugador es llevar al juego a una posición en la cual se maximice su puntuación, mientras que el objetivo del rival es el opuesto, lograr una posición que sea mínima en puntuación.

Pasos del algoritmo minimax:

- Generar el árbol de juego. Se generarán todos los nodos hasta llegar a un estado terminal.
- Calcular los valores de la función de utilidad para cada nodo terminal.
- Calcular el valor de los nodos superiores a partir del valor de los inferiores. Dependiendo de si el nivel es MAX o MIN se elegirán los valores mínimos y máximos representando los movimientos de cada jugador, de ahí el nombre del algoritmo.
- Elegir la jugada dependiendo del valor que ha llegado al nivel superior.

El algoritmo explorará todos los nodos del árbol y les asignará un valor numérico gracias a la función de evaluación; se empieza por los nodos terminales y se sube hasta el nodo raíz. La función de utilidad es la que define lo buena que es la posición para un jugador cuando la alcanza. En el caso del ajedrez los posibles valores son (+1,0,-1) que se corresponden con ganar, empatar y perder respectivamente.

En la siguiente figura vamos a mostrar un ejemplo del algoritmo minimax donde el jugador MAX seleccionará las acciones que lo lleven a maximizar sus ganancias y el jugador MIN aquellas que lleven a minimizar lo máximo posible las ganancias de su adversario.

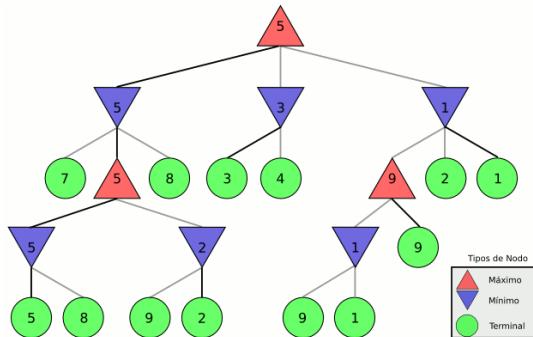


Figura 5.5.: Ejemplo del funcionamiento de minimax en un árbol. [87]

En determinados juegos de cálculo extenso como es el caso del ajedrez los programas examinan una parte del árbol de variantes sin llegar hasta el final del juego, es decir, llegan hasta un cierto límite de nodos, estimando en ellos la posibilidad que tiene cada bando de ganar mediante la función de evaluación.

Existen algunos algoritmos como la poda alfa-beta que ayudan a mejorar la eficiencia del

algoritmo minimax. Otras implementaciones incluyen la tabla de transposiciones las cuales guardan la información en posiciones que ya han sido examinadas. Posteriormente las analizaremos.

Por desgracia el algoritmo minimax es del orden $O(bn)$ donde b es el factor de anchura, número de movimientos legales disponibles en promedio en cualquier momento y n es la profundidad, número de movimientos que se calculan donde cada movimiento pertenece a un bando. Puesto que este número crece de forma exponencial a medida que aumenta la profundidad es necesario desarrollar algoritmos que minimicen el esfuerzo realizado en buscar movimientos a una determinada profundidad.

La manera más común de implementar este algoritmo en juegos de suma cero con acciones alternadas entre jugadores es mediante negamax que pasamos a explicar.

5.3.5.2. Negamax

Negamax es una variante del algoritmo minimax. Cada nodo toma el valor máximo de sus hijos independientemente de si fuese MIN o MAX, lo único que los valores de los nodos se tienen que cambiar de signo de un nivel a otro. Así obtenemos el mismo efecto.

La igualdad matemática en la que se basa negamax es la siguiente:

$$\min(\max(x_1, \dots, x_n), \max(y_1, \dots, y_n)) = -\max(\min(-x_1, \dots, -x_n), \min(-y_1, \dots, -y_n))$$

donde, x_1, \dots, x_n son los valores de utilidad de los sucesores de un nodo MAX, M_x , mientras que y_1, \dots, y_n son los valores de utilidad de los sucesores de un nodo MAX, M_y . El cálculo final es el valor de utilidad de un nodo MIN, cuyos sucesores son M_x y M_y . La operación de un nodo MIN es equivalente a la de un nodo MAX pero con signo opuesto.

Este algoritmo tiene la misma complejidad que el algoritmo básico minimax pero permite representaciones e implementaciones más sencillas. A este algoritmo al igual que a minimax se le pueden aplicar heurísticas y podas, una mejora de este algoritmo es negascout.

5.3.5.3. Algoritmo de Poda Alfa-Beta

El algoritmo alfa-beta es una mejora significativa del algoritmo de búsqueda minimax que elimina la necesidad de buscar en grandes porciones del árbol de juego aplicando una técnica de ramificación y acotamiento conocida como branch and bound. Si ya se ha encontrado una jugada bastante buena y se buscan alternativas encontrando algunas en las que un movimiento es peor que el mejor encontrado hasta el momento no es necesario dedicar tiempo en calcular cómo es de malo.

La búsqueda minimax es primero en profundidad, por ello en cualquier momento solamente se consideran los nodos a lo largo de un camino en el árbol. El algoritmo tiene este nombre ya que utiliza una ventana llamada alfa-beta, en la que emplea dos valores o parámetros que describen los límites sobre los valores hacia atrás que aparecen a lo largo del camino. Alfa (α) es el valor de la mejor opción hasta el momento a lo largo del camino para MAX, lo que implica la elección del valor más alto, el límite inferior. Beta (β) es el valor de la mejor opción

5. Elementos de un Motor de Ajedrez

hasta el momento a lo largo del camino para MIN, es decir, la elección del valor más bajo, el límite superior.

Esta búsqueda conforme se recorre el árbol va actualizando los parámetros de la ventana. El método realizará la poda de las ramas restantes cuando el valor actual que se está examinando sea peor que el valor actual de alfa o beta para MAX o MIN, respectivamente. Podemos ver un ejemplo en la siguiente figura.

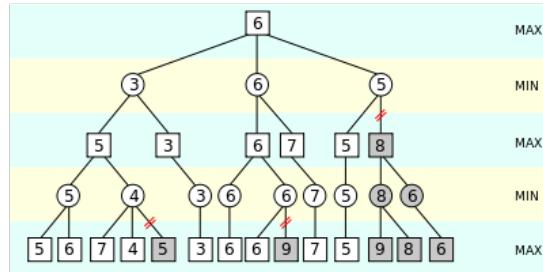


Figura 5.6.: Ejemplo de poda alfa-beta. [87]

Vamos a poner un ejemplo de una partida de ajedrez, supongamos que es el turno de las blancas y buscamos a profundidad 2, es decir, consideramos todos los movimientos de las blancas y todas las posibles respuestas de las negras a cada uno de los movimientos. En primer lugar escogemos una jugada posible por parte de las blancas que será la jugada número 1. Consideramos esta jugada y todas las posibles respuestas de las negras, una vez hecho esto consideramos que el resultado es una posición igualada. Ahora consideramos una segunda jugada, la jugada 2. Cuando consideramos esta jugada nos damos cuenta que una posible respuesta por parte de las negras es perder una torre. En esta situación, podemos ignorar con seguridad todas las demás respuestas posibles de las negras sobre la jugada 2 ya que sabemos que la posible jugada 1 es mejor. No nos interesa cómo de mala es la jugada 2 ni si hay posibles respuestas mejores sobre ella, lo que importa es que sabemos que la jugada 1 es mejor. El análisis completo de la posible jugada 1 nos aportó un límite inferior y como sabemos que podemos llegar a una posición igualada con ella cualquier cosa que sea claramente peor será ignorada.

Sin embargo, la situación se complica al pasar a una profundidad mayor, de 3 o más porque ahora ambos jugadores pueden efectuar movimientos que afecten al árbol de juego. Ahora es necesario mantener un límite inferior y un límite superior que son alfa y beta. Es necesario mantener un límite superior porque si una jugada en la profundidad 3 o superior conduce a una continuación que es demasiado buena, el otro jugador no hará ese movimiento para llegar a ella, ya que había una jugada mejor más arriba en el árbol de juego que podría evitar dicha posición. El límite inferior de un jugador es el límite superior del otro jugador.

En resumen, los cortes ocurren según las siguientes condiciones: en un nivel de maximización, los nodos sucesores con un valor más pequeño que el límite inferior serán cortados y en un nivel de minimización los nodos sucesores con un valor más grande que el límite superior serán cortados. Así, podemos podar estos subárboles completamente puesto que estas podas no afectan al algoritmo minimax.

El ahorro mediante la poda alfa-beta puede ser considerable. Si tenemos un árbol de búsqueda minimax estándar con x nodos al que le aplicamos la poda alfa-beta en un programa

bien escrito podemos llegar a tener un número de nodos cercano a la raíz cuadrada de x . Sin embargo el número de nodos que podemos llegar a podar depende realmente de lo bien que esté ordenado el árbol de juego.

Ahora veremos algunas posibles mejoras y optimizaciones de la poda alfa-beta que emplean los principales motores de ajedrez para encontrar las mejores jugadas de manera eficiente.

5.3.6. Optimizaciones y Mejoras

5.3.6.1. Profundidad iterativa

La profundidad iterativa (ID) ha sido adoptada como estrategia básica de gestión del tiempo en las búsquedas de *profundidad primero*⁶, pero ha demostrado ser beneficiosa en lo que respecta al ordenamiento de los movimientos en la poda alfa-beta y sus mejoras. Se ha comprobado que si se va a buscar hasta una profundidad dada, esa profundidad iterativa es más rápida que la búsqueda de la profundidad dada inmediatamente. Esto se debe a las técnicas de ordenación de movimientos dinámicos 5.3.6.6 que veremos de manera breve en este capítulo.

El problema es que los programas de ajedrez van a tener que tomar decisiones en un espacio de tiempo y puede que no hayan conseguido decidir qué movimiento realizar. Como es imposible determinar el tiempo que tardará una búsqueda es necesario que sepa qué jugada realizar.

La profundidad iterativa es una estrategia que pretende solucionar este problema. El funcionamiento es el siguiente, el programa empieza con una búsqueda de una capa, posteriormente incrementa la profundidad de búsqueda y realiza otra búsqueda. El proceso se repite hasta que se agota el tiempo asignado a la búsqueda o se alcanza la profundidad deseada. El programa tiene la opción de volver al movimiento seleccionado en la última iteración de la búsqueda si esta no finaliza. Sin embargo, si nos aseguramos de que este movimiento se busque primero en la siguiente iteración no será necesario sobreescribir el nuevo movimiento con el anterior. Por tanto, también se pueden aceptar los resultados de la búsqueda parcial, aunque en caso de caída severa de la puntuación se le debe asignar algo más de tiempo, puesto que la primera alternativa suele ser una mala captura, retrasando la pérdida en lugar de evitándola. Así al realizar la búsqueda de manera incremental tendremos una estimación de un buen movimiento para una profundidad previa y si se termina el tiempo de búsqueda el programa sabrá qué movimiento realizar.

La profundidad iterativa utiliza una tabla de transposición 5.3.6.3 que incrusta los algoritmos de profundidad primero como es el caso de alfa-beta en un marco con características de la búsqueda primero el mejor 5.3.8 que veremos más adelante.

5.3.6.2. Búsqueda de quietud

La mayoría de los programas de ajedrez realizan una búsqueda silenciosa o de reposo al final de la búsqueda principal. Esta búsqueda es más limitada y contiene menos movimientos. El propósito de esta búsqueda es evaluar solo posiciones tranquilas o posiciones en las que no

⁶La búsqueda de profundidad primero comienza en la raíz y explora todo lo posible a lo largo de cada rama antes de retroceder, como es el caso de minimax.

5. Elementos de un Motor de Ajedrez

hay jugadas tácticas ganadoras que hacer, tanto jugadas que no alteran el material, es decir, en las que no hay ni capturas ni ascensos como jugadas de jaque. Esta búsqueda es necesaria para evitar el problema irresoluble que sufren todos los algoritmos de búsqueda en árboles de juego, el efecto horizonte. Detener la búsqueda cuando se alcanza la profundidad deseada y luego evaluar es muy peligroso como ya hemos visto previamente, de ahí la necesidad de asegurarse de que se están evaluando posiciones tranquilas. Por este motivo, el paso de la búsqueda alfa-beta a la función de evaluación se lleva a cabo de manera dinámica, de manera que se espera una constancia aproximada por debajo de la profundidad de búsqueda mínima con respecto a la función de evaluación. Así es como analizan hasta el final el intercambio de golpes en una partida de ajedrez como lo haría un jugador humano.

5.3.6.3. Tabla de transposición

Una tabla de transposición es una base de datos que almacena los resultados de las búsquedas realizadas anteriormente. Es una forma de reducir el espacio de búsqueda de un árbol de ajedrez en gran medida y teniendo poco impacto negativo. Los programas de ajedrez durante su búsqueda por fuerza bruta se encuentran con las mismas posiciones una y otra vez, pero a partir de diferentes secuencias de movimientos, lo que se denomina *transposición*⁷. Las tablas de transposición y las de refutación que no comentaremos ya que no se usan actualmente son técnicas que provienen de la programación dinámica.

El funcionamiento consiste en recordar lo que se determinó la última vez que se examinó esa misma posición en vez de realizar toda la búsqueda de nuevo. Por este motivo, los programas de ajedrez tienen una tabla de transposiciones, que no es más que una tabla hash que almacena información sobre las posiciones buscadas anteriormente, la profundidad a la que se buscaron y la conclusión que se tuvo sobre ellas. Incluso si la profundidad de la tabla de transposición relacionada no es lo suficientemente grande, o no contiene el límite correcto para un corte, una jugada mejor o lo suficientemente buena de una búsqueda anterior puede mejorar el orden de las jugadas y ahorrar tiempo de búsqueda. Esto se cumple en un marco de profundización iterativa, en el que se obtienen valiosos aciertos en la tabla a partir de iteraciones anteriores.

Las funciones hash convierten posiciones de ajedrez en una firma escalar casi única, lo que permite un cálculo rápido de los índices, así como una verificación de las posiciones almacenadas que ahorra espacio. Algunos ejemplos son el hashing Zobrist o el hashing BCH que utilizan funciones hash rápidas para proporcionar claves hash o firmas como una especie de *número de Gödel*⁸ de posiciones de ajedrez. Se actualizan de manera incremental.

El objetivo principal de los códigos hash de Zobrist en la programación de ajedrez es obtener un número de índice casi único para cualquier posición de ajedrez de manera que dos posiciones similares generen índices totalmente diferentes. Estos números de índices se utilizan para obtener tablas hash más rápidas y eficientes en términos de espacio, por ejemplo, las tablas de transposición que acabamos de explicar.

⁷En ajedrez, una transposición es una posición en el tablero que puede ser alcanzada por más de una línea o camino.

⁸En lógica matemática, una numeración de Gödel es una función que asigna a cada símbolo y fórmula bien formada de algún lenguaje formal un número natural único.

5.3.6.4. Ventana de aspiración

Una manera de reducir el espacio de búsqueda en una búsqueda alfa-beta son las ventanas de aspiración. La técnica consiste en acotar los límites de alfa y beta para producir mayor cantidad de podas. Al ser la ventana más estrecha se consiguen más límites y la búsqueda tarda menos tiempo. El problema surge si la puntuación verdadera está fuera de esta ventana, ya que habría que hacer una rebúsqueda con una ventana mayor que es muy costosa.

5.3.6.5. Búsqueda de variación principal

La búsqueda de variación principal (PVS) es una mejora de la poda alfa-beta, está basada en búsquedas de *ventanas nulas*⁹ o nodos que no son *PV-nodos*¹⁰ para probar que una jugada es peor o no que una puntuación ya segura de la *variación principal*¹¹.

En la mayoría de los nodos solo necesitamos un límite que demuestre que una jugada no es buena para nosotros o para el adversario y no la puntuación exacta. Esto es necesario solo en la llamada variación principal. Si una búsqueda de menor profundidad ya ha establecido dicha secuencia encontrando una serie de jugadas cuyo valor es mayor que alfa pero menor que beta en toda la rama, lo más probable es que no sirva para nada desviarse de ella. Así que en un PV-nodo solo se busca en la ventana completa el primer movimiento (el que se considera mejor por la iteración anterior de un marco de profundización iterativo) para establecer el valor esperado del nodo.

Cuando ya tenemos el movimiento PV que es el definido como el movimiento que elevó el valor de alfa en un PV-nodo asumimos que vamos a seguir con él. Para corroborar que es cierta nuestra suposición, se realiza una búsqueda de la ventana nula o cero centrada en alfa para comprobar si una nueva jugada puede ser mejor. Si es así, con respecto a la ventana nula pero no con respecto a la ventana completa, tenemos que hacer una nueva búsqueda con la ventana normal completa. Puesto que las búsquedas con la ventana nula son más baratas si tenemos una buena ordenación de los movimientos nos ahorraremos en torno a un 10 % del esfuerzo de búsqueda.

La mayoría de las implementaciones son empleadas por la mayoría de los programas de ajedrez actuales. Se basa en la precisión del ordenamiento de las jugadas. Normalmente los programas modernos de ajedrez encuentran fallos en la primera jugada en alrededor del 90 % de las ocasiones

5.3.6.6. Ordenación de movimientos

El orden de generación de jugadas de una dada es uno de los factores que más influyen en la poda alfa-beta. Si el orden es perfecto, la poda alfa-beta podrá explorar casi el doble de nodos que el algoritmo minimax en el mismo tiempo.

⁹En la poda alfa-beta se dice que la ventana es nula cuando el valor de alfa y el de beta son iguales.

¹⁰Los PV-nodos son nodos que tienen una puntuación que acaba estando dentro de la ventana. Así, si la puntuación del nodo es s y los límites de la ventana son $[a, b]$ entonces $s \in [a, b]$. Estos nodos tienen todos los movimientos buscados y el valor devuelto es exacto, no un límite, que se propaga hasta la raíz junto con una variación principal.

¹¹Una secuencia de movimientos aceptables para ambos jugadores, (es decir, que no provoquen un corte beta en ningún camino) que se espera que se propague hasta la raíz.

5. Elementos de un Motor de Ajedrez

Las jugadas se pueden ordenar de la siguiente manera, pueden generarse al azar y entonces no tendríamos orden o pueden tener un cierto orden mediante un parámetro denominado profundidad de cambio que establece la profundidad a partir de la cual ya no se ordenan las jugadas. Hay dos tipos:

- *Orden estático*: el orden de las jugadas siempre es el mismo y no se cambia el orden mientras se estudian los nodos sucesores.
- *Orden dinámico*: las jugadas se ordenan y se van estudiando una a una. Si alguna devuelve un valor distinto al esperado y no se han estudiado todavía todas las jugadas se estudia si se reordenan o no todas las jugadas no exploradas.

Para el juego del ajedrez se suelen examinar antes los movimientos que capturan piezas que los que no lo hacen y también aquellos movimientos que obtuvieron una puntuación alta. Vamos a pasar a ver ahora una de las heurísticas más conocidas, la heurística asesina (Killer Heuristic), hay otras como heurística histórica (History Heuristic) generalización de la anterior, Mate Killers, etc, en las que no vamos a profundizar.

Heuristica asesina. Esta técnica de ordenación de movimientos dinámica mejora la eficiencia de la poda alfa-beta y depende de la trayectoria. Se basa en la ordenación de que una jugada fuerte o un pequeño conjunto de ellas en una determinada posición puede ser igualmente fuerte en posiciones similares en el mismo movimiento en el árbol de la partida. Esta heurística intenta producir un corte asumiendo que una jugada que produjo un corte en otra rama del árbol de juego en la misma profundidad es probable que produzca un corte en la posición actual. Conservar este tipo de jugadas llamadas jugadas asesinas evita redescubrirlas en los nodos hermanos y se ordenan en lo alto de la lista para probarlas antes que otras jugadas. Así, un programa podría producir un corte temprano y ahorrarse un esfuerzo considerable en comprobar otra serie de jugadas. Cuando un nodo falla en lo alto, un movimiento silencioso que causó un corte se almacena en una tabla indexada por capa, que suele contener dos o tres movimientos por capa.

5.3.7. Búsqueda en Anchura

La búsqueda en anchura, en inglés BFS, es un algoritmo de búsqueda no informada para buscar en una estructura de datos en forma de árbol un nodo que satisfaga una determinada propiedad. El algoritmo comienza en la raíz del árbol y se van explorando los vecinos de este nodo. Cuando se han explorado todos los vecinos para cada uno de ellos se exploran los respectivos vecinos adyacentes y así hasta recorrer el árbol entero. Normalmente se implementa con una cola FIFO y se necesita memoria adicional para llevar la cuenta de los nodos hijos que se han encontrado pero que aún no se han explorado. Este algoritmo no usa ninguna estrategia heurística.

Al igual que hicimos para la búsqueda en profundidad vamos a ilustrar con un ejemplo el procedimiento para recorrer un árbol mediante la búsqueda en anchura, el recorrido se realiza en orden numérico de forma consecutiva.

El procedimiento que sigue el algoritmo es el siguiente:

- Partimos del vértice fuente s y exploramos los vértices del grafo G para descubrir aquellos vértices alcanzables desde s .

- Calculamos la distancia (menor número de vértices) desde s a todos los vértices alcanzables.
- Así obtenemos un árbol con raíz en s y que contiene a todos los vértices alcanzables.
- El camino desde el vértice fuente s a cada vértice en este recorrido es el camino más corto medido en número de vértices.
- El nombre de este algoritmo se debe a que expande de manera uniforme la frontera entre lo descubierto y lo no descubierto. Si no hemos llegado a todos los nodos de distancia $k - 1$ no podemos llegar a los de distancia k .

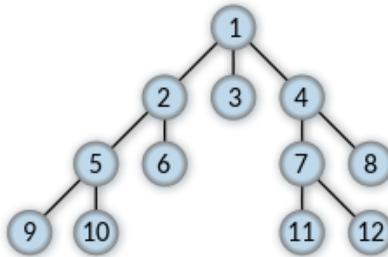


Figura 5.7.: Búsqueda en anchura.[\[87\]](#)

5.3.8. Búsqueda Primero el Mejor

La búsqueda primero el mejor, en inglés Best-First Search, es una búsqueda en el espacio de estados para recorrer los nodos de estructuras de datos en forma de árbol, es decir, en árboles de búsqueda como la búsqueda en anchura. Se implementa con una cola de prioridad en vez de una cola FIFO, para expandir primero el nodo más prometedor de cada nivel. Este método convierte la búsqueda en anchura no informada en una *búsqueda informada*¹². La complejidad de espacio y los requisitos de memoria son proporcionales al número de nodos del nivel más profundo, ya que todos los nodos de un nivel deben guardarse hasta que se hayan generado sus nodos hijos en el siguiente nivel.

Vamos a nombrar algunos de los algoritmos de búsqueda primero el mejor que fueron inventados e implementados para programas de ajedrez por ordenador, así como para otros juegos de mesa de suma cero para dos jugadores con información perfecta: B^* , búsqueda minimax primero el mejor (LCF), búsqueda en árboles de Monte Carlo (MCTS), UCT, etc.

Mostramos un ejemplo de búsqueda primero el mejor donde vemos cómo se exploran los nodos escogiendo el que tenga menor coste según la heurística elegida, ya que es una búsqueda informada.

5.3.8.1. Búsqueda en Árboles de Monte Carlo

La búsqueda en árboles de Monte Carlo es un algoritmo de búsqueda primero el mejor basado históricamente en jugadas aleatorias, es un algoritmo de búsqueda heurístico para

¹²También llamada búsqueda heurística, este tipo de búsqueda cuenta con información adicional del problema que busca resolver el agente, es por esto que puede encontrar las soluciones de manera más eficiente.

5. Elementos de un Motor de Ajedrez

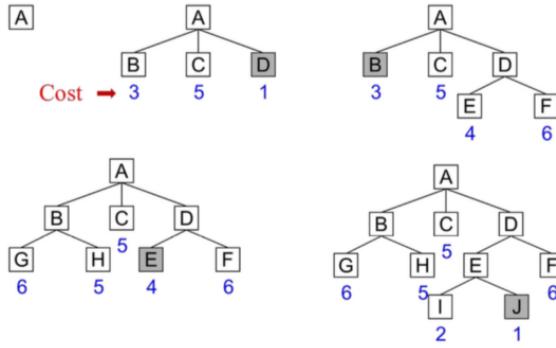


Figura 5.8.: Ejemplo búsqueda primero el mejor. [34]

algunos tipos de procesos de toma de decisiones como los juegos. La búsqueda en árboles de Monte Carlo se basa en exploraciones aleatorias del espacio de búsqueda. Utilizando los resultados de exploraciones previas, el algoritmo hace crecer de forma gradual un árbol de juego en la memoria y se va volviendo mejor de manera sucesiva para estimar con precisión los valores de las jugadas más prometedoras.

Algunas ventajas que ofrece este método es que se puede aplicar a juegos con un factor de ramificación muy grande, se puede configurar para que se detenga tras una cantidad de tiempo prefijada. No requiere de un conocimiento previo del juego y es adaptable a juegos que incorporan aleatoriedad en las reglas. Sin embargo, presenta algunas desventajas como que es un método probabilístico, por lo que no siempre tiene que encontrar la solución óptima y realizar las simulaciones de jugadas es costoso, convirtiendo el proceso completo a veces en inabordable.

En un proceso estándar de Monte Carlo aplicado a un juego se realizan las siguientes acciones:

- En primer lugar, se generan un gran número de simulaciones aleatorias a partir de la posición del tablero para la que se busca encontrar el mejor movimiento.
- En segundo lugar, se almacenan las estadísticas de cada movimiento posible a partir de la posición inicial.
- Por último se devuelve el movimiento con los mejores resultados generales.

El inconveniente de este método general es que, por cada turno puede haber muchos movimientos posibles pero unos pocos buenos, por lo que eligiendo de manera aleatoria alguno de ellos es improbable que la simulación encuentre el mejor camino hacia la solución.

Ante esto, se han propuesto variantes como la llamada cota de confianza superior aplicada a árboles, en inglés Upper Confidence bounds applied to Trees, conocida como UCT. UCT supone una mejora que permite minimizar riesgos y que está basada en la idea de que si tenemos estadísticas de bondad disponibles para todos los estados sucesores del estado actual entonces podemos extraer una estadística de bondad para el mismo.

En lugar de efectuar muchas simulaciones de manera aleatoria, esta variante hace muchas iteraciones de un proceso formado por varias fases y que tiene como objetivo mejorar nuestra información del sistema, exploración, a la vez que potenciar las opciones más prometedoras,

explotación. Cada ronda consta de cuatro fases, las cuales presentamos a continuación junto con una figura para facilitar la comprensión:

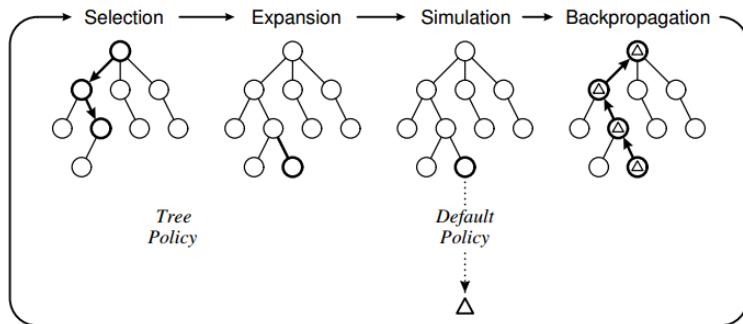


Figura 5.9.: Fases. [32]

- *Selección*: comenzando en el nodo raíz se selecciona recursivamente el nodo más urgente según una función de utilidad hasta que se alcanza un nodo que representa un estado terminal o no está completamente extendido (nodo en el que faltan posibles movimientos o acciones por considerar).
- *Expansión*: se elige una posición sucesora no visitada del nodo que no estaba completamente extendido y se añade un nuevo nodo hoja al árbol.
- *Simulación*: en esta fase se simula una partida completa, ya sea al azar o mediante heurísticas y evaluaciones computacionalmente costosas para una estrategia más elaborada. Junto a la partida completa se obtiene un valor, en el caso del ajedrez el resultado es 1, 0 o -1, que determina la utilidad de la rama.
- *Actualización o Backpropagation*: una vez hemos alcanzado el estado final del juego se actualizan todas las estadísticas de todas las posiciones previas visitadas durante la simulación completa que se ejecutó a partir del nuevo nodo, es decir, se propagan los resultados a través del árbol.

5.3.9. Búsqueda Paralela

La búsqueda paralela conocida también como búsqueda SMP o búsqueda multihilo es una forma de aumentar la velocidad de búsqueda utilizando procesadores adicionales. Con la generalización de los ordenadores con multiprocesadores este tema ha ido ganando importancia y popularidad. La utilización de estos procesadores adicionales es un campo de investigación interesante puesto que el recorrido de un árbol de búsqueda es intrínsecamente serial. Se han ideado varios enfoques, los más populares en el caso del ajedrez son Lazy SMP conocida como búsqueda paralela perezosa que emplea la tabla hash compartida, la idea de los hermanos Young llamada YBWC, Principal Variation Splitting conocido en español como división de la variación principal o DTS.

5.4. Función de Evaluación

La función de evaluación ([24], [15]) es una de las partes más importantes de un programa de ajedrez, ya que determina la fuerza de juego de cada programa. Esta función heurística se emplea para determinar el valor relativo de una posición, es decir, las posibilidades de ganar la partida. En el caso del ajedrez [16] si pudiéramos ver el final de la partida los resultados serían 1 victoria, 0 empate y -1 derrota y el motor de ajedrez buscaría hasta encontrar la jugada que nos permita ganar la partida. En la práctica, no sabemos el valor exacto de una posición y por eso es necesario emplear una función de aproximación, puesto que no existe la posibilidad de explorar a través del árbol de juego del ajedrez todos los movimientos posibles según una posición dada. La función de evaluación normalmente es una función multivariante que mide cómo de buena es la posición, donde cada variable de la función representa un factor que caracteriza la posición. Debemos señalar que es muy complicado indicar las características más importantes de una posición y todavía más, determinar cómo de importantes son en cada situación de la partida. Cuanto mejor sea capaz de seleccionar el programa las características de una determinada posición mayor será su poder.

La posición de ajedrez queda determinada por el conjunto de piezas blancas y negras situadas en ciertas posiciones del tablero de ajedrez. Cada pieza tiene diferente valor y se le asigna a cada una un determinado valor, dependiendo de la posición que ocupa cada pieza tendrá mayor o menor importancia. La diferencia que hay entre las piezas de un bando con respecto al otro se llama balance material. Esta característica es una de las más importantes para la función de evaluación y suele considerarse sobre las demás, debido a que todas las posibles características de la posición dependen directamente de las piezas. El resto de las características importantes a tener en cuenta son llamadas estratégicas o posicionales y algunas de ellas son control del tablero y del centro, estructura de peones, movilidad, espacio, seguridad del rey, etc.

Actualmente, hay dos formas principales de construir una evaluación, la evaluación tradicional hecha a mano y las hechas por las redes neuronales multicapa. Una función de evaluación viene definida como $f : A \rightarrow [-\text{Mate}, \text{Mate}]$, donde A representa el conjunto de todas las posibles posiciones que se pueden dar en el ajedrez. Dada una posición del tablero p , para realizar la evaluación de esta se determinan las características que van a ser consideradas y se calcula el valor de cada una de ellas. La función de evaluación más sencilla y más empleada es:

$$f(n) = \sum_{i=1}^N X_i(n) \cdot W_i(n)$$

donde N representa el número de características consideradas, $X_i : A \rightarrow \{0,1\}$ es una función de decisión, que considera si el valor W_i se tiene en cuenta en la evaluación, así podemos realizar evaluaciones diferentes dependiendo del momento de la partida. W_i es una función de evaluación parcial que tiene en cuenta una única característica de la posición.

Gracias a esta estructura de la función de evaluación se pueden agregar nuevas características conforme aumenta el conocimiento del juego o probar varias combinaciones. El número de características a tener en cuenta puede variar y dependerá de la robustez deseada para la función de evaluación. A menudo se realizan varias fases para procesar primero las características simples y una vez se han construido las estructuras de datos adecuadas se

utilizan en fases consecutivas características más complejas basadas en patrones y trozos. Una posición será mejor representada cuanto mayor sea el número de características que intervienen en el proceso de evaluación de esa posición.

Las características principales para estimar cómo de buena es una posición de ajedrez son las siguientes:

- *Balance material.*
- *Control del tablero, movilidad, espacio y tiempo.*
- *Estructura de peones.*
- *Seguridad del rey.*

Debemos tener en cuenta que cada programa de ajedrez evaluará cada característica con un valor diferente que dependerá de su función de evaluación.

5.4.1. Balance material

Cada una de las piezas en ajedrez tiene un determinado valor que va variando durante la partida dependiendo de una serie de características. Estos valores se utilizan como heurísticas para ayudar a determinar cómo de importante o valiosa es una pieza tanto estratégicamente como tácticamente. En ajedrez intentamos capturar piezas de nuestro oponente preservando las nuestras hasta poder realizar jaque mate y ganar la partida. También intentaremos localizar la pieza más valiosa de nuestro oponente y conseguir eliminarla para ir obteniendo cierta ventaja a lo largo de la partida.

Una forma sencilla de asignar valores a piezas específicas en casillas específicas es emplear tablas de casillas y piezas. Se crea una tabla para cada pieza de cada color y se asignan valores a cada casilla. Este esquema es rápido, puesto que el término de evaluación de las tablas de casillas y piezas puede actualizarse de forma incremental a medida que se van efectuando movimientos en el árbol de búsqueda.

Para contabilizar el valor de las piezas se emplea un valor, centipawn (cp), en español centipeones, que es un valor de medida mínimo. El centipeón es la unidad de medida utilizada en ajedrez como medida de la ventaja. Un centipeón tiene el valor de $\frac{1}{100}$ peones, es decir, cien centipeones equivalen a un peón. Estos valores no desempeñan ningún papel formal en el juego pero son útiles para los jugadores e imprescindibles en el ajedrez computacional para evaluar posiciones. El valor de cada pieza, los valores posicionales y tácticos, así como el resto de características que hay en una posición de ajedrez se miden en términos de este valor de medida. El valor de las piezas en esta unidad depende del programa de ajedrez que estemos utilizando y no solo depende de la pieza, ya que una pieza de menor calidad puede tener mayor valor si se encuentra colocada en una posición estratégica.

5.4.2. Control del tablero, movilidad, espacio y tiempo

En ajedrez es vital controlar el mayor número de casillas en el tablero para obtener ventajas significativas, ya que tendremos más opciones para atacar y defender y nuestro adversario tendrá menos. Esto es lo que se conoce como control del tablero. Normalmente, se busca una estrategia para controlar u ocupar el centro del tablero y también se pretende controlar el

5. Elementos de un Motor de Ajedrez

centro extendido del tablero, ya que así obtenemos una zona de suma importancia posicional. Al controlar el centro conseguimos mayor espacio y permitimos a las piezas un acceso rápido a la mayoría de las áreas del tablero mejorando nuestras posibilidades en ataque y en defensa. En la evaluación, el control del centro se tiene en cuenta en las tablas de casillas y piezas y en la movilidad, donde la ocupación y las áreas controladas se ponderan de manera acorde.

La movilidad de las piezas está relacionada con el control del tablero, es una medida del número de opciones, movimientos legales, que tiene un jugador en una posición determinada. Se utiliza de manera frecuente en la función de evaluación de los programas de ajedrez. Se basa en la idea de que a más opciones de movimiento, más fuerte será la posición.

Uno de los aspectos que está relacionado tanto con el control del tablero como con la movilidad es el espacio. Es una característica de evaluación relacionada con el control del tablero, en particular con el control del centro considerado por la colocación de piezas que depende de la estructura de los peones. Se dice que el jugador que controla más casillas es el que tiene ventaja espacial. Algunos programas como Stockfish tienen términos de evaluación explícitos relativos al espacio. Otro de los aspectos a tener en cuenta es el tiempo, que se refiere a un turno o a una sola jugada, y está relacionado con la movilidad de las piezas. Si un jugador consigue un resultado en un movimiento menos gana un tiempo y si tarda más pierde uno o varios tiempos. Así, si por ejemplo en una apertura se han ganado varios tiempos en el desarrollo de las piezas con respecto al rival obtendremos cierta ventaja que tienen en cuenta las funciones de evaluación.

Algunas de las características relacionadas con el control del tablero, la movilidad, el espacio y el tiempo son:

- *Posición de las piezas*: Cada pieza recibe una determinada puntuación dependiendo de la casilla en la que se encuentre independientemente de dónde estén las demás piezas, para la evaluación emplea tablas de piezas cuadradas. Este término de evaluación permite, por ejemplo, que el motor se anime a avanzar peones, desarrollar alfiles y caballos, etc.
- *Piezas atrapadas*: las piezas atrapadas son los ejemplos extremos de mala movilidad. Son piezas que están colocadas de forma incómoda y normalmente se encuentran en peligro y puede que sean capturadas produciendo una situación de desventaja en la posición. Un ejemplo típico de pieza atrapada es un alfil blanco en *h7* bloqueado por los peones negros de *f7* y *g6*. La mayoría de las veces terminará siendo capturado y seguramente lo mejor será sacrificarlo por un peón.
- *Casillas controladas*: cuanto mayor sea el número de casillas controladas por nuestras piezas mayores posibilidades de victoria tendremos.
- *Control del centro*: como dijimos previamente, controlar el centro permite en muchas ocasiones obtener una ventaja que puede ser determinante. La función de evaluación otorgará cierta puntuación dependiendo del control del centro y de qué piezas lo controlan.
- *Conexión de las piezas*: si las piezas no se encuentran protegidas entre sí aparecerán puntos débiles en la posición que podrán ser atacados por el oponente. Cada pieza protegida aumentará el valor de la posición dependiendo de la función de evaluación.
- *Torres en columnas abiertas o semiabiertas*: una columna abierta se suele definir como una columna que no tiene peones y una columna semiabierta como aquella en la que

hay solo peones enemigos. En ambos casos una torre que ocupe dicha fila obtiene una mayor movilidad vertical, así como la posibilidad de penetrar en campo contrario posicionándose en la séptima fila. La bonificación típica para una columna semiabierta es la mitad de la de una columna abierta.

- *Conexión de torres en la misma fila*: dos torres se encontrarán conectadas en la misma fila cuando se encuentren en la misma fila y no haya piezas entre ellas. Esta configuración suele ser más poderosa si se encuentran en el campo rival.
- *Torres dobladas*: dos torres se encuentran dobladas cuando se encuentran conectadas en la misma columna y no existe ningún tipo de pieza entre ellas. Esta configuración permite que se protejan entre ellas y aumentar el poder de ataque ambas. Obtienen mayor bonificación si la fila está abierta y si alguna de ellas se encuentra en la séptima fila.
- *Torre en séptima*: si conseguimos infiltrar una torre en líneas enemigas llegando a la séptima fila en numerosas ocasiones obtendremos ventaja, ya que podremos atacar los peones sin que puedan defenderse entre sí y también ejerceremos presión al rey del oponente.
- *Pareja de alfiles*: contar con la pareja de alfiles en vez de contar con un caballo y un alfil o con los dos caballos supone en la mayoría de las ocasiones cierta ventaja sobre todo en posiciones abiertas.
- *Fianchetto*: el fianchetto es un patrón en el que un alfil se desarrolla hasta la segunda fila de la columna del caballo adyacente habiendo movido el peón de caballo normalmente una o raramente dos casillas hacia delante. Especialmente cuando el fianchetto es parte de la formación defensiva del rey, se concede una bonificación para evitar un intercambio, especialmente contra un caballo.
- *Puestos de avanzada*: este término está relacionado con los caballos en el centro o en la mitad del tablero del oponente, defendidos por un peón propio que no puede ser atacado por los peones enemigos. Este tipo de posiciones permite en ocasiones alcanzar cierta ventaja.

5.4.3. Estructura de peones

La estructura de peones ([49], [16]) es un término utilizado para describir las posiciones de todos los peones en el tablero, ignorando el resto de piezas. La estructura de peones abarca una amplia gama de ideas, desde la forma general de los peones hasta las características de los peones individuales. Es considerada una de las características más importantes para saber si una posición es fuerte o débil, ya que puede significar una ventaja significativa para un jugador.

La evaluación de estas estructuras resulta ser muy importante para la función de evaluación, puesto que los peones solamente pueden moverse hacia delante y no en múltiples direcciones. La evaluación completa de la estructura de peones puede resultar bastante costosa. Muchos programas emplean una tabla hash de peones separada para acelerar la evaluación, está dirigida por una clave especial derivada de la posición del peón (y posiblemente también del rey como veremos posteriormente). Dado que la estructura de los peones cambia con bastante lentitud, el porcentaje de aciertos de dicha tabla suele ser superior al 95 %. Cualquier cosa

5. Elementos de un Motor de Ajedrez

relacionada estrechamente con los peones puede ser almacenada en esta tabla hash, incluida la estructura de peones que mantiene protegido al rey.

Ahora presentamos las estructuras de peones más importantes que tienen en cuenta los programas de ajedrez para evaluar una posición de ajedrez:

- *Peones doblados*: dos peones del mismo color están doblados si se encuentran en la misma columna. Los peones se doblan en la columna que capturan una pieza enemiga y en la mayoría de los casos son considerados una debilidad debido a la incapacidad que tienen para defenderse entre sí.
- *Peón retrasado*: un peón retrasado es aquel que se encuentra adyacente detrás de otro peón del mismo color y no es defendible por los peones propios. Este tipo de peones son difíciles de defender al encontrarse detrás del resto de peones por lo que se consideran como una desventaja posicional. También son una desventaja, ya que el oponente puede colocar una pieza enfrente del peón (suele ser un caballo) sin riesgo de que un peón lo expulse.
- *Peón aislado*: el peón aislado es un peón sin peones del mismo color en las columnas vecinas y suele ser fácil de atacar, por eso normalmente se le evalúa con mayor penalización que al peón retrasado. Puede ser más débil si se encuentra en una columna semiabierta. Muchos programas evalúan un peón aislado en el centro más débil que en el ala, puesto que puede ser atacado desde más direcciones.
- *Peón pasado*: un peón pasado es un peón que no tiene peones contrarios delante en la misma columna o en columnas adyacentes. En el caso de los peones doblados no es correcto tener en cuenta al peón trasero como pasado. Un peón pasado suele ser de gran ventaja, puesto que la única forma de pararlo y evitar que corone será con el resto de piezas y así reduciremos su potencial.
- *Cadena de peones*: la cadena de peones está formada por una serie de peones colocados en la misma diagonal, sin interrupción, de manera que solo uno de ellos queda sin defender. Este peón es conocido con el nombre de base de la cadena de peones y será el peón retrasado.
- *Isla de peones*: las islas de peones son grupos de peones (al menos uno), que no se encuentran conectados al resto. En igualdad de condiciones, el bando con menos islas de peones tiene ventaja.
- *Peones ligados*: los peones ligados son dos o más peones en columnas adyacentes (si se encuentran en la misma fila se llama *falange*) protegiendo mutuamente sus casillas de parada.
- *Peones colgantes*: los peones colgantes son un dúo abierto y medio aislado, es decir, dos peones unidos uno al lado del otro que no tienen peones en las columnas adyacentes. El bando con peones colgados suele tener ventaja de espacio. La mayoría de los programas no tienen una evaluación especial de los peones colgantes.

5.4.4. Seguridad del rey

La tarea principal en ajedrez es proteger a tu rey y dar caza al rey enemigo, por eso mantener bajo seguridad a tu rey es un criterio de suma importancia y a veces de mayor relevancia

que el valor material. Una buena evaluación de la seguridad del rey es una de las tareas más complejas al escribir una función de evaluación, pero también la más gratificante. Existen numerosas maneras de proteger al rey y que van variando conforme avanza la partida. Normalmente se busca que esté protegido por una buena estructura de peones que hacen de escudo ante los ataques enemigos y que el resto de piezas se encuentren relativamente cerca para defender a su rey en caso de ataque enemigo.

La seguridad del rey es primordial a lo largo de la partida pero especialmente importante es en el medio juego. Una vez se han capturado la mayoría de piezas y hemos llegado al final de la partida, el rey se convierte en una de las piezas más efectivas e importantes para atacar y proteger, por lo que se intenta que el rey consiga una posición centralizada.

Algunas de las características o heurísticas que se emplean con más frecuencia son:

- *Enroque*: este movimiento especial que comentamos previamente [3.4.4.2](#) permite que el rey se encuentre en una posición de mayor seguridad cerca de una esquina, disminuyendo así la cantidad de flancos por donde puede ser atacado. Si el rey no se puede enrocar recibe una penalización, obtiene más valoración si se puede enrocar en ambos lados que si solo se puede enrocar en uno de los flancos.
- *Escudo de peones*: es muy importante que el rey tenga una buena estructura de peones protegiéndolo, ya que son la principal línea defensiva de este sobre todo tras el enroque. En general, lo mejor es mantener los peones sin mover o posiblemente desplazados una casilla hacia arriba. La falta de un peón de protección conlleva una penalización, más aún si hay una columna abierta junto al rey.
- *Tormenta de peones*: si los peones enemigos se encuentran cerca del rey enroulado, es peligroso abrir una columna, incluso si el escudo de peones está intacto. Las penalizaciones por asaltar peones enemigos deben ser menores que las penalizaciones por columnas semiabiertas, puesto que los peones enemigos podrían bloquear la posición y acabar protegiendo al rey enemigo.
- *Zona del rey*: la zona del rey suele definirse como aquellas casillas a las que puede moverse el rey, más dos o tres casillas adicionales frente a la posición del enemigo. Los esquemas más elaborados de evaluación de la seguridad del rey recogen información sobre el control de las casillas cercanas al rey.
- *Tropismo de rey*: el tropismo de rey es una forma simplificada de evaluación de la seguridad del rey. Tiene en cuenta la distancia entre el rey y las piezas atacantes, ponderada posiblemente en función del valor de las piezas. Este tipo de evaluación se lleva a cabo de forma probabilística, no es seguro que estar cerca del rey ayude a atacarlo. Por ejemplo, si las blancas se enrocan en corto, la torre negra situada en *h8* obtiene un valor de tropismo más alto, independientemente de si la columna está abierta.
- *Amenazas*: el rey no se debe encontrar en la línea de ataque de piezas enemigas. Aunque el ataque no sea directo y haya una pieza interponiéndose, esa línea puede abrirse en cualquier momento dejando al rey bajo ataque u obligándolo a mover a ciertas posiciones desfavorables. Existe una heurística que se basa en sustituir temporalmente al rey por una dama para emplear su movilidad virtual como medida de las posibilidades de ataque de las piezas deslizantes del adversario y así comprobar cómo de inseguro se encuentra el rey.

6. Motores de Ajedrez

En este capítulo se define el concepto que se conoce como motor de ajedrez. También se presentan algunos de los más famosos, donde se analizan sus elementos, características, la forma en la que han evolucionado con el paso de los años, en qué destacan, etc. Los cuatro motores de ajedrez seleccionados son: Stockfish, Leela Chess Zero, Nemorino y Ethereal.

6.1. Motores de Ajedrez

Un motor de ajedrez ([11], [24]) es un programa de computadora que sabe jugar al ajedrez, abstrae una posición de ajedrez, encuentra todos los movimientos candidatos, itera a través del árbol de candidatos hasta una profundidad determinada y evalúa la relevancia de esos movimientos para encontrar el mejor. El motor es la inteligencia artificial o parte pensante de un programa de ajedrez y está compuesto fundamentalmente por tres partes como hemos visto, el generador de movimientos que requiere una codificación del tablero, la función de búsqueda y la función de evaluación.

La interfaz de usuario o GUI permite a un jugador enfrentarse a un motor en un ambiente gráfico agradable y también que se puedan enfrentar diferentes motores, funcionan como frontend. Las interfaces más famosas son: XBoard en Linux, ChessBase y WinBoard en Windows. El motor de ajedrez suele ser el backend y funciona con una interfaz de línea de comandos sin gráficos ni ventanas por lo que requieren en la mayoría de las ocasiones de una interfaz de usuario.

La interfaz de línea de comandos de GNU Chess se convirtió en el primer estándar de facto y se llamó Protocolo de Comunicación de Motores de Ajedrez. A finales del año 2000 apareció un protocolo más nuevo, *UCI*¹. Algunos motores soportan ambos protocolos.

Los motores de ajedrez conforme pasa el tiempo van incrementando su fuerza. Esto se debe en parte al incremento de la capacidad de procesamiento que permite hacer cálculos más profundos en un tiempo dado. Además, las técnicas de programación van mejorando con el paso de los años permitiendo a los módulos ser más selectivos en las líneas que analizan y conseguir así entender mejor las diferentes posiciones que pueden darse en una partida.

Hay cientos de motores de ajedrez disponibles de manera gratuita, muchos de ellos son de código abierto y los podemos encontrar en GitHub. Hay motores pedagógicos que sirven para enseñar el arte de la programación de ajedrez y hay otros que se distribuyen de manera comercial como Fritz o Rybka. Vamos a analizar las principales características de cuatro motores de código abierto: Stockfish, Leela Chess Zero, Ethereal y Nemorino para los cuales hemos visitado sus repositorios de GitHub.

¹Universal Chess Interface, es un protocolo de comunicación abierto para que los motores de ajedrez jueguen partidas automáticamente, es decir, que se comuniquen con otros programas incluyendo las interfaces gráficas de usuario.

6. Motores de Ajedrez

6.1.1. Stockfish

Stockfish [13] es un motor de ajedrez de código abierto derivado de Glaurung 2.1. Fue desarrollado por Tod Romstad, Marco Costalba, Joonas Kiiski y Gary Linscott con licencia GPLv3.0. Stockfish no es un programa de ajedrez completo y requiere una interfaz gráfica de usuario compatible con UCI para poder ser utilizado cómodamente. Stockfish está escrito en C++, puede compilarse y construirse para varios procesadores y sistemas operativos. El nombre de Stockfish refleja la ascendencia del motor, puesto que Tod es noruego y Marco italiano y hay una larga historia de comercio de stockfish entre ambos países. Actualmente, el proyecto está siendo desarrollado y mantenido por la comunidad de Stockfish y la versión más reciente es Stockfish 15.

Para la representación del tablero, Stockfish empleaba listas de piezas 5.2.2.1 hasta Stockfish 12, ahora la parte que no emplea NNUE ² utiliza listas cuadradas 8x8 y además, bitboards y magic bitboards. Para llevar a cabo la búsqueda y encontrar movimientos candidatos utiliza multiples técnicas, entre otras: profundidad iterativa con ventanas de aspiración, búsqueda paralela perezosa, búsqueda de variaciones principales, tablas de transposiciones y tablas hash compartidas. Además, también hace uso de la búsqueda de quietud, ordenación de jugadas para emplear heurística asesina, history heuristic, etc; y también utiliza la poda de jugadas nulas y jugadas asesinas entre otras técnicas. También soporta bases Syzygy ³.

Stockfish NNUE es una rama de Stockfish creada por Hisayori Noda que utiliza redes neuronales actualizables eficientemente para reemplazar su evaluación estándar. Las NNUE introducidas en 2018 por Yu Nasu en un documento llamado *Efficiently updatable neural network - NNUE* [65] se aplicaron previamente con éxito en las funciones de evaluación del Shogi. En 2019 se llegó a obtener una prometedora rama de Stockfish NNUE cuando se introdujo NNUE en Stockfish, la nueva clase de funciones de evaluación no lineales basadas en redes neuronales. En agosto de 2020 Stockfish NNUE era más fuerte que el clásico y el día 2 de septiembre de 2020 apareció Stockfish 12 que a pesar de que la velocidad de búsqueda se redujo, incrementó de forma notable su fuerza gracias a NNUE y a una mayor afinación del motor, tenía mejor ajuste. Posteriormente, se lanzaron Stockfish 13 y Stockfish 14 que mejoró de manera notable gracias a Tomasz Sobczyk y Gary Linscott al diseñar una nueva arquitectura NNUE junto con un entrenador acelerado por la GPU escrito en PyTorch. Además, también se produjo una colaboración con Leela Chess Zero 6.1.2 que proporcionó miles de millones de posiciones para entrenar la nueva NNUE. El pasado 18 de abril de 2022 el equipo de desarrolladores sacó la última versión del motor, Stockfish 15.

El motor de Stockfish cuenta con dos funciones de evaluación para el ajedrez, una es la evaluación basada en una red neuronal actualizable (NNUE) que es la predeterminada y mucho más fuerte y la otra es la evaluación clásica basada en términos artesanales. Se ejecutan en la CPU de manera eficiente y NNUE utiliza varias técnicas de aceleración como el cálculo incremental. Además cuenta desde 2017 con una guía de evaluación interactiva a la cual le podemos introducir una cadena FEN de una posición para obtener la puntuación resultante del término de evaluación principal considerando las fases del juego dentro de su evaluación cónica ⁴ y se puede navegar por el árbol de búsqueda para encontrar las mejores

²Red neuronal actualizable eficientemente.

³Las bases Syzygy son bases de datos compactas de finales de seis piezas desarrolladas por Ronald de Man. También existen las bases de datos de siete piezas.

⁴La evaluación cónica es una técnica que se emplea en la evaluación para hacer una transición suave entre las fases de la partida. La idea consiste en eliminar la discontinuidad de la evaluación.

jugadas.

Ahora vamos a ver unos pequeños ápices del funcionamiento interno de la parte neuronal y posteriormente veremos los métodos de evaluación clásicos.

La función de evaluación neuronal se basa en la arquitectura NNUE de Yu Nasu. Esta red neuronal evalúa una posición de juego en una CPU sin necesidad de un procesamiento gráfico. La red neuronal está formada por cuatro capas, las primeras tres capas se encuentran totalmente conectadas. La primera capa está fuertemente sobreparametrizada y recibe una posición de ajedrez codificada como una estructura *HalfKP*⁵, la segunda y la tercera son capas ocultas y la última capa, la cuarta, es la capa de salida que puede expresarse como puntuación de material obtenida gracias a la evaluación.

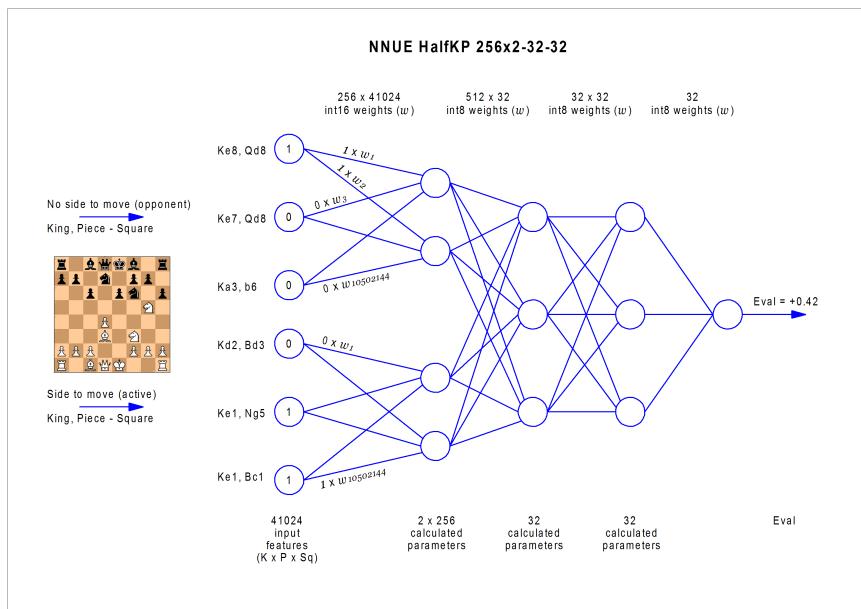


Figura 6.1.: Representación de una NNUE. [71]

NNUE no entiende los bitboards y la posición se codifica mediante codificación binaria que tiene varias ventajas:

- La actualización de las entradas y las neuronas subsiguientes después de un cambio de posición es más rápida porque esta codificación permite el cálculo incremental, siendo este el punto fuerte de NNUE.
- El cambio de turno de jugador también es más rápido.
- Esta codificación permite introducir más conocimientos en la red para disminuir su coste de entrenamiento, ya que emplea sobreespecialización, proporciona más información de la necesaria en la codificación.

El objetivo de la red es construir una función de evaluación que sea una combinación de las neuronas de las diferentes capas separadas por no linealidades. Esta función de evaluación será optimizada en la fase de entrenamiento con el fin de emitir la evaluación más precisa

⁵HalfKP es una combinación del rey y las piezas propias.

6. Motores de Ajedrez

possible de cada posición. La idea básica para el entrenamiento es construir un enorme conjunto de datos de entrada de posiciones generadas aleatoriamente que se evalúan a baja profundidad con el clásico Stockfish. Así, en vez de entrenar el motor completamente de cero, NNUE se entrena con posiciones que Stockfish 11 ya ha analizado. La posición, la profundidad de búsqueda y la evaluación se introducen en el modelo de NNUE.

Stockfish sigue basándose en la evaluación clásica. La función de evaluación clásica se basa en una serie de conceptos y criterios que se ponderan y se suman. Esta función puede ser escalada para expresar una ventaja en centipeones. Algunos de los criterios asociados a esta función de evaluación son: la evaluación cónica, el balance material, control del tablero, movilidad de las piezas, espacio, estructura de peones, evaluación de patrones como puestos de avanzada, amenazas que puedan existir, contar con la pareja de alfiles, etc.

6.1.2. Leela Chess Zero

Leela Chess Zero, conocido también como LCZero (Lc0) es una adaptación del proyecto Leela Zero Go de Gian-Carlo Pascutto al ajedrez. El programador Gary Linscott quien también es desarrollador del motor de ajedrez Stockfish fue quien encabezó el desarrollo del proyecto que se anunció por primera vez el 9 de enero de 2018. Leela Chess Zero es un motor de código abierto, se encuentra bajo los términos de la versión 3 de GPL o posterior y soporta UCI. Leela Chess Zero está escrito en C++ y puede ser compilado por varias plataformas y backends, siendo las plataformas preferidas las GPUs de Nvidia que soportan las librerías CUDA y CuDNN.

Tiene como objetivo construir una fuerte entidad del juego de ajedrez siguiendo el mismo tipo de aprendizaje profundo junto con las técnicas de búsqueda en árboles de Monte Carlo (MCTS) de AlphaZero pero empleando un entrenamiento distribuido de los pesos de la red neuronal convolucional profunda (CNN, DNN, DCNN).

Leela Chess Zero está formado por un ejecutable que juega o analiza partidas, inicialmente apodado LCZero, posteriormente se reescribió y mejoró su rendimiento pasando a ser Lc0. Este ejecutable es el verdadero motor de ajedrez, realiza un búsqueda en árboles de Monte Carlo (MCTS) [5.3.8.1](#) y lee la CNN de manera autodidacta, cuyos pesos se encuentran en un archivo separado de manera persistente.

Para representar el tablero, Lc0 emplea cinco bitboards, un tablero para las piezas propias y otro para las del adversario, otro para los peones, y emplea otros dos, uno para las piezas deslizantes ortogonales y otro para las piezas deslizantes diagonales. Incluyen la información de si se puede realizar una captura al paso, si hay derecho a enroque y una bandera que indica si se ha invertido el color del tablero.

Lc0 evalúa las posiciones utilizando una aproximación de función no lineal basada en una red neuronal profunda, en vez de la aproximación de función lineal que se utiliza en la mayoría de programas de ajedrez clásico. Esta red neuronal toma la posición del tablero como entrada y emite la evaluación de la posición y un vector de probabilidades de movimiento. Una vez ha sido entrenada se combina con la MCTS utilizando la política para limitar la búsqueda a las jugadas de alta probabilidad y utilizando el valor junto con una política de despliegue rápido para evaluar las posiciones del árbol. La selección MCTS se realiza mediante una variación del mejor UCT (explicado en el capítulo anterior) denominada PUCT. Con respecto a los nodos por segundo de la MCTS, los modelos más pequeños son más rápidos de calcular que

los modelos más grandes pero también se saturan antes, de modo que en algún momento aunque entrenemos más estos nodos no mejoraremos el motor. Los modelos de red más grandes y profundos mejorarán la receptividad, la cantidad de conocimientos y los patrones a extraer de las muestras de entrenamiento.

6.1.3. Nemorino

El motor de ajedrez Nemorino debe su nombre al personaje de la ópera cómica de Gaetano Donizetti, *L'elisir d'amore* estrenada el 12 de mayo de 1832 en Milán. Es un motor de código abierto creado por Christian Günther, compatible con UCI y también soporta el protocolo de comunicación del motor de ajedrez. No trae su propia interfaz de usuario, por lo que para ejecutarlo se necesita una interfaz gráfica de usuario que soporte UCI. El autor escribió este motor debido a que quería aprender cómo funciona un motor de ajedrez, fue escrito en sus inicios en C# y posteriormente comenzó a escribirlo en C++, fue publicado por primera vez en 2016 bajo la licencia GPLv3.

La representación del tablero y la estructura de los movimientos de Nemorino están tomadas de Stockfish. Para la representación del tablero además de bitboards y magic bitboards para la generación de movimientos deslizantes emplea Copy/Make que mantiene y actualiza copias de ciertos aspectos de una posición de ajedrez de una matriz indexada por capas. En general se refiere a los aspectos irreversibles de una posición como los derechos de enroque. También soporta bases Syzygy y tablas hash compartidas. Otros sistemas que emplea para la búsqueda de posiciones son: búsqueda paralela muy perezosa, tablas de transposiciones, búsqueda de variaciones principales y profundidad iterativa con ventanas de aspiración, poda de jugadas nulas, jugadas asesinas, códigos hash de Zobrist y búsqueda de quietud, entre otras.

Para llevar a cabo la evaluación de las diferentes posiciones emplea una evaluación cónica basada en el material, la movilidad, el control del tablero, las amenazas, la seguridad del rey y la estructura de peones, así como funciones especiales para la evaluación de algunos finales. Además tiene soporte de evaluación basada en redes NNUE.

La última versión de Nemorino es la 6 y se puede encontrar en GitHub cuya principal novedad es la compatibilidad con las redes NNUE, las cuales permiten que mejore mucho su fuerza de juego. Viene con su propia red siendo ligeramente diferente al formato de Stockfish. Sin embargo, Nemorino puede utilizar los archivos de red de Stockfish también.

6.1.4. Ethereal

Ethereal es un motor de ajedrez de código abierto compatible con UCI escrito por Andrew Grant en C bajo la licencia pública general de GNU y publicado por primera vez oficialmente en 2016. Ethereal se inspira en una serie de proyectos de código abierto como Stockfish, Crafty, TSCP, MadChess y Fruit. Pretende servir tanto de motor de gama alta como de referencia para otros autores. Andrew Grant lanzó Ethereal 13.00 NNUE, una implementación de NNUE basada en Stockfish NNUE, la versión estándar se encuentra actualmente en GitHub. El programa viene con dos NNUEs para la evaluación, una para el ajedrez estándar y otra para el ajedrez 960⁶.

⁶Variante del ajedrez propuesta por Boby Fischer en 1996.

6. Motores de Ajedrez

Ethereal dispone de una serie de características que lo diferencian de otros motores, tanto en la codificación del tablero, como en la generación de jugadas y en la función de evaluación. Para la representación del tablero emplea bitboards, fancy magic bitboards además de una lista cuadrada 8x8. El motor también es conocido por su potente libro de aperturas que viene con más de cuatro millones de posiciones, lo que le da cierta ventaja, en las primeras etapas del juego. Algunas técnicas de búsqueda de posiciones que emplea son: búsqueda paralela perezosa, profundidad iterativa con ventanas de aspiración, tablas de transposiciones, búsqueda de variaciones principales, poda de jugadas nulas, jugadas asesinas, búsqueda de quietud, etc. Además posee una gran fuerza bruta.

La evaluación de las diferentes posiciones de ajedrez se lleva a cabo mediante funciones de evaluación basadas en redes neuronales (NNUE), también emplea una evaluación cónica basada en el balance material, la movilidad, el control del tablero, las amenazas, la seguridad del rey y la estructura de peones haciendo uso de una tabla hash de peones. A diferencia de Nemorino, Ethereal le da más importancia a algunos detalles como que la torre se encuentre en la séptima fila, o que esté en columnas abiertas y semiabiertas, así como a ciertos patrones de la posición como los puestos de avanzada.

Además, es capaz de manejar múltiples variantes y puede utilizarse con una variedad de interfaces gráficas de ajedrez. El motor dispone de una interfaz intuitiva que permite que el análisis de ajedrez sea sencillo y directo, también dispone de estadísticas de rendimiento. Posee aspectos negativos, algunos usuarios han reportado problemas de estabilidad con el motor y la documentación es limitada.

Actualmente, la última versión que ha sido lanzada es Ethereal 13.75 (NNUE). En esta versión se han realizado gran cantidad de cambios para mejorar la búsqueda, la gestión del tiempo, la arquitectura y los métodos de entrenamiento. Ethereal ha logrado un par de redes que son más grandes y más rápidas que las anteriores. Estas no están entrenadas ni duplicadas a partir de trabajos de ningún otro equipo de ajedrez computacional y han sido entrenadas utilizando la suma total de los datos de entrenamiento acumulados desde la versión 13.00.

Parte III.

Fundamentos Teóricos

El propósito de esta parte consiste en presentar los fundamentos teóricos que se han utilizado para la elaboración de este proyecto. En los dos primeros capítulos se explicarán dos teoremas de gran relevancia, el teorema de No Free Lunch de optimización y el teorema de aproximación universal de redes neuronales. En el tercer capítulo se profundizará en los conceptos relacionados con la teoría del aprendizaje y se tratará el problema de la generalización. En el cuarto capítulo se introducirá la parte del aprendizaje automático que se ha utilizado para desarrollar los modelos, el aprendizaje profundo, donde se definirán formalmente las redes neuronales, su proceso de entrenamiento y algunas de sus características. En el último capítulo de este bloque se estudiarán las redes neuronales convolucionales.

7. Teorema de No Free Lunch

En este capítulo se va a explicar el teorema de No Free Lunch (NFL). Este teorema tiene muchas variantes que proporcionan resultados ligeramente diferentes y formaliza una idea importante en el aprendizaje estadístico. La más famosa dice, a grandes rasgos, que todos los algoritmos de aprendizaje tienen el mismo rendimiento cuando se promedian entre todas las funciones objetivo posibles. En otras palabras, no existe un algoritmo de aprendizaje universal que supere a todos los demás algoritmos en todas las tareas de aprendizaje.

En este caso, el teorema de NFL que se describe es el que se recoge en el libro *Understanding Machine Learning* [75] que dice, en líneas generales, que para cada algoritmo de aprendizaje se puede encontrar una distribución que puede hacer que el algoritmo de aprendizaje falle.

En la primera parte del capítulo se presentan los conceptos básicos sobre teoría de la medida y los lemas necesarios para demostrar el teorema. En la segunda parte se introducen una serie de conceptos sobre aprendizaje, necesarios para comprender el teorema. En la última parte de este capítulo se presenta el enunciado y la demostración del teorema de No Free Lunch y sus principales consecuencias.

7.1. Preliminares

7.1.1. Conceptos básicos sobre Teoría de la Medida

En este capítulo se van a introducir unas nociones sobre teoría de la medida necesarias para la comprensión de esta parte del trabajo. Los conceptos que se presentan se pueden encontrar en multitud de libros, el libro que se ha escogido en este caso ha sido *Probability Theory* [52].

Considere un conjunto no vacío Ω y sea $\mathcal{P}(\Omega)$ el conjunto potencia de Ω (conjunto de todos los subconjuntos de Ω). Se define una σ -álgebra sobre el conjunto Ω como sigue:

Definición 7.1 (σ -álgebra). Una clase de conjuntos $\mathcal{A} \subset \mathcal{P}(\Omega)$ es una σ -álgebra si satisface las siguientes condiciones:

- $\Omega \in \mathcal{A}$
- \mathcal{A} es cerrado bajo complementos, es decir, si $A \in \mathcal{A}$, entonces $A^c := \Omega \setminus A \in \mathcal{A}$.
- \mathcal{A} es cerrado bajo uniones numerables, si $A_n \in \mathcal{A}$ para todo $n \in \mathbb{N}$ y $B = \cup_{n \in \mathbb{N}} A_n$, entonces $B \in \mathcal{A}$.

Nótese que, si \mathcal{A} es una σ -álgebra, entonces $\emptyset = \Omega \setminus \Omega$ está en \mathcal{A} .

Los conjuntos $A \in \mathcal{A}$ se llaman conjuntos medibles y la tupla (Ω, \mathcal{A}) donde Ω es un conjunto no vacío y $\mathcal{A} \subset \mathcal{P}(\Omega)$ es una σ -álgebra que se conoce como espacio medible. Veamos algunos ejemplos.

7. Teorema de No Free Lunch

Ejemplo 7.1.

1. El conjunto de las partes de Ω , $\mathcal{A} = \mathcal{P}(\Omega)$.
2. $\mathcal{A} = \{\emptyset, \Omega\}$.
3. $\mathcal{A} = \{\emptyset, \Omega, A, A^c\}$, donde $A \subset \Omega$.
4. $\mathcal{A} = \{A \in \mathbb{R} : A \text{ numerable}\} \cup \{A \in \mathbb{R} : A^c \text{ numerable}\}$.

La intersección de σ -álgebras es una σ -álgebra. En general, la unión de σ -álgebras no es una σ -álgebra.

Teorema 7.1 (σ -álgebra generada). *Sea Ω un conjunto no vacío arbitrario y $\mathcal{C} \subset \mathcal{P}(\Omega)$ una familia de subconjuntos de Ω . Se define la σ -álgebra generada por \mathcal{C} como la intersección de todas las σ -álgebras que contienen a \mathcal{C} (la mínima σ -álgebra que contiene a \mathcal{C}). Su expresión viene dada por*

$$\sigma(\mathcal{C}) := \bigcap_{\substack{\mathcal{A} \subset \mathcal{P}(\Omega) \text{ es } \sigma-\text{álgebra} \\ \mathcal{C} \subset \mathcal{A}}} \mathcal{A}.$$

Ahora se muestra un ejemplo muy famoso de σ -álgebra conocido como la σ -álgebra de Borel que es la mínima σ -álgebra que contiene una topología dada.

Definición 7.2 (σ -álgebra de Borel). *Sea (Ω, τ) un espacio topológico. La σ -álgebra de Borel $\mathcal{B} := \mathcal{B}(\Omega)$ es la generada por los conjuntos abiertos de Ω .*

Los elementos de $\mathcal{B}(\Omega)$ son los llamados conjuntos de Borel o conjuntos Borel medibles. Normalmente la topología que se usa es la topología usual de \mathbb{R}^n .

Definición 7.3 (Medida). *Sea (Ω, \mathcal{A}) un espacio medible. Una medida sobre (Ω, \mathcal{A}) es una aplicación $\mu : \mathcal{A} \rightarrow \mathbb{R}_0^+$ que cumple las siguientes condiciones:*

1. $\mu(\emptyset) = 0$.
2. Si $\{A_n\}_{n \in \mathbb{N}} \subset \mathcal{A}$ es una colección numerable de subconjuntos tales que $A_i \cap A_j = \emptyset$ si $i \neq j$, entonces

$$\mu \left(\bigcup_{n=1}^{\infty} A_n \right) = \sum_{n=1}^{\infty} \mu(A_n).$$

En el siguiente ejemplo se presentan dos de las medidas que se van a utilizar a lo largo del presente documento.

Ejemplo 7.2.

1. Se define la función indicadora sobre el conjunto $A \subset \Omega$ por

$$\mathbb{1}_A(x) := \begin{cases} 1 & \text{si } \omega \in A \\ 0 & \text{si } \omega \notin A \end{cases}$$

Sea $\omega \in \Omega$ y $\delta_\omega(A) = \mathbb{1}_A(\omega)$. Entonces δ_ω es una medida de probabilidad sobre cualquier σ -álgebra $\mathcal{A} \subset \mathcal{P}(\Omega)$, conocida como *medida de Dirac* para el punto ω . Esta medida se empleará para demostrar el teorema de No Free Lunch.

2. Existe una medida únicamente determinada definida en $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$ denotada por λ^n . Se caracteriza por la siguiente propiedad:

$$\lambda^n((a, b]) = \prod_{i=1}^n (b_i - a_i), \forall a, b \in \mathbb{R}^n \text{ con } a < b.$$

Esta medida se conoce como *medida de Lebesgue-Borel*.

La tripleta $(\Omega, \mathcal{A}, \mu)$ recibe el nombre de espacio de medida, esto es, un espacio medible (Ω, \mathcal{A}) sobre el que se define una medida μ . En el caso de que μ sea una medida de probabilidad esa tripleta se conoce como espacio de probabilidad y los conjuntos $A \in \mathcal{A}$ se llaman sucesos.

Definición 7.4 (Espacio de Probabilidad). Un espacio de probabilidad es una terna (Ω, \mathcal{A}, P) donde Ω es un conjunto no vacío llamado espacio muestral que contiene a todos los posibles resultados del experimento. \mathcal{A} es una σ -álgebra de subconjuntos de Ω y P es una medida de probabilidad sobre \mathcal{A} , es decir, $P : \mathcal{A} \rightarrow [0, 1]$, que asigna una probabilidad a todo suceso de manera que se verifican los tres axiomas de Kolmogorov

1. $P(\Omega) = 1$.
2. $P(A) \geq 0 \quad \forall A \in \mathcal{A}$
3. Si $\{A_n\}_{n \in \mathbb{N}}$ son sucesos de \mathcal{A} tales que $A_i \cap A_j = \emptyset$ si $i \neq j$ entonces

$$P\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} P(A_n)$$

Por último, se presenta el concepto de aplicación medible el cual no se debe confundir con el de medida.

Definición 7.5 (Aplicación medible). Sean (Ω, \mathcal{A}) y (Ω', \mathcal{A}') dos espacios medibles. Una aplicación $X : \Omega \rightarrow \Omega'$ se dice que es medible cuando la imagen inversa por X de todo elemento de \mathcal{A}' es un elemento de \mathcal{A} , esto es, $X^{-1}(A) \in \mathcal{A}$ para todo $A \in \mathcal{A}'$. Cuando X sea medible, la denotaremos como $X : (\Omega, \mathcal{A}) \rightarrow (\Omega', \mathcal{A}')$.

A continuación, se presentan algunos ejemplos de aplicaciones medibles.

Ejemplo 7.3.

1. La identidad es una aplicación medible.
2. La composición de aplicaciones medibles es medible.
3. La función indicadora es una aplicación medible.
4. Cualquier aplicación $X : (\Omega, \mathcal{P}(\Omega)) \rightarrow (\Omega', \{\emptyset, \Omega'\})$, donde $\mathcal{P}(\Omega)$ representa las partes de Ω es medible.

7.1.2. Resultados previos

En esta sección se van a exponer algunos resultados necesarios para demostrar el teorema de aproximación universal.

En teoría de la probabilidad la desigualdad de Markov aumenta la probabilidad de que una variable aleatoria real con valores positivos sea mayor o igual que una constante positiva. La desigualdad debe su nombre a Andrei Markov.

Lema 7.1 (Desigualdad de Markov). *Sea X una variable aleatoria real que solo toma valores no negativos. Entonces, para cualquier número real positivo a se tiene*

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}[X]}{a},$$

siempre que existe $\mathbb{E}[X]$.

Demostración. Realizaremos la demostración tanto para el caso discreto como para el caso continuo.

- Si X es discreta, tenemos por definición, $\mathbb{E}[X] = \sum_x x\mathbb{P}[X = x]$. Dividiremos esta suma en dos partes, dependiendo de si $x \geq a$

$$\mathbb{E}[X] = \sum_{x \geq a} x\mathbb{P}[X = x] + \sum_{x < a} x\mathbb{P}[X = x].$$

Por una parte, sabemos que

$$\sum_{x < a} x\mathbb{P}[X = x] \geq 0$$

y por otra al suponer que $x \geq a$ llegamos a

$$\sum_{x \geq a} x\mathbb{P}[X = x] \geq \sum_{x \geq a} a\mathbb{P}[X = x],$$

por tanto,

$$\begin{aligned} \mathbb{E}[X] &= \sum_{x \geq a} x\mathbb{P}[X = x] + \sum_{x < a} x\mathbb{P}[X = x] \\ &\geq \sum_{x \geq a} a\mathbb{P}[X = x] \\ &= a \sum_{x \geq a} \mathbb{P}[X = x] \\ &= a\mathbb{P}[X \geq a] \end{aligned}$$

y el resultado se obtiene despejando $\mathbb{P}[X \geq a]$.

- Si X es continua, su función de densidad es f_X y como X es no negativa tenemos

$$\begin{aligned}\mathbb{E}[X] &= \int_{-\infty}^{\infty} xf_X(x)dx \\ &= \int_{-\infty}^0 xf_X(x)dx + \int_0^{\infty} xf_X(x)dx \\ &= 0 + \int_0^{\infty} xf_X(x)dx.\end{aligned}$$

Usando ahora que $a > 0$

$$\begin{aligned}\mathbb{E}[X] &= \int_0^{\infty} xf_X(x)dx \\ &= \int_0^a xf_X(x)dx + \int_a^{\infty} xf_X(x)dx \\ &\geq a \int_a^{\infty} f_X(x)dx \\ &= a\mathbb{P}[X \geq a],\end{aligned}$$

de donde se deduce la desigualdad buscada.

□

Lema 7.2. *Sea X una variable aleatoria que toma valores en $[0, 1]$ tal que $\mathbb{E}[X] = \mu$. Entonces, para cada $a \in (0, 1)$ tenemos*

$$\mathbb{P}[X > 1 - a] \geq \frac{\mu - (1 - a)}{a} \quad y \quad \mathbb{P}[X > a] \geq \frac{\mu - a}{1 - a} \geq \mu - a.$$

Demostración. Sea $Y = 1 - X$ una variable aleatoria no negativa con $\mathbb{E}[Y] = 1 - \mathbb{E}[X] = 1 - \mu$. Por la desigualdad de Markov:

$$\mathbb{P}[X \leq 1 - a] = \mathbb{P}[1 - X \geq a] = \mathbb{P}[Y \geq a] \leq \frac{\mathbb{E}[Y]}{a} = \frac{1 - \mu}{a}.$$

Por tanto,

$$\mathbb{P}[X > 1 - a] \geq 1 - \frac{1 - \mu}{a} = \frac{a + \mu - 1}{a} = \frac{\mu - (1 - a)}{a}$$

Sustituyendo a por $1 - a$ tenemos:

$$\mathbb{P}[X > a] \geq \frac{\mu - a}{1 - a} \geq \mu - a.$$

□

Gracias a la desigualdad de Markov y al lema 7.2 podemos enunciar y demostrar el lema que necesitamos para probar el teorema de aproximación universal.

7. Teorema de No Free Lunch

Lema 7.3. Sea θ una variable aleatoria con rango en el intervalo $[0, 1]$ tal que $\mathbb{E}[\theta] \geq \frac{1}{4}$. Tenemos

$$\mathbb{P}\left[\theta > \frac{1}{8}\right] \geq \frac{1}{7}.$$

Demostración. A partir de la segunda desigualdad del lema previo se deduce que

$$\mathbb{P}[\theta > a] \geq \frac{\mathbb{E}[\theta] - a}{1 - a}.$$

Sustituyendo a por $\frac{1}{8}$ obtenemos:

$$\mathbb{P}\left[\theta > \frac{1}{8}\right] \geq \frac{\frac{1}{4} - \frac{1}{8}}{1 - \frac{1}{8}} = \frac{1}{7}.$$

□

7.2. Conceptos sobre Aprendizaje

En esta parte vamos a describir una serie de conceptos previos sobre aprendizaje necesarios para comprender el teorema de No Free Lunch, estos se pueden encontrar en [75]. En primer lugar, definiremos el concepto de algoritmo de aprendizaje. Un algoritmo de aprendizaje es un algoritmo al cual se le pasa una entrada y genera una salida, lo denotaremos por A . El término *aprendizaje* se debe a que el proceso interno que realiza el algoritmo para proporcionar la salida se conoce como aprendizaje. Este concepto lo veremos en más detalle en el capítulo 9. Presentamos ahora los componentes de un modelo formal de aprendizaje:

- *Elementos de entrada:*
 - *Espacio muestral:* conjunto arbitrario \mathcal{X} que contiene a todas las posibles entradas. Es el conjunto de objetos que pretendemos etiquetar, generados de acuerdo a una distribución de probabilidad D desconocida. También nos referiremos a los puntos del dominio como instancias o ejemplos y a \mathcal{X} como espacio de instancias.
 - *Espacio de etiquetas:* es el conjunto de etiquetas posibles, será denotado por \mathcal{Y} .
 - *Datos de entrenamiento:* es un conjunto finito de pares en el dominio $\mathcal{X} \times \mathcal{Y}$, es decir, una secuencia de puntos de dominio etiquetados. Es denotado por $S = ((x_1, y_1), \dots, (x_m, y_m))$ y son los elementos a los que tiene acceso el algoritmo de aprendizaje A , los que utiliza para aprender. A veces también nos referiremos a S como *conjunto de entrenamiento*¹.
- *Elementos de salida:* el algoritmo de aprendizaje proporciona una salida, la cual será una aplicación $h : \mathcal{X} \rightarrow \mathcal{Y}$. Esta función recibe el nombre de *predictor*, *hipótesis* o *clasificador* y será escogida en un conjunto de posibles hipótesis \mathcal{H} . El predictor puede utilizarse para predecir la etiqueta de los nuevos puntos de dominio o ejemplos. Utilizaremos la

¹A pesar de la notación conjunto, S es una secuencia. En particular, el mismo ejemplo puede aparecer dos veces en S y algunos algoritmos pueden tener en cuenta el orden de los ejemplos de S .

notación $A(S)$ para denotar la hipótesis que un algoritmo de aprendizaje, A , devuelve al recibir los datos de entrenamiento S .

- *Generación de datos de entrenamiento:* tomamos una muestra finita de \mathcal{X} de manera que sea independiente e idénticamente distribuida (i.i.d.) generada por una distribución conjunta. Denotemos esta distribución de probabilidad sobre \mathcal{X} por \mathcal{D} . La hipótesis (i.i.d.) la denotaremos por $S \sim \mathcal{D}^m$ donde m es el tamaño de S y \mathcal{D}^m denota la probabilidad inducida sobre la m -tupla al aplicar \mathcal{D} para escoger cada elemento de la tupla de manera independiente del resto de elementos.
- *Función objetivo/Distribución objetivo:* la función objetivo es una aplicación denotada por $f : \mathcal{X} \rightarrow \mathcal{Y}$ que a cada $x \in \mathcal{X}$ le hace corresponder una etiqueta $y \in \mathcal{Y}$. Esta función es desconocida y es la que pretendemos aprender, tan solo conocemos de ella su imagen sobre \mathcal{X} . Vamos a definir $Z' = \{(x, f(x)) \mid x \in \mathcal{X}\}$ como el conjunto formado por los ejemplos con sus respectivas etiquetas, cuyo dominio es $Z = \mathcal{X} \times \mathcal{Y}$. En la práctica, existe ruido en el proceso de generación de las etiquetas, por tanto, la salida no está totalmente determinada por la entrada. Luego, debemos considerar la salida y como una variable aleatoria que se ha visto afectada por la entrada x . Esto es, tenemos una distribución de probabilidad (distribución objetivo) $D(y|x)$ en vez de una función objetivo $y = f(x)$. Por tanto, partiendo de una distribución objetivo, podemos considerar $D(y|x) = 0$ para cualquier y excepto cuando $y = f(x)$, obteniendo así la función objetivo.
- *Medidas de éxito:* definiremos el error de un clasificador como la probabilidad de que no prediga la etiqueta correcta en un punto de datos aleatorio generado por la mencionada distribución subyacente. Es decir, el error de h es la probabilidad de extraer una instancia aleatoria x , según la distribución \mathcal{D} , tal que $h(x) \neq f(x)$.

Formalmente, dado un subconjunto del espacio muestral, $A \subset \mathcal{X}$, la distribución de probabilidad, \mathcal{D} , asigna un número, $\mathcal{D}(A)$, que determina la probabilidad de observar un punto $x \in A$. En muchos casos nos referiremos a A como un suceso y lo expresaremos mediante una función $\pi : \mathcal{X} \rightarrow \{0, 1\}$, es decir, $A = \{x \in \mathcal{X} : \pi(x) = 1\}$. En ese caso, también utilizaremos la notación $\mathbb{P}_{x \sim \mathcal{D}}[\pi(x)]$ para expresar $\mathcal{D}(A)$. Definimos el error de una regla de predicción, $h : \mathcal{X} \rightarrow \mathcal{Y}$, como

$$L_{\mathcal{D},f}(h) := \mathbb{P}_{x \sim \mathcal{D}}[h(x) \neq f(x)] := \mathcal{D}(\{x : h(x) \neq f(x)\}).$$

Esto es, el error de la hipótesis h es la probabilidad de elegir al azar un ejemplo x para el que $h(x) \neq f(x)$. El subíndice (\mathcal{D}, f) indica que el error se mide con respecto a la distribución de probabilidad \mathcal{D} y a la función de etiquetado correcta f aunque en lo que sigue lo omitiremos porque quedará claro por el contexto.

El espacio de probabilidad está definido por $(\Omega, \mathcal{A}, \mathbb{P})$, donde normalmente $\Omega = \mathbb{R}^n$, $\mathcal{A} = \mathcal{B}(\mathbb{R}^n)$ y \mathbb{P} es cualquier medida de probabilidad. Sea $X : (\Omega, \mathcal{A}) \rightarrow (Z, \mathcal{B}(Z))$ una variable aleatoria definida en este espacio de probabilidad, donde $\mathcal{B}(Z)$ es la σ -álgebra de Borel generada por Z y su distribución de probabilidad $D = \mathbb{P} \circ X^{-1}$ tiene como dominio $(Z, \mathcal{B}(Z))$.

Vamos a generalizar el formalismo de la medida del éxito como sigue:

7.2.1. Funciones de pérdida generalizadas

Formalmente, dado cualquier conjunto de hipótesis \mathcal{H} y algún dominio Z , definimos el error como una aplicación $\ell : \mathcal{H} \times Z \rightarrow \mathbb{R}^+$ que toma valores en el conjunto de números reales no negativos. Llamamos a estas funciones, *funciones de pérdida*. Observemos que para los problemas de predicción tenemos que $Z = \mathcal{X} \times \mathcal{Y}$. Sin embargo, la noción de función de pérdida se generaliza más allá de las tareas de predicción, y por tanto, permite que Z sea cualquier dominio de ejemplos.

Definición 7.6. Sea $h \in \mathcal{H}$ un clasificador, f una función objetivo. Definimos la función de error de generalización como la esperanza del error de hipótesis h , con respecto a una distribución de probabilidad \mathcal{D} sobre Z , a saber,

$$L_{\mathcal{D}}(h) := \mathbb{E}_{z \sim \mathcal{D}}[\ell(h, z)]$$

donde $z = (x, f(x))$ pertenece al conjunto de pares $Z \in \mathcal{X} \times \mathcal{Y}$.

Nos gustaría encontrar un predictor, h , que mimimice el error de generalización $L_{\mathcal{D}}(h)$. Sin embargo, el algoritmo de aprendizaje no conoce los datos que genera D . A lo que sí tiene acceso es a los datos de entrenamiento, S .

Definición 7.7. Sea $h \in \mathcal{H}$ un clasificador, f una función objetivo, $S = (z_1, \dots, z_m) \in Z^m$ una muestra i.i.d. dada de tamaño m donde cada z_i , $1 \leq i \leq m$, es un elemento del conjunto de pares $Z = \mathcal{X} \times \mathcal{Y}$. Definimos el error de entrenamiento como la pérdida esperada sobre una muestra dada S ,

$$L_S(h) := \frac{1}{m} \sum_{i=1}^m \ell(h, z_i).$$

Vamos a definir ahora una función de pérdida que se utiliza en problemas de clasificación binaria o multiclase y que será la que empleemos en el teorema de No Free Lunch.

Definición 7.8. Sea $h \in \mathcal{H}$ un clasificador, f una función objetivo, definimos para todo $z = (x, f(x)) \in Z$ la función de pérdida o error 0-1 como

$$\ell_{0-1}(h(x, y)) := \begin{cases} 0 & \text{si } h(x) = y \\ 1 & \text{si } h(x) \neq y \end{cases}$$

Hay que tener en cuenta que, para una variable aleatoria α que toma valores en $\{0, 1\}$, $\mathbb{E}_{\alpha \sim \mathcal{D}}[\alpha] = \mathcal{P}_{\alpha \sim \mathcal{D}}[\alpha = 1]$.

En el capítulo 9 volveremos a tratar con el concepto de aprendizaje en el cual profundizaremos y también explicaremos uno de los principales problemas al que nos enfrentamos, el problema de la generalización, y cómo se construye un algoritmo de aprendizaje.

7.3. Teorema de No Free Lunch

Hasta ahora hemos visto que una tarea específica de aprendizaje está definida por una distribución desconocida \mathcal{D} sobre $\mathcal{X} \times \mathcal{Y}$, donde el objetivo del algoritmo de aprendizaje es

encontrar un predictor $h : \mathcal{X} \rightarrow \mathcal{Y}$, cuyo error de generalización $L_{\mathcal{D}}(h)$, sea lo suficientemente pequeño. La cuestión que se plantea es, si existe un algoritmo de aprendizaje A y un conjunto de entrenamiento de tamaño m de forma que, para cada distribución \mathcal{D} , si A recibe m ejemplos i.i.d. de \mathcal{D} , hay una alta probabilidad de que produzca un predictor h que tenga un error bajo.

El teorema de No Free Lunch afirma que no existe un algoritmo de aprendizaje universal. De forma más precisa, el teorema afirma que para las tareas de predicción de *clasificación binaria*², para cada algoritmo de aprendizaje existe una distribución en la que falla. Decimos que el algoritmo falla si, al recibir ejemplos i.i.d. de esa distribución, su hipótesis de salida es probable que tenga un error de generalización grande, digamos, mayor o igual que 0.3, mientras que para la misma distribución, existe otro algoritmo de aprendizaje que dará una hipótesis con un riesgo pequeño. En otras palabras, el teorema asegura que ningún algoritmo de aprendizaje puede tener éxito en todas las tareas que se pueden aprender: todos los algoritmos tienen tareas en las que fallan mientras que otros tienen éxito.

Teorema 7.2 (Teorema No Free Lunch). *Sea A cualquier algoritmo de aprendizaje para la tarea de clasificación binaria con respecto a la función de pérdida 0-1 (7.8) sobre un dominio \mathcal{X} . Sea m un número menor que $|\mathcal{X}|/2$, que representa el tamaño del conjunto de entrenamiento. Entonces, existe una distribución \mathcal{D} sobre $\mathcal{X} \times \{0,1\}$ tal que:*

1. Existe una función objetivo $f : \mathcal{X} \rightarrow \{0,1\}$ (para este caso binaria) con $L_{\mathcal{D}}(f) = 0$.
2. Con probabilidad de al menos $1/7$ sobre la elección $S \sim \mathcal{D}^m$ tenemos que $L_{\mathcal{D}}(A(S)) \geq 1/8$, es decir, $\mathbb{P}[L_{\mathcal{D}}(A(S)) \geq 1/8] \geq 1/7$.

Demostración. Sea C un subconjunto de X de tamaño $2m$. La *intuición* de la prueba es que cualquier algoritmo de aprendizaje que observe solo la mitad de las instancias de C no tiene información sobre cuáles deberían ser las etiquetas del resto de las instancias de C . Por tanto, existe una función objetivo f que contradiría las etiquetas que $A(S)$ predice sobre las instancias no observadas de C .

Obsérvese que hay $T = 2^{2m}$ funciones posibles de C a $\{0,1\}$. Denotamos estas funciones por f_1, \dots, f_T . Para cada una de estas funciones, sea \mathcal{D}_i una distribución sobre $C \times \{0,1\}$ definida por

$$\mathcal{D}_i(\{(x,y)\}) = \begin{cases} 1/|C| & \text{si } y = f_i(x) \\ 0 & \text{en otro caso.} \end{cases}$$

Esto es, la probabilidad de elegir un par (x,y) es $1/|C|$ si la etiqueta y es realmente la verdadera etiqueta según f_i y 0 en otro caso (si $y \neq f_i(x)$). Es evidente que $L_{\mathcal{D}_i}(f_i) = 0$.

Vamos a probar que para todo algoritmo A , que recibe un conjunto de entrenamiento de m ejemplos de $C \times \{0,1\}$ y devuelve una función $A(S) : C \rightarrow \{0,1\}$, se cumple que

$$\max_{1 \leq i \leq T} \mathbb{E}_{S \sim \mathcal{D}_i^m} [L_{\mathcal{D}_i}(A(S))] \geq 1/4. \quad (7.1)$$

Esto significa que para todo algoritmo, A' , que recibe un conjunto de entrenamiento de m ejemplos de $\mathcal{X} \times \{0,1\}$ existe una función $f : \mathcal{X} \rightarrow \{0,1\}$ y una distribución \mathcal{D} sobre

²La clasificación binaria es la tarea de clasificar los elementos de un conjunto en dos grupos en función de una regla de clasificación.

7. Teorema de No Free Lunch

$\mathcal{X} \times \{0,1\}$, tal que $L_{\mathcal{D}}(f) = 0$ y

$$\mathbb{E}_{S \sim \mathcal{D}^m}[L_{\mathcal{D}}(A'(S))] \geq 1/4. \quad (7.2)$$

Sabemos por el lema 7.3 que lo anterior es suficiente para demostrar que $\mathbb{P}[L_{\mathcal{D}}(f)(A'(S)) \geq 1/8] \geq 1/7$.

Pasamos ahora a demostrar que la ecuación (7.1) se cumple. Hay $k = (2m)^m$ posibles secuencias de m ejemplos de C . Denotemos estas secuencias por S_1, \dots, S_k . Además, si $S_j = (x_1, \dots, x_m)$ denotamos por S_j^i a la secuencia que contiene las instancias en S_j etiquetadas por la función f_i , es decir, $S_j^i = ((x_1, f_i(x_1)), \dots, (x_m, f_i(x_m)))$. Si la distribución es D_i , entonces los posibles conjuntos de entrenamiento que puede recibir A son S_1^i, \dots, S_k^i , y todos estos conjuntos de entrenamiento tienen la misma probabilidad de ser muestreados. Por lo tanto,

$$\mathbb{E}_{S \sim \mathcal{D}_i^m}[L_{\mathcal{D}_i}(A(S))] = \frac{1}{k} \sum_{j=1}^k L_{\mathcal{D}_i}(A(S_j^i)). \quad (7.3)$$

Utilizando los hechos de que el máximo es mayor que la media y que la media es mayor que el mínimo, tenemos

$$\begin{aligned} \max_{1 \leq i \leq T} \frac{1}{k} \sum_{j=1}^k L_{\mathcal{D}_i}(A(S_j^i)) &\geq \frac{1}{T} \sum_{i=1}^T \frac{1}{k} \sum_{j=1}^k L_{\mathcal{D}_i}(A(S_j^i)) \\ &= \frac{1}{k} \sum_{j=1}^k \frac{1}{T} \sum_{i=1}^T L_{\mathcal{D}_i}(A(S_j^i)) \\ &\geq \min_{1 \leq j \leq k} \frac{1}{T} \sum_{i=1}^T L_{\mathcal{D}_i}(A(S_j^i)). \end{aligned} \quad (7.4)$$

Fijemos ahora un j , $1 \leq j \leq k$. Denotemos $S_j = (x_1, \dots, x_m)$ y sea v_1, \dots, v_p los ejemplos de C que no aparecen en S_j . Claramente, $p \geq m$. Por tanto, para cada función $h : C \rightarrow \{0,1\}$ y cada i tenemos

$$\begin{aligned} L_{\mathcal{D}_i}(h) &= \frac{1}{2m} \sum_{x \in C} \mathbb{1}_{[h(x) \neq f_i(x)]} \\ &\geq \frac{1}{2m} \sum_{r=1}^p \mathbb{1}_{[h(v_r) \neq f_i(v_r)]} \\ &\geq \frac{1}{2p} \sum_{r=1}^p \mathbb{1}_{[h(v_r) \neq f_i(v_r)]}. \end{aligned} \quad (7.5)$$

Por tanto,

$$\begin{aligned}
 \frac{1}{T} \sum_{i=1}^T L_{\mathcal{D}_i}(A(S_j^i)) &\geq \frac{1}{T} \sum_{i=1}^T \frac{1}{2p} \sum_{r=1}^p \mathbb{1}_{[A(S_j^i)(v_r) \neq f_i(v_r)]} \\
 &= \frac{1}{2p} \sum_{r=1}^p \frac{1}{T} \sum_{i=1}^T \mathbb{1}_{[A(S_j^i)(v_r) \neq f_i(v_r)]} \\
 &\geq \frac{1}{2} \min_{1 \leq r \leq p} \frac{1}{T} \sum_{i=1}^T \mathbb{1}_{[A(S_j^i)(v_r) \neq f_i(v_r)]}.
 \end{aligned} \tag{7.6}$$

Fijamos algún r , $1 \leq r \leq p$. Podemos dividir todas las funciones f_1, \dots, f_T en $T/2$ conjuntos disjuntos, donde para un par $(f_i, f_{i'})$ tenemos que para todo $c \in C$, $f_i(c) \neq f_{i'}(c)$ si y solo si $c = v_r$. Como para tal par debemos tener $S_j^i = S_j^{i'}$, se deduce que

$$\mathbb{1}_{[A(S_j^i)(v_r) \neq f_i(v_r)]} + \mathbb{1}_{[A(S_j^{i'})(v_r) \neq f_{i'}(v_r)]} = 1,$$

lo que implica que

$$\frac{1}{T} \sum_{i=1}^T \mathbb{1}_{[A(S_j^i)(v_r) \neq f_i(v_r)]} = \frac{1}{2}. \tag{7.7}$$

Vamos a combinar las ecuaciones 7.3, 7.4, 7.6 y 7.7 para obtener la ecuación 7.1 lo que concluye la demostración. En primer lugar, utilizando 7.7 en la ecuación 7.6 tenemos que

$$\frac{1}{T} \sum_{i=1}^T L_{\mathcal{D}_i}(A(S_j^i)) \geq \frac{1}{4}.$$

Así pues, la ecuación 7.4 implica

$$\max_{1 \leq i \leq T} \frac{1}{k} \sum_{j=1}^k L_{\mathcal{D}_i}(A(S_j^i)) \geq \frac{1}{4}.$$

Por último, empleamos la ecuación 7.3 y llegamos a

$$\max_{1 \leq i \leq T} \mathbb{E}_{S \sim \mathcal{D}_i^m}[L_{\mathcal{D}_i}(A(S))] \geq \frac{1}{4}.$$

Por tanto, se cumple la ecuación 7.1 lo que significa que para todo algoritmo A' que recibe un conjunto de entrenamiento de m ejemplos de $\mathcal{X} \times \{0, 1\}$, existe una función $f : \mathcal{X} \rightarrow \{0, 1\}$ y una distribución \mathcal{D} sobre $\mathcal{X} \times \{0, 1\}$ tal que $L_{\mathcal{D}}(f) = 0$ y

$$\mathbb{E}_{S \sim \mathcal{D}^m}[L_{\mathcal{D}}(A'(S))] \geq \frac{1}{4}.$$

Por el lema 7.3 esto implica:

$$\mathbb{P}[L_{\mathcal{D}}(A'(S)) \geq 1/8] \geq 1/7.$$

7. Teorema de No Free Lunch

□

En resumen, para cualquier algoritmo de aprendizaje, queríamos mostrar que existe una distribución para la cual tiene una alta pérdida esperada, es decir, el algoritmo falla. Empezamos considerando un conjunto de distribuciones que se corresponden con el conjunto de todas las funciones binarias de nuestro espacio de entrada. Acotamos esto considerando la media de todas las distribuciones (y conjuntos de entrenamiento). A continuación, demostramos que la pérdida se acotaba superiormente considerando el conjunto de entrenamiento con pérdida mínima y promediando todas las distribuciones posibles, que tenían que ser al menos una mitad debido a la partición que hemos considerado. Demostramos que se verificaba la ecuación 7.1 y por último, empleamos el lema 7.3 para obtener el teorema en su forma original.

Veamos cuáles son las dos principales consecuencias que nos proporciona este teorema. La primera es que nos debemos olvidar de que existe un algoritmo inteligente capaz de optimizar cualquier problema, por tanto, tenemos que desarrollar muchos tipos diferentes de modelos para cubrir la gran variedad de datos que se dan en el mundo real. Y para cada modelo, puede haber muchos algoritmos diferentes que podemos utilizar para entrenar el modelo. La segunda consecuencia es que es necesario incorporar en el algoritmo cierto conocimiento específico del problema. Un tipo de ese conocimiento previo es que \mathcal{D} procede de alguna familia paramétrica específica de distribuciones, esto se puede ver en detalle en el libro *Understanding Machine Learning* [75].

8. Teorema de Aproximación Universal

En este capítulo se estudiará el teorema de aproximación universal. Este teorema establece que una sola capa intermedia es suficiente para aproximar, con una precisión arbitraria, cualquier función con un número finito de discontinuidades, siempre y cuando las funciones de activación de las neuronas ocultas sean no lineales. El teorema establece que las redes multicapa no añaden capacidad a menos que la función de activación de las capas sea no lineal.

Al principio del capítulo se introducirán algunas definiciones y teoremas útiles del análisis funcional, como el teorema de la convergencia dominada de Lebesgue, el teorema de Hahn-Banach en su versión geométrica con un corolario derivado de este y el teorema de Representación de Riesz. También se definirá el concepto de aproximador universal y las funciones de activación (sigmoide y ReLU) para las que se demostrará el teorema. Antes de presentar el teorema de aproximación universal se debe introducir el concepto de función discriminatoria y un lema que relaciona las funciones sigmoidales con las funciones discriminatorias. El capítulo termina con un teorema que permite demostrar que una red con anchura acotada y con una estructura más profunda conserva la propiedad de aproximación universal.

8.1. Definiciones y Teoremas útiles del Análisis Funcional

Muchas de las pruebas relacionadas con la aproximación universal se basan en resultados del análisis funcional. La mayoría de las demostraciones de los teoremas que vamos a utilizar se pueden encontrar en casi cualquier libro de análisis funcional como [6].

Teorema 8.1 (Teorema de la Convergencia Dominada de Lebesgue). *Sea X un espacio medible, sea μ una medida de Borel sobre X , sea $g \in L^1$ una función de $X \rightarrow \mathbb{R}$, y sea $\{f_n\}$ una secuencia de funciones medibles de $X \rightarrow \mathbb{R}$ tal que $|f_n(x)| \leq g(x)$ para todo $x \in X$ y $\{f_n\}$ converge puntualmente a una función f . Entonces f es integrable y*

$$\lim_{n \rightarrow \infty} \int f_n(x) d\mu(x) = \int f(x) d\mu(x). \quad (8.1)$$

Otro resultado importante y necesario para demostrar el teorema de aproximación universal es el teorema de Hahn-Banach. En este caso veremos su versión geométrica que consiste en encontrar condiciones suficientes para separar dos subconjuntos de un espacio vectorial.

Teorema 8.2 (Teorema de Hahn-Banach - Versión Geométrica). *Sea V un espacio vectorial normado y sean $A, B \subset V$ dos subconjuntos no vacíos, cerrados, disjuntos y convexos, tales que uno de ellos sea compacto. Entonces existe un funcional lineal $f \not\equiv 0$, algún $\alpha \in \mathbb{R}$ y un $\epsilon > 0$ tal que $f(x) \leq \alpha - \epsilon$ para cualquier $x \in A$ y $f(y) \geq \alpha + \epsilon$ para cualquier $y \in B$.*

Vamos a centrarnos en un corolario derivado de este teorema.

8. Teorema de Aproximación Universal

Corolario 8.1. Sea V un espacio vectorial normado sobre \mathbb{R} y $U \subset V$ un subespacio lineal de manera que $\overline{U} \neq V$. Entonces existe una aplicación lineal y continua $f : V \rightarrow \mathbb{R}$ con $f(x) = 0$ para cualquier $x \in U$, y $f \not\equiv 0$.

Demostración. Sea $z \in V \setminus \overline{U}$. Por el teorema anterior 8.2 (y observando que el conjunto $\{z\}$ es compacto), existe $f : V \rightarrow \mathbb{R}$ y $\alpha > 0$ tal que $f(x) < \alpha$ para cualquier $x \in \overline{U}$ y $f(z) > \alpha$, lo que significa que

$$f(x) < \alpha < f(z) \quad \forall x \in \overline{U}.$$

Como f es lineal, y \overline{U} es un subespacio, esto significa que, para cualquier $x_0 \in \overline{U}$, $\lambda \in \mathbb{R}$ tenemos que

$$f(\lambda x_0) = \lambda f(x_0) < \alpha$$

lo que significa que

$$f(x_0) < \frac{\alpha}{\lambda} \quad \forall \lambda > 0 \quad \text{y} \quad f(x_0) > \frac{\alpha}{\lambda} \quad \forall \lambda < 0$$

lo que implica que $f(x) = 0$ para todo $x \in \overline{U}$ y $f(z) > \alpha > 0$. \square

Pasamos a definir el conjunto de las funciones continuas en un espacio topológico.

Definición 8.1. Dado un espacio topológico Ω , definimos $C(\Omega) := \{f : \Omega \rightarrow \mathbb{R} \mid f \text{ es continua}\}$.

Teorema 8.3 (Teorema de Representación de Riesz). *Sea Ω un subconjunto de \mathbb{R}^n y $F : C(\Omega) \rightarrow \mathbb{R}$ un funcional lineal sobre el espacio de funciones reales continuas con dominio en Ω . Entonces existe una medida de Borel con signo μ sobre Ω tal que para cualquier $f \in C(\Omega)$, tenemos que*

$$F(f) = \int_{\Omega} f(x) d\mu(x) \tag{8.2}$$

Nos centramos ahora en algunas definiciones específicas del problema de la aproximación universal en redes neuronales.

Definición 8.2. Para $f : \mathbb{R} \rightarrow \mathbb{R}$ una función de activación, establece

$$\Sigma_n(f) = \langle \{f(y \cdot x + \theta) \mid y \in \mathbb{R}^n, \theta \in \mathbb{R}\} \rangle. \tag{8.3}$$

En la ecuación anterior, $y \cdot x$ representa el producto escalar en \mathbb{R}^n .

El conjunto $\Sigma_n(f)$ está formado por todas las funciones que puede calcular una red neuronal con una sola capa oculta y función de activación f .

Definición 8.3. Sea Ω un espacio topológico y $f : \mathbb{R} \rightarrow \mathbb{R}$ una función. Decimos que una red neuronal con función de activación f es un aproximador universal en Ω si $\Sigma_n(f)$ es denso en el conjunto de funciones continuas de Ω a \mathbb{R} y este conjunto se denota por $C(\Omega)$.

La definición anterior se podría, en teoría, modificar para que el conjunto de funciones generadas por una red neuronal con función de activación f sea denso en $C(\Omega)$. Esto se relacionaría

8.2. Teorema de Aproximación Universal para la función de activación Sigmoide y ReLU

más estrechamente con lo que entendemos por aproximación universal y encerraría la definición anterior (que supone que la red tiene una sola capa oculta). Sin embargo, por razones históricas, esta es la definición que ha prevalecido, y en ningún caso dificulta ninguno de los resultados, por lo que será la que empleemos.

Vamos a definir el concepto de función discriminatoria el cual es necesario para demostrar el teorema de aproximación universal.

Definición 8.4. Sea n un número natural. Decimos que una función de activación $f : \mathbb{R} \rightarrow \mathbb{R}$ es n -discriminatoria si la única medida de Borel con signo μ tal que

$$\int f(y \cdot x + \theta) d\mu(x) = 0 \quad \forall y \in \mathbb{R}^n, \theta \in \mathbb{R} \quad (8.4)$$

es la medida cero.

Definición 8.5. Decimos que una función de activación $f : \mathbb{R} \rightarrow \mathbb{R}$ es discriminatoria si es n -discriminatoria para cualquier n .

Definición 8.6. Sea n un número natural. Entonces definimos

$$I_n := [0, 1]^n = \{x = (x_1, \dots, x_n) \in \mathbb{R}^n \mid x_i \in [0, 1], i = 1, \dots, n\}$$

como el cubo unitario n -dimensional

Vamos a definir ahora las funciones de activación para las que demostraremos el teorema.

Definición 8.7. Una función $f : \mathbb{R} \rightarrow \mathbb{R}$ se llama sigmoide si satisface las siguientes propiedades:

$$\lim_{x \rightarrow \infty} f(x) = 1 \quad \text{y} \quad \lim_{x \rightarrow -\infty} f(x) = 0$$

Definición 8.8. La unidad lineal rectificada (también denominada ReLU) es una función $\mathbb{R} \rightarrow \mathbb{R}$ definida por:

$$\text{ReLU}(x) := \max(0, x)$$

8.2. Teorema de Aproximación Universal para la función de activación Sigmoide y ReLU

Un artículo publicado por George Cybenko en 1989 titulado *Approximation by Superpositions of a Sigmoidal Function* [21] desencadenó una serie de trabajos durante los siguientes años. En estos se intentaba determinar qué funciones de activación conducen a la propiedad de aproximación universal. En [21], se demostró para funciones sigmoidales continuas y esto es lo que pasamos a ver. Además, estudiaremos el caso de funciones de activación ReLU.

Teorema 8.4. Sea f una función discriminatoria continua. Entonces una red neuronal con función de activación f es un aproximador universal.

8. Teorema de Aproximación Universal

Demostración. Vamos a demostrar el teorema llegando a una contradicción. Supongamos que $\sum_n(f)$ no es denso en $C(I_n)$. Se deduce que $\overline{\sum_n(f)} \neq C(I_n)$. Entonces aplicamos el corolario del teorema de Hahn-Banach 8.1 para concluir que existe algún funcional lineal continuo $F : C(I_n) \rightarrow \mathbb{R}$ tal que $F \neq 0$ pero $F(g) = 0$ para cualquier $g \in \overline{\sum_n(f)}$. Por el teorema de Representación de Riesz, existe alguna medida de Borel μ tal que

$$F(g) = \int_{I_n} g(x) d\mu(x) \quad \forall g \in C(I_n)$$

Sin embargo, como para cualquier y y θ la función $f(y \cdot x + \theta)$ es un elemento de $\overline{\sum_n(f)}$, esto significa que para todo $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$ tenemos $\int f(y \cdot x + \theta) d\mu(x) = 0$, lo que significa que $\mu = 0$ (al ser f discriminatoria) y por tanto $F(g) = 0$ para cualquier $g \in C(I_n)$. Llegamos a un absurdo, ya que esto contradice el corolario del teorema de Hahn-Banach y, por tanto, termina la demostración. \square

Lema 8.1. *Todas las funciones sigmoidales acotadas y medibles por Borel son discriminatorias.*

Demostración. Sea f una función sigmoide acotada y Borel medible, y supongamos que para una medida dada $\mu \in M(I_n)$ tenemos que

$$\int_{I_n} f(y \cdot x + \theta) d\mu(x) = 0 \quad \forall y \in \mathbb{R}^n, \forall \theta \in \mathbb{R}.$$

Nuestro objetivo es demostrar que $\mu = 0$. Para ello, consideramos una función γ , obtenida fijando $\lambda, \phi \in \mathbb{R}$ y tomando el límite

$$\gamma(x) = \lim_{\lambda \rightarrow \infty} f(\lambda(y \cdot x + \theta) + \phi) = \begin{cases} 1 & \text{si } y \cdot x + \theta > 0 \\ f(\phi) & \text{si } y \cdot x + \theta = 0 \\ 0 & \text{si } y \cdot x + \theta < 0 \end{cases}$$

Por el teorema de la Convergencia Dominada de Lebesgue, tenemos que

$$\int_{I_n} \gamma(x) d\mu(x) = \lim_{\lambda \rightarrow \infty} \int_{I_n} f(\lambda(y \cdot x + \theta) + \phi) d\mu(x) = 0.$$

Usando la información anterior, llegamos a

$$\begin{aligned} \int_{I_n} \gamma(x) d\mu(x) &= \int_{H_{y,\theta}^+} 1 d\mu(x) + \int_{\Pi_{y,\theta}} f(\phi) d\mu(x) + \int_{H_{y,\theta}^-} 0 d\mu(x) \\ &= \mu(H_{y,\theta}^+) + f(\phi)\mu(\Pi_{y,\theta}) \\ &= 0. \end{aligned}$$

donde

$$\begin{aligned} H_{y,\theta}^+ &:= \{x \in I_n \mid y \cdot x + \theta > 0\}, \\ \Pi_{y,\theta}^+ &:= \{x \in I_n \mid y \cdot x + \theta = 0\}, \\ H_{y,\theta}^- &:= \{x \in I_n \mid y \cdot x + \theta < 0\}. \end{aligned}$$

8.2. Teorema de Aproximación Universal para la función de activación Sigmoide y ReLU

Esto es cierto para cualquier elección de y, θ . Como esto es cierto para cualquier ϕ y $f(\phi) \rightarrow 1$ cuando $\phi \rightarrow \infty$, tenemos que

$$\mu(H_{y,\theta}^+) + \mu(\Pi_{y,\theta}) = 0.$$

De forma similar, si $\phi \rightarrow -\infty$, obtenemos que $f(\phi) \rightarrow 0$ y

$$\mu(H_{y,\theta}^+) = 0.$$

Ahora consideremos el funcional $F : L^\infty(\mathbb{R}) \rightarrow \mathbb{R}$ definido como:

$$F(h) := \int_{I_n} h(y \cdot x) d\mu(x).$$

Como $L^\infty(\mathbb{R})$ es el espacio de las funciones acotadas, esta integral (y por tanto el funcional) está bien definida para cualquier $h \in L^\infty(\mathbb{R})$. Si tomamos $\mathbb{1}_{[\theta, \infty)}$ como la función indicadora del intervalo $[\theta, \infty)$, obtenemos que

$$F(\mathbb{1}_{[\theta, \infty)}) = \int_{I_n} \mathbb{1}_{[\theta, \infty)}(y \cdot x) d\mu(x) = \mu(H_{y,\theta}^+) + \mu(\Pi_{y,\theta}) = 0.$$

Del mismo modo, para cualquier intervalo abierto (θ, ∞) , tenemos que

$$F(\mathbb{1}_{(\theta, \infty)}) = \int_{I_n} \mathbb{1}_{(\theta, \infty)}(y \cdot x) d\mu(x) = \mu(H_{y,\theta}^+) = 0.$$

Utilizando la linealidad, obtenemos que $F(h) = 0$ para la función indicadora h de cualquier intervalo. Se deduce que $F(h) = 0$ para cualquier función simple. Utilizando que las funciones simples son densas en $L^\infty(\mathbb{R})$, se llega a que $F = 0$. Además, como \sin y \cos son elementos de $L^\infty(\mathbb{R})$, podemos utilizar que

$$\begin{aligned} F(\cos) + iF(\sin) &= \int_{I_n} (\cos(y \cdot x) + i \sin(y \cdot x)) d\mu(x) \\ &= \int_{I_n} e^{iy \cdot x} d\mu(x) \\ &= 0. \end{aligned}$$

Esto es cierto para todo $y \in \mathbb{R}^n$, lo que significa que la transformada de Fourier de μ es 0. Esto solo es posible si $\mu = 0$, lo que concluye la demostración. \square

Acabamos de demostrar que la propiedad de aproximación universal es válida para redes con funciones sigmoides continuas. Sin embargo, la mayoría de algoritmos de última generación utilizan actualmente ReLU como función de activación (ReLU es una de las funciones de activación que hemos utilizado en nuestros modelos). Aunque los resultados generales, como los que aparecen en el artículo [53], se aplican a esta función, nos apoyaremos en el artículo *An Overview Of Artificial Neural Networks for Mathematicians* [47] escrito por Guilhoto que siguiendo el trabajo de Cybenko [21] demuestra que la función ReLU es 1-discriminatoria y utiliza un lema para mostrar que esto implica la propiedad de aproximación universal para funciones con entradas de dimensión finita.

8. Teorema de Aproximación Universal

Lema 8.2. *La función ReLU es 1-discriminatoria.*

Demostración. Sea μ una medida de Borel con signo, y supongamos que se cumple lo siguiente para todo $y \in \mathbb{R}$ y $\theta \in \mathbb{R}$:

$$\int \text{ReLU}(yx + \theta) d\mu(x) = 0.$$

Queremos demostrar que $\mu = 0$. Para ello, vamos a construir una función sigmoide acotada y continua (por tanto medible por Borel) a partir de la resta de dos funciones ReLU con parámetros diferentes. En particular, consideramos la función

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \in [0, 1] \\ 1 & \text{si } x > 1 \end{cases} \quad (8.5)$$

Entonces cualquier función de la forma $g(x) = f(yx + \theta)$ con $y \neq 0$ puede describirse como

$$g(x) = \text{ReLU}(yx + \theta_1) - \text{ReLU}(yx + \theta_2) \quad (8.6)$$

estableciendo $\theta_1 = \frac{-\theta}{y}$ y $\theta_2 = \frac{1-\theta}{y}$. Si $y = 0$, entonces en su lugar se establece

$$g(x) = f(\theta) = \begin{cases} \text{ReLU}(f(\theta)) & \text{si } f(\theta) \geq 0 \\ -\text{ReLU}(-f(\theta)) & \text{si } f(\theta) \leq 0 \end{cases}$$

Lo que significa que para cualquier $y \in \mathbb{R}$, $\theta \in \mathbb{R}$

$$\begin{aligned} \int f(yx + \theta) d\mu(x) &= \int (\text{ReLU}(yx + \theta_1) - \text{ReLU}(yx + \theta_2)) d\mu(x) \\ &= \int \text{ReLU}(yx + \theta_1) d\mu(x) - \int \text{ReLU}(yx + \theta_2) d\mu(x) \\ &= 0 - 0 \\ &= 0. \end{aligned}$$

Por el lema anterior, f es discriminatoria, y además $\mu = 0$. □

Lema 8.3. *Si $\sum_1(f)$ es denso en $C([0, 1])$ entonces $\sum_n(f)$ es denso en $C([0, 1]^n)$*

Demostración. Usaremos el hecho de que $\langle \{g(a \cdot x) \mid a \in \mathbb{R}^n, g \in C([0, 1])\} \rangle$ es denso en $C([0, 1]^n)$. Es decir, dada cualquier función $h \in C([0, 1]^n)$ y $\epsilon > 0$ existen funciones g_k en $C([0, 1])$ tales que

$$|h(x) - \sum_{k=1}^N g_k(a_k \cdot x)| < \frac{\epsilon}{2}.$$

Si ahora examinamos cada función $g_k(a_k \cdot x)$, y utilizamos la suposición de que $\sum_1(f)$ es densa en $C([0, 1])$, concluimos que para cualquier función de este tipo, existe una suma de funciones tal que

$$|g_k(a_k \cdot x) - \sum_{i=1}^{N_k} f(y_{k,i} \cdot x + \theta_{k,i})| < \frac{\epsilon}{2k}.$$

8.2. Teorema de Aproximación Universal para la función de activación Sigmoidal y ReLU

Aplicando la desigualdad triangular, obtenemos que

$$\begin{aligned} |h(x) - \sum_{k=1}^N \sum_{i=1}^{N^k} f(y_{k,i} \cdot x + \theta_{k,i})| &< |h(x) - \sum_{k=1}^N g_k(a_k \cdot x)| + k\left(\frac{\epsilon}{2k}\right) \\ &< \frac{\epsilon}{2} + \frac{\epsilon}{2} \\ &= \epsilon. \end{aligned}$$

Esto demuestra que podemos acercarnos tanto como queramos a cualquier función en $C([0, 1]^n)$ utilizando funciones en $\Sigma_n(f)$. \square

Como consecuencia directa del teorema 8.4 y de los lemas anteriores, llegamos al resultado deseado:

Corolario 8.2 (Teorema de Aproximación Universal). *Las redes neuronales con funciones de activación ReLU o funciones de activación sigmoides son aproximadores universales.*

Hasta aquí hemos demostrado que una red neuronal con una sola capa oculta y un número suficiente de nodos es capaz de aproximar uniformemente cualquier función continua. Sin embargo, este diseño difiere drásticamente de la arquitectura habitual utilizada en las redes neuronales, en las que se emplean multitud de capas (lo que da lugar al término *aprendizaje profundo*, como veremos en el capítulo 10). Vamos a demostrar que una red con anchura acotada conserva la propiedad de aproximación universal. El siguiente resultado consigue este objetivo, a costa de utilizar una estructura más profunda (más capas) para la red.

Teorema 8.5. *Sea $f : [0, 1]^d \rightarrow [0, 1]$ la función calculada por una red neuronal de una sola capa con función de activación ReLU de dimensión de entrada d , dimensión de salida 1 y n nodos ocultos. Entonces existe otra red neuronal, con n capas ocultas y anchura $d+2$, que calcula la misma función f cuyas funciones de activación son funciones ReLU.*

La prueba de este teorema tiene como estrategia utilizar d de los nodos de cada capa para almacenar la entrada inicial, un nodo para calcular cada uno de los nodos de la red original, y el último nodo para almacenar el valor de la suma de las funciones.

Demostración. Dado que f es generada por una red ReLU de una sola capa con n nodos ocultos, es de la forma

$$f(x) = \text{ReLU}\left(b + \sum_{i=1}^n w_i \text{ReLU}(g_i(x))\right),$$

donde cada g_i es de la forma $g_i = y_i \cdot x + b_i$ con $y_i \in \mathbb{R}^d$ y $b_i \in \mathbb{R}$. Como $[0, 1]^d$ es compacto y cualquier función generada por una red ReLU es continua, el término $\sum_{j=1}^i w_j \text{ReLU}(g_j(x))$ alcanza un mínimo para cualquier $i \leq n$ y, por tanto, existe un número $T > 0$ tal que

$$T + \sum_{j=1}^i w_j \text{ReLU}(g_j(x)) > 0$$

para cualquier $x \in [0, 1]^d$ y cualquier $i \leq n$.

8. Teorema de Aproximación Universal

Consideremos ahora una nueva red en la que cada capa oculta utiliza d nodos para copiar las entradas originales. Además, cada capa oculta i^{th} también está equipada con otro nodo que calcula la función $\text{ReLU}(g_i(x))$ de los nodos de entradas copiados en la capa anterior, y un último nodo que computa

$$\text{ReLU}\left(T + \sum_{j=1}^{i-1} w_j \text{ReLU}(g_j(x))\right) = T + \sum_{j=1}^{i-1} w_j \text{ReLU}(g_j(x)) > 0$$

haciendo una combinación lineal de los dos nodos adicionales de la capa anterior. En resumen, la capa i^{th} hasta $i < n$ calculará la función

$$h^i(x) = (x_1, \dots, x_d, \text{ReLU}(g_i(x)), \text{ReLU}\left(T + \sum_{j=1}^{i-1} w_j \text{ReLU}(g_j(x))\right)).$$

La última capa calcula entonces

$$\begin{aligned} h^{n+1}(x) &= \text{ReLU}[w_n \text{ReLU}(g_n(x)) + \text{ReLU}\left(T + \sum_{j=1}^{n-1} w_j \text{ReLU}(g_j(x))\right) - T + b] \\ &= \text{ReLU}\left(b + \sum_{i=1}^n w_i \text{ReLU}(g_i(x))\right) \\ &= f(x), \end{aligned}$$

que completa la prueba utilizando n capas ocultas, cada una de ellas de anchura $d + 2$. \square

En sus orígenes, el teorema de aproximación universal fue demostrado para funciones de activación sigmoidales, pero existen versiones para otro tipo de funciones de activación como es el caso de las funciones ReLU que acabamos de explicar.

Al igual que hemos demostrado que cualquier función continua en el cubo unitario puede ser aproximada por una red neuronal con una capa con una función sigmoidal o ReLU continua y no lineal arbitraria también existe una versión del teorema para funciones de clasificación que se puede consultar en [21].

9. Aprendizaje Automático

En este capítulo se presenta el término aprendizaje automático. También se explica en más profundidad el concepto de algoritmo de aprendizaje y qué componentes tiene, así como los problemas de generalización que aparecen en un algoritmo de aprendizaje automático.

9.1. Aprendizaje Automático

Antiguamente, para poder resolver un problema con la ayuda de un ordenador era necesario diseñar e implementar un algoritmo muy específico, mediante el cual el ordenador supiera lo que tenía que hacer. Esta estrategia en algunos casos específicos de algunos problemas no funcionaba.

El aprendizaje automático o *machine learning* ([7], [46]) es una rama de la computación que proporciona mecanismos a través de los cuales el ordenador tiene la capacidad de aprender a resolver un problema por sí mismo. El programador es el que se ocupa de diseñar el algoritmo de aprendizaje adecuado para el problema planteado que se desea resolver y es el ordenador el que se encarga de resolverlo, para ello hace uso de las diferentes heurísticas de aprendizaje incorporadas en el algoritmo desarrollado por el programador y de los datos de los que dispone.

Otra definición se encuentra en [4] y fue realizada por Arthur Samuel, de IBM, él definió el aprendizaje automático como el campo de estudio que dota a los ordenadores de la capacidad de aprender a resolver problemas para los cuales no han sido explícitamente programados. En inteligencia artificial, un proceso por el cual un ordenador es capaz de mejorar su habilidad en la resolución de un problema mediante la adquisición de conocimiento obtenido gracias a la experiencia es lo que se conoce como aprendizaje.

9.2. Algoritmos de aprendizaje

Un algoritmo de aprendizaje automático es aquel que puede aprender a partir de unos datos. En 1997, Tom Mitchell [62] definió lo que entendía él por aprendizaje: *Se dice que un programa de computadora 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.* En las diferentes secciones vamos a describir ejemplos de los diferentes tipos de tareas, medidas de rendimiento y experiencias que podemos emplear para crear algoritmos de aprendizaje automático, para ello seguiremos el libro *Deep Learning* [46].

9.2.1. La tarea

El aprendizaje automático nos permite abordar tareas que son demasiado difíciles de resolver con programas fijos escritos y diseñados por seres humanos. El proceso de aprendizaje en sí mismo no es la tarea, es nuestro medio para alcanzar la capacidad de realizar la tarea.

En general, las tareas de aprendizaje automático se describen en términos de cómo debe actuar el sistema de aprendizaje a partir de un ejemplo. Un ejemplo es una colección de características de aprendizaje automático que se han medido de manera cuantitativa a partir de algún evento u objeto que queremos que procese el sistema de aprendizaje automático. Normalmente un ejemplo lo representamos como un vector $x \in \mathbb{R}^n$, donde cada componente del vector es una característica.

Vamos a nombrar algunas de las muchas tareas que se pueden resolver con aprendizaje automático, siendo las más comunes:

- *Clasificación:* En esta clase de tareas el programa especifica a cuál de las k categorías establecidas pertenece cada entrada. Para poder resolver esta tarea, normalmente es necesario que el algoritmo de aprendizaje logre encontrar una función

$$f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$$

Si se produce una salida $y = f(x)$ entonces el modelo ha asignado a la entrada definida por x la clase identificada por el valor y .

Hay otras variantes de la tarea de clasificación, como por ejemplo donde f genera una distribución de probabilidad sobre las clases consideradas.

Un ejemplo de tarea de clasificación es el reconocimiento de objetos, donde la entrada es una imagen y la salida es un código numérico que identifica el objeto en la imagen. El problema que tratamos en este trabajo es un problema de clasificación, ya que buscamos dada una posición en un tablero de ajedrez una función que nos permita clasificar diferentes características de la posición en una serie de categorías.

- *Regresión:* en este tipo de tareas, el programa de computadora debe predecir un valor numérico dada una entrada. Para solucionar esta tarea, se le pide al algoritmo que genere una función

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

Un ejemplo de este tipo es la predicción de precios futuros de los valores a partir de una serie de características. También se emplean para el comercio algorítmico.

- *Transcripción:* en esta tarea, se le pide al modelo que observe una representación relativamente desestructurada de algún tipo de datos y que transcriba la información en forma de texto discreto.

Un ejemplo es el reconocimiento de voz, el programa recibe una forma de onda de audio y emite una secuencia de caracteres que describen las palabras pronunciadas en la grabación de audio.

- *Traducción automática:* la tarea de traducción automática tiene como entrada una secuencia de símbolos de algún idioma y el sistema de aprendizaje convierte la entrada en una secuencia de símbolos de otro idioma.

- *Detección de anomalías*: en esta clase de tareas el programa se encarga de observar un conjunto de eventos u objetos y señalar algunos de ellos como atípicos o inusuales.

Como ejemplo de tarea de detección de anomalías tenemos la detección de fraudes con tarjetas de crédito. La empresa de tarjetas de crédito modela los hábitos de compra de los clientes y así detectará el uso inadecuado de sus tarjetas.

- *Síntesis y muestreo*: en esta clase de tareas el modelo de aprendizaje a partir de los datos de entrenamiento es capaz de generar nuevos casos similares a estos. Emplear el aprendizaje automático para la síntesis y el muestreo de grandes volúmenes de contenido nos permite ahorrarnos tiempo y costes.

Un ejemplo de uso se da en los videojuegos, se puede prescindir de un artista que etiquete manualmente cada píxel al poder generar de manera automática texturas para objetos grandes o paisajes.

Existen casos donde queremos que el procedimiento de síntesis o muestreo genere dada una entrada un tipo específico de salida. Un ejemplo sería la tarea de síntesis de voz.

- *Imputación de valores perdidos*: en este tipo de tareas, el algoritmo de aprendizaje automático recibe una nuevo ejemplo $x \in \mathbb{R}^n$ pero con algunas entradas x_i de x perdidas. El algoritmo debe aportar una predicción de los valores de las entradas perdidas.
- *Eliminación de ruido*: en esta clase de tareas, el algoritmo recibe como entrada un ejemplo corrupto $\tilde{x} \in \mathbb{R}^n$ que ha sido obtenido a partir de un ejemplo limpio $x \in \mathbb{R}^n$ mediante un proceso de corrupción desconocido. El algoritmo debe ser capaz de predecir el ejemplo limpio $x \in \mathbb{R}^n$ a partir de su versión corrupta, es decir, predecir la distribución de probabilidad condicionada $p(x|\tilde{x})$
- *Agrupamiento o Clustering*: en este tipo de tareas, el algoritmo de aprendizaje se encarga de agrupar las entradas en diferentes clases. A diferencia del caso de clasificación, en el cual se especificaban las clases previamente, ahora es el algoritmo el que se encarga de crear dichas clases en función de los datos de entrada.

Se suele emplear para analizar conjunto de datos muy grandes por ejemplo para identificar comunidades en las redes sociales.

9.2.2. La medida de rendimiento

Para evaluar las capacidades de un algoritmo de aprendizaje automático es necesario diseñar una medida cuantitativa de su rendimiento. En general, esta medida de rendimiento depende de la tarea que realiza el sistema. Para tareas como clasificación o transcripción a menudo medimos la precisión del modelo, esto es, la proporción de ejemplos para los que el modelo produce un resultado correcto. También podemos obtener información equivalente midiendo la tasa de error, es decir, la proporción de ejemplos para los que el modelo produce una salida incorrecta. Hay casos donde esta medida no tiene sentido como es el caso de la regresión, donde normalmente empleamos como medida el error cuadrático medio que se comete al predecir valores.

Por lo general, nos interesa saber cómo de bien se desempeña nuestro algoritmo de aprendizaje automático con datos que no ha visto anteriormente, es decir, datos que no han sido

9. Aprendizaje Automático

utilizados para entrenar el modelo, ya que esto determina lo bien que funcionará cuando se implemente en el mundo real. Por lo tanto, vamos a dividir el conjunto de datos que tenemos en dos subconjuntos, uno es el formado por los datos de entrenamiento y el resto de datos conforman el equipo de prueba o test de datos sobre el que evaluamos las medidas de desempeño o rendimiento.

La elección de la medida de rendimiento a priori puede parecer sencilla y objetiva pero a menudo es difícil escoger una medida que se corresponda bien con el comportamiento deseado del sistema.

9.2.3. La experiencia

Los algoritmos de aprendizaje automático se pueden clasificar en términos generales como supervisados o no supervisados según la filosofía o experiencia utilizada en el proceso de adquisición de conocimiento.

- *Algoritmos de aprendizaje supervisado:* en este tipo de aprendizaje los ejemplos de entrenamiento van acompañados de la salida correcta en forma de etiqueta u objetivo que el sistema deberá ser capaz de reproducir, es decir, tenemos una función de manera que $f(x) = y$ donde conocemos x e y pero no conocemos la fórmula que los relaciona. El entrenamiento de un modelo de aprendizaje supervisado consiste en ajustar sus parámetros para que sea capaz de reproducir una salida lo más parecida posible a la deseada. Una vez entrenado el modelo, lo importante es que sea capaz de generalizar correctamente. Esta capacidad de generalización consiste en que el modelo proporcione salidas adecuadas para datos de entrada diferentes a los datos utilizados durante su entrenamiento. Normalmente en esta categoría se engloban por ejemplo los algoritmos de clasificación y regresión mencionados previamente.
- *Algoritmos de aprendizaje no supervisado:* en este tipo de modelos se construyen descripciones, hipótesis o teorías a partir de un conjunto de hechos u observaciones, sin que exista información adicional acerca de cómo deberían clasificarse los ejemplos del conjunto de entrenamiento, no conocemos la salida deseada. Como ejemplo tenemos los métodos de agrupamiento o clustering y la detección de anomalías.

El aprendizaje supervisado y el aprendizaje no supervisado no son términos formalmente definidos, las líneas entre ellos a menudo son borrosas. Se pueden utilizar muchas técnicas de aprendizaje automático para realizar ambas tareas pero ambos conceptos nos ayudan a categorizar aproximadamente algunas de las cosas que hacemos con los algoritmos de aprendizaje automático.

En nuestro caso, donde tenemos tableros de entrada con la posición de cada pieza en el tablero y conocemos la salida deseada, estamos en un problema de aprendizaje supervisado donde no conocemos cómo llegar a partir de la entrada a la salida deseada.

9.3. El problema de la generalización, overfitting y underfitting

El reto principal del aprendizaje automático es que el algoritmo debe tener un buen comportamiento con entradas nuevas que no hayan sido observadas previamente, y no solo con

9.3. El problema de la generalización, overfitting y underfitting

los datos que nuestro modelo fue entrenado. La capacidad de predecir correctamente las entradas no observadas anteriormente recibe el nombre de generalización.

Normalmente, a la hora de entrenar un modelo de aprendizaje automático, tenemos acceso a un conjunto de datos de entrenamiento, por tanto, podemos calcular alguna medida de error en el conjunto de entrenamiento, llamada error de entrenamiento y tratar de minimizar dicho error. Así pues, tenemos un problema de optimización, siendo la función a optimizar el error cometido al clasificar un conjunto de datos de entrenamiento y pretendemos minimizar lo máximo posible dicho error. La principal diferencia entre un problema de aprendizaje automático y uno de optimización es que queremos que el error de generalización o de test sea también lo más pequeño posible. El error de generalización se define como el valor esperado del error en una nueva entrada.

Para dar una estimación del error de generalización de un modelo de aprendizaje automático se mide su rendimiento en un conjunto de test, que es un conjunto de datos de prueba que se recogen por separado del conjunto de entrenamiento.

Ahora surge la pregunta de cómo podemos mejorar la capacidad de predicción de un modelo en el conjunto de datos de prueba cuando solo podemos observar el conjunto de datos de entrenamiento. Esta pregunta se puede contestar haciendo uso del campo de la teoría de la estadística y la probabilidad. Si el conjunto de entrenamiento y el de prueba se recogen de forma arbitraria, no podremos hacer mucho pero podremos hacer algunos progresos si se nos permite hacer algunas suposiciones sobre cómo se recogen los datos de entrenamiento y de prueba.

Los datos de entrenamiento y de prueba se generan mediante una distribución de probabilidad sobre los conjuntos de datos denominada proceso de generación de datos. Las suposiciones que tendremos que hacer son un conjunto de suposiciones conocidas colectivamente como i.i.d., trabajamos con un conjunto de variables aleatorias independientes e idénticamente distribuidas. Esto significa que, las muestras del conjunto de datos con el que trabajamos son todas independientes entre sí y que el conjunto de entrenamiento y el conjunto de prueba están idénticamente distribuidos, todos los ejemplos han sido extraídos de una misma variable aleatoria con cierta distribución de probabilidad. La misma distribución se utiliza para generar cada ejemplo de entrenamiento y cada ejemplo de prueba. Esta distribución subyacente compartida recibe el nombre de distribución generadora de datos, p_{data} . Este marco probabilístico y los supuestos de i.i.d. nos permiten estudiar matemáticamente la relación entre el error cometido en el entrenamiento y el error de generalización o de test.

Una consecuencia inmediata que podemos observar es que el error de entrenamiento esperado de un modelo seleccionado al azar es igual al error de test esperado de ese modelo. Supongamos que disponemos de una distribución de probabilidad $p(x, y)$ de la cual tomamos muestras aleatorias para generar el conjunto de entrenamiento y el conjunto de prueba. Para algún valor fijo, sea este w , el error cometido esperado en ambos conjuntos es el mismo, ya que ambas esperanzas se calculan tomando la misma distribución de probabilidad. La única diferencia es el nombre asignado a cada conjunto de datos que muestreamos pero el error esperado es el mismo al estar ambos conjuntos idénticamente distribuidos.

Cuando realizamos una aplicación real de un algoritmo de aprendizaje automático no fijamos los parámetros del modelo antes de muestrear ambos conjuntos de datos. En primer lugar, muestreamos el conjunto de entrenamiento, después elegimos los parámetros que reduzcan el error cometido en el conjunto de entrenamiento y posteriormente muestreamos el conjunto

9. Aprendizaje Automático

de test comprobando si el error de este conjunto es similar al error cometido en el conjunto de entrenamiento. Al realizar este proceso, el error de test esperado es mayor o igual que el error cometido durante el entrenamiento. Por tanto, tenemos dos factores que determinan el rendimiento de un algoritmo de aprendizaje automático:

1. La capacidad de reducir el error de entrenamiento.
2. La capacidad de minimizar la diferencia entre el error de entrenamiento y el de generalización.

Estos dos fenómenos se corresponden con los dos principales retos del aprendizaje automático, los dos problemas más alarmantes a los que nos enfrentamos cuando diseñamos un modelo de aprendizaje, el *underfitting* o subajuste y el *overfitting* o sobreajuste. Cuando el modelo no es capaz de reducir el error lo suficiente en el conjunto de entrenamiento tenemos un caso de *underfitting*, el número de parámetros no es suficiente para modelar la complejidad del problema que estamos resolviendo. El fenómeno que se produce cuando la diferencia entre el error de entrenamiento y el de prueba es demasiado grande es lo que se conoce como *overfitting* que intentaremos prevenir mediante mecanismos de regularización que veremos en el siguiente capítulo.

Podemos en parte controlar la aparición de ambos fenómenos dependiendo de la complejidad de nuestros modelos. Los modelos con poca complejidad pueden tener dificultades para ajustarse al conjunto de entrenamiento y los modelos con alta complejidad pueden memorizar propiedades del conjunto de entrenamiento que no le sirven en el conjunto de test siendo incapaces de reconocer nuevos datos. Veamos un ejemplo en el que tenemos una serie de puntos que queremos ajustar mediante una gráfica.

Si tenemos un modelo infraajustado, se produce *underfitting*, no tenemos suficientes parámetros para capturar las tendencias del sistema. Imaginemos que tenemos datos de naturaleza parabólica que los intentamos ajustar con una función lineal de un solo parámetro. Como la función no tiene complejidad necesaria para ajustarse a los datos, acabamos teniendo un predictor pobre. En términos matemáticos, el modelo tendrá un sesgo elevado.

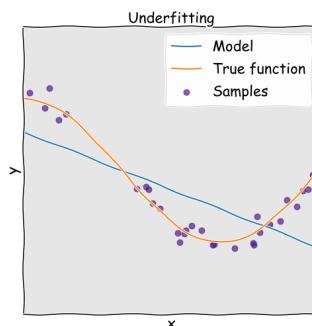


Figura 9.1.: Ejemplo de underfitting [9]

Si por el contrario hemos sobreajustado el modelo, significa que tenemos demasiados parámetros y, por tanto, hemos construido un modelo demasiado complejo. Imaginemos el mismo ejemplo de manera que el verdadero sistema es una parábola pero ahora utilizamos

9.3. El problema de la generalización, overfitting y underfitting

un polinomio de orden superior para ajustarlo. Debido a que tenemos ruido natural en los datos utilizados para el ajuste, el modelo excesivamente complejo trata estas fluctuaciones y el ruido como si fueran propiedades intrínsecas del sistema e intenta ajustarse a ellas. Como resultado obtenemos un modelo que tiene una alta varianza y como consecuencia no obtendremos predicciones consistentes de los resultados futuros.

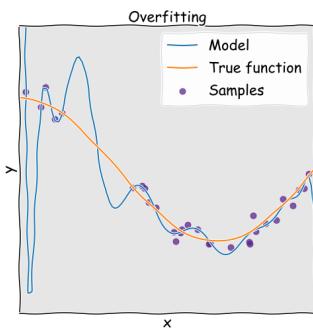


Figura 9.2.: Ejemplo de overfitting [9]

Para encontrar la complejidad óptima tenemos que entrenar el modelo cuidadosamente y luego validarla frente a los datos de test. El rendimiento del modelo frente al conjunto de validación mejorará inicialmente pero acabará sufriendo y desmejorando. El punto de inflexión representa el modelo óptimo.

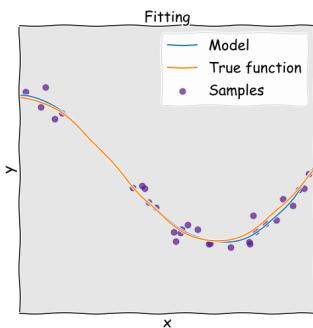


Figura 9.3.: Ejemplo de un modelo bien escogido [9]

Con este ejemplo hemos puesto de manifiesto cómo la complejidad del modelo escogido afecta a la capacidad de predicción del mismo, otro factor importante es la forma de buscar, dentro de la clase de funciones con las que trabajamos, la función óptima para la tarea que estamos desarrollando. Ahora vamos a ver los componentes principales para la construcción de un algoritmo de aprendizaje automático.

9.4. Construcción de un algoritmo de aprendizaje automático

La mayor parte de los algoritmos de aprendizaje automático se construyen teniendo en cuenta cuatro componentes principales:

- Recopilar un conjunto de datos asociados al problema.
- Diseñar una función de coste apropiada para el problema, también conocida como función de pérdida.
- Aplicar un procedimiento de optimización para minimizar la función de coste.
- Seleccionar un modelo y establecer sus hiperparámetros.

Dado el conjunto de datos y el modelo escogido, el objetivo será minimizar el valor de la función de coste utilizando el procedimiento de optimización. En muchos casos, la función de coste será una función que no podemos evaluar realmente debido a razones computacionales, nos encontramos con un problema de optimización no resoluble por los métodos tradicionales de optimización y el espacio de parámetros de nuestro modelo puede ser infinito, por lo que es inviable computacionalmente encontrar la frontera de clasificación óptima. En la práctica, los algoritmos de aprendizaje automático tratan de buscar una función que reduce significativamente el error de entrenamiento en un tiempo aceptable, ya que no pueden encontrar la función óptima asociada a la restricción del problema.

En nuestro caso tenemos como conjunto de datos el conjunto de posiciones de ajedrez, en total cincuenta mil posiciones de las que extraeremos algunos aspectos relevantes. La función de pérdida seleccionada es la entropía cruzada categórica, en inglés *Categorical Cross-Entropy*. El optimizador por el que nos hemos decantado ha sido *Adam*, un método de descenso del gradiente estocástico que veremos posteriormente, y el modelo escogido son las redes neuronales.

10. Aprendizaje Profundo

La rama del aprendizaje automático dedicada al estudio de las redes neuronales se conoce como aprendizaje profundo y es un área de investigación muy activa. En este capítulo se explica de manera breve el concepto de aprendizaje profundo para centrarnos posteriormente en las redes neuronales. Se definirá formalmente el modelo y se explicará cómo se lleva a cabo el entrenamiento de una red neuronal. También se presentan las funciones de activación y se mostrarán algunas de las más empleadas. Por último, se estudian técnicas de regularización, ya que las redes neuronales tienen un gran poder de aproximación y necesitan este tipo de técnicas para evitar el sobreajuste.

El aprendizaje profundo, en inglés *deep learning* ([62], [63]) es un subconjunto del aprendizaje automático que imita la forma en la que los seres humanos obtenemos ciertos tipos de conocimiento, está basado en redes neuronales artificiales. El proceso de aprendizaje se denomina profundo debido a que la estructura de redes neuronales artificiales se compone de varias capas de entrada, salida y ocultas, y el grafo de estas puede ser tan profundo como sea necesario. Cada una de estas capas como veremos a continuación transforma los datos de entrada en información que la capa siguiente usa para realizar una tarea de predicción determinada. Algunos problemas que nos permite resolver el aprendizaje profundo son: identificación de objetos en imágenes, reconocimiento facial, método antifraude, reconocimiento de voz, traducción automática, etc. Pasamos ahora a adentrarnos en el mundo de las redes neuronales.

10.1. Introducción a las Redes Neuronales

Las redes neuronales [46], también llamadas redes neuronales artificiales o perceptrones multicapa (MLP) son los modelos de aprendizaje profundo por excelencia. El objetivo de una red es aproximar una función f^* . Por ejemplo, una red neuronal que se emplea para resolver una tarea de clasificación, tendrá como misión aprender una función $y = f^*(x)$ que asigna una entrada x a una categoría y . Para llevar a cabo esta tarea la red define una función $\hat{y} = h(x; w)$ y aprende el valor de los parámetros w que dan lugar a la mejor aproximación de la función.

Las redes con las que vamos a trabajar en esta sección se conocen como redes neuronales prealimentadas o *feedforward*. Estos modelos se denominan así, ya que la información fluye en una sola dirección, se introduce por las capas de entrada el vector x , en las capas intermedias se llevan a cabo una serie de cálculos que se utilizan para definir h y finalmente se obtiene el resultado \hat{y} , por una capa de salida. En estos modelos no hay conexiones de retroalimentación en las que las salidas del modelo se retroalimenten a sí mismas, sin embargo, existe un tipo de redes neuronales que sí permite conexiones hacia atrás, son las redes neuronales recurrentes pero estas no las comentaremos en este trabajo.

Las redes prealimentadas se llaman redes porque suelen representarse componiendo muchas funciones diferentes. El modelo se asocia a un grafo acíclico dirigido que describe cómo se componen las funciones. Por ejemplo, podemos tener tres funciones h_1, h_2, h_3 y componerlas para obtener una función $h(x) = h_3(h_2(h_1(x)))$. Estas estructuras son las más empleadas en las redes neuronales y esta composición se ve como un conjunto de capas apiladas. En el ejemplo que hemos puesto, una red neuronal que intentase aprender la función f tendría tres capas. La primera capa calcularía h_1 , la segunda h_2 y la tercera capa calcularía h_3 . El número total de capas de nuestro modelo es lo que da la profundidad de este y de ahí el nombre de aprendizaje profundo, ya que proviene de esta terminología. Las arquitecturas de aprendizaje profundo normalmente cuentan con una gran cantidad de capas anidadas, creando de esta forma redes neuronales de mucha profundidad, permitiendo así, poder aproximar virtualmente cualquier función. Las capas de una red neuronal que explicaremos en detalle más adelante suelen ser las siguientes, la primera capa es la capa de entrada, la capa final, la cual da la respuesta \hat{y} para una entrada x es la capa de salida y las capas intermedias son las capas ocultas del modelo.

Durante el entrenamiento de una red neuronal, buscamos que $h(x)$ coincida con $f^*(x)$. Los datos de entrenamiento nos proporcionan ejemplos ruidosos y aproximados de $f^*(x)$ evaluados en diferentes puntos de entrenamiento. Cada ejemplo x va acompañado de una etiqueta $y \approx f^*(x)$. Los ejemplos de entrenamiento especifican directamente lo que debe hacer la capa de salida, la capa final de la red, en cada punto x ; debe producir un valor que se acerca a y . El comportamiento de las otras capas no está directamente especificado por los datos de entrenamiento. El algoritmo de aprendizaje debe decidir cómo utilizar estas capas para implementar de la mejor manera una aproximación de f^* . Dado que los datos de entrenamiento no muestran la salida deseada para cada una de estas capas, se denominan ocultas.

Por último, estas redes se denominan neuronales porque se inspiran en la neurociencia, estas arquitecturas fueron diseñadas en primera instancia como una imitación del cerebro humano. Cada capa oculta de la red suele ser una función vectorial. Cada elemento del vector se puede interpretar como una neurona, ya que recibe una serie de entradas o estímulos, los procesa y devuelve un valor en función de dichas entradas, de manera similar a cómo las neuronas procesan la información en nuestro cerebro. Sin embargo, la investigación moderna sobre redes neuronales se guía por muchas disciplinas matemáticas y de ingeniería, y el objetivo de estas no es modelar perfectamente el cerebro. Debemos pensar en las redes neuronales como máquinas de aproximación de funciones diseñadas para lograr la generalización estadística, extrayendo ocasionalmente algunas ideas de lo que sabemos sobre el cerebro, más que como modelos de la función cerebral.

10.1.1. Estructura básica de una red neuronal

La estructura básica de una red neuronal está formada por un conjunto de capas encadenadas, de manera que la salida de una capa es utilizada como entrada para las capas posteriores. Estas capas están formadas por neuronas que calculan funciones que vienen denotadas como $h : \mathbb{R}^n \rightarrow \mathbb{R}$. En primer lugar, vamos a ver la estructura de una neurona y posteriormente veremos cómo se construyen las capas de una red neuronal finalizando con un ejemplo de red neuronal completa donde definiremos la notación de esta.

10.1.1.1. La Neurona

El primer modelo neuronal fue propuesto por McCulloch y Pitts en el año 1943 en términos de un sistema computacional de actividad nerviosa, con el fin de llevar a cabo tareas simples. Este término sirvió de ejemplo para los modelos posteriores que fueron refinando la idea inicial. Una neurona es una función que recibe como entrada un vector $x = (x_1, \dots, x_n)$, realiza una combinación lineal de la entrada dependiente de un vector de pesos $W = (w_1, \dots, w_n)$, a la que se le suma un sesgo o bias que se denota como b o w_0 y posteriormente se emplea una función de activación, denotada por $\theta : \mathbb{R}^n \rightarrow \mathbb{R}$, de dicha combinación lineal para generar la salida. Los parámetros de la neurona los agruparemos y los denotaremos como $W = (b, w_1, \dots, w_n)$ donde hemos incluido el sesgo en el vector de pesos. Vamos a describir este comportamiento gracias a la siguiente figura.

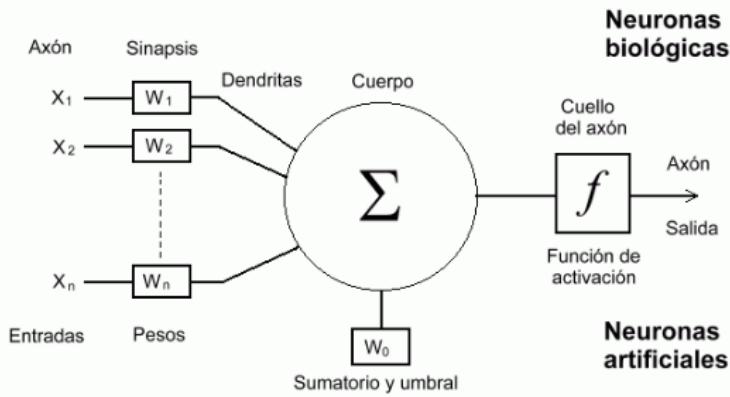


Figura 10.1.: Ejemplo de un modelo neuronal con n entradas. [74]

Las entradas son el estímulo que la neurona artificial recibe del entorno que la rodea, y la salida es la respuesta a tal estímulo. La neurona tiene la capacidad de adaptarse al medio circundante y aprender de él modificando el valor de sus pesos sinápticos, y por ello son conocidos como los parámetros libres del modelo, ya que pueden ser modificados y adaptados a realizar una tarea determinada. Por tanto, una vez visto todo esto, la neurona calcula la siguiente función:

$$h(x) = \theta\left(\sum_{i=1}^n w_i x_i + b\right)$$

En sus inicios, la función de activación propuesta era simplemente un umbral, el resultado de la neurona era

$$h(x) = \begin{cases} 1 & \text{si } \sum_{i=1}^n w_i x_i + b > 0 \\ 0 & \text{si } \sum_{i=1}^n w_i x_i + b \leq 0 \end{cases}$$

Actualmente, las funciones de activación que se utilizan son funciones más complejas que veremos en la sección 10.4, puesto que la función que acabamos de ver carece de algunas propiedades deseables para facilitar su optimización, ya que por ejemplo ni siquiera es

continua.

10.1.1.2. Capas de una red neuronal

Una vez hemos estudiado la unidad básica de procesamiento de una red neuronal vamos a ver una estructura más compleja dentro de las mismas, las capas. Del mismo modo que nuestro cerebro está compuesto por neuronas interconectadas entre sí, una red neuronal artificial está formada por neuronas artificiales conectadas entre sí y agrupadas en diferentes niveles que denominamos capas. Una capa es, simplemente, un conjunto de neuronas cuyas entradas provienen de una capa anterior (o de los datos de entrada en el caso de la primera capa) y cuyas salidas son la entrada de una capa posterior (o la respuesta de la red neuronal). Vamos a ver un ejemplo de una red neuronal.

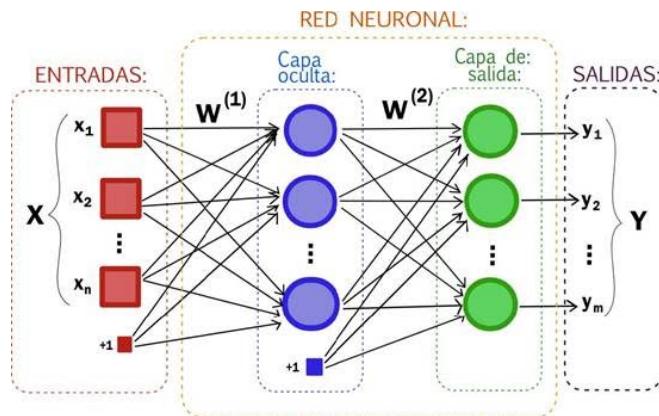


Figura 10.2.: Ejemplo de un esquema de red neuronal con vector de entrada de tamaño n , m neuronas en la capa de salida y una capa oculta. [69]

Existen esencialmente tres tipos de capas en una red neuronal:

- *Capa de entrada.* Esta capa es la primera de la red neuronal, está compuesta por $n + 1$ neuronas donde n es la dimensión del vector de entrada y la neurona restante recibe el nombre de sesgo o bias.
- *Capa de salida.* La capa de salida es la última capa del modelo. Las neuronas de esta capa proporcionan la respuesta visible de la red neuronal y suele ser la capa con menos neuronas.
- *Capas ocultas.* Las capas que se sitúan entre la capa de entrada y la capa de salida se conocen como capas ocultas, ya que desconocemos tanto los valores de entrada como los de salida. Estas capas recogen la entrada de la capa anterior, las neuronas de la capa computan una función, una combinación lineal seguida de una transformación no lineal de la misma, y mandan la salida a la siguiente capa.

10.1.1.3. Red Neuronal

Como hemos dicho previamente, hay muchos tipos de redes neuronales y en este capítulo nos vamos a centrar en las redes neuronales prealimentadas. Estas las podemos entender como

un grafo dirigido acíclico donde los nodos son las neuronas y las aristas son las conexiones entre ellas. Las capas se notarán por $l = 0, 1, 2, \dots, L$, veamos un ejemplo.

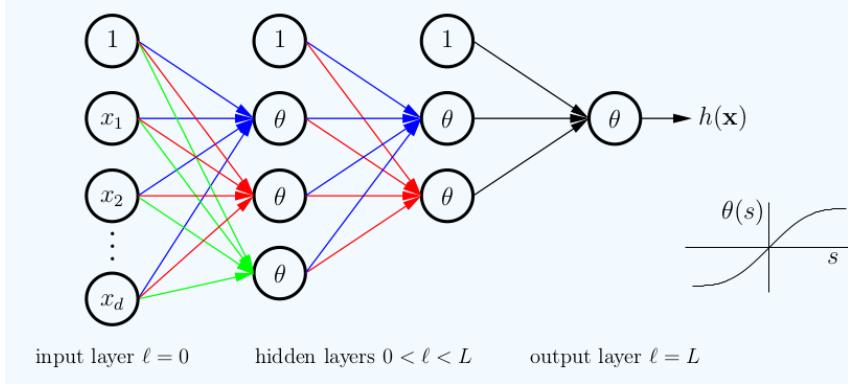


Figura 10.3.: Representación de una red neuronal de 3 capas mediante un grafo. El sesgo viene representado por un 1 y θ representa una función de activación. [1]

La capa de entrada será $l = 0$, esta no se suele considerar una capa, está pensada para alimentar las entradas, y la capa de salida será $l = L$. Usaremos el superíndice (l) para referirnos a una capa, cada capa tiene dimensión $d^{(l)}$ que representa el número de nodos que hay en la capa l . Como vimos en 10.1.1.1 cada capa tiene un nodo especial, el nodo cero, llamado nodo de sesgo o bias que viene representado por b . Estos nodos no tienen peso de entrada pero sí de salida y están configurados para tener una salida constante con valor $x_0 = 1$. Cada flecha representa un peso de conexión desde un nodo en una capa a un nodo de la capa inmediatamente superior. Un nodo con un peso de entrada indica que alguna señal se introduce en este nodo. Cada nodo de este tipo con una entrada tiene una función de activación θ como podemos observar en la figura 10.3.

El modelo de una red neuronal H_{nn} se especifica una vez que se determina la arquitectura de la red, es decir, la dimensión de cada capa. Una hipótesis $h \in H_{nn}$ se especifica seleccionando los pesos de los enlaces. Veamos un nodo de la capa oculta l para ver qué pesos hay que especificar.

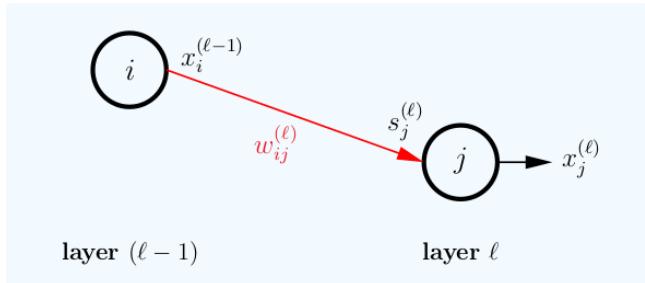


Figura 10.4.: Nodo de una capa oculta. [1]

Cada nodo tiene una señal de entrada s y una salida x . Los pesos de los enlaces hacia el nodo desde la capa anterior vienen denotados por $w^{(l)}$, por lo que los pesos están indexados por la capa a la que van. Así, la salida de los nodos de la capa $l - 1$ se multiplican por los

pesos $w^{(l)}$. Usamos subíndices para indexar los nodos de una capa. Así, $w_{i,j}^{(l)}$ es el peso que va del nodo i de la capa anterior al nodo j de la capa l . La señal que entra en el nodo j de la capa l es $s_j^{(l)}$ y la salida de este nodo es $x_j^{(l)}$.

Hemos visto una imagen de un nodo conectado a otro de la capa anterior pero las redes neuronales están totalmente conectadas, es decir, todos los nodos de una capa se conectan con todos los nodos de la siguiente. Esto se refleja en la siguiente figura.

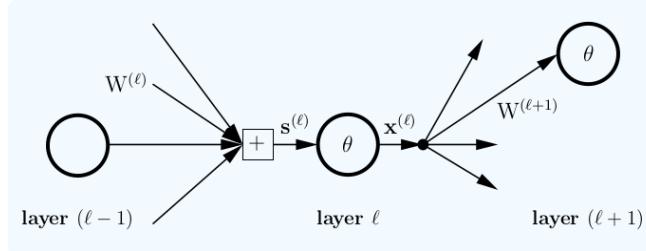


Figura 10.5.: Interconexión de capas ocultas. [1]

En su mayor parte, solo necesitamos tratar la red capa por capa, por lo que introducimos la notación vectorial y matricial para ello. Recogemos todas las señales de entrada a los nodos $1, \dots, d^{(l)}$ en la capa l en el vector $s^{(l)}$. Del mismo modo, recogemos la salida de los nodos $0, \dots, d^{(l)}$ en el vector $x^{(l)}$. Hay enlaces que conectan las salidas de todos los nodos de la capa anterior a las entradas de la capa l . Por lo que la capa l tiene dimensión $d^{(l)}$, recibe como entrada $s^{(l)}$ y la salida que produce es $x^{(l)}$. Los pesos de cada capa los agrupamos en una matriz de pesos, por ejemplo, para la capa l , la matriz de pesos es $W^{(l)}$ y tiene como dimensión $(d^{(l-1)} + 1) \times d^{(l)}$.

Después de fijar los pesos $W^{(l)}$ para $l = 1, \dots, L$ podemos especificar una hipótesis de red neuronal concreta $h \in H_{nn}$. Recogemos todas las matrices de pesos en un único parámetro de pesos $w = (W^{(1)}, W^{(2)}, \dots, W^{(L)})$ y a veces escribiremos $h(x; w)$ para indicar explícitamente la dependencia de la hipótesis de w . Por último, veamos una definición formal de red neuronal prealimentada.

Definición 10.1. (Red Neuronal Prealimentada). Sea $L \in \mathbb{N}$ el número de capas, $d = (d^{(0)}, \dots, d^{(L)}) \in \mathbb{N}^L$ la dimensión de cada capa, $w = (W^{(1)}, \dots, W^{(L)})$ los pesos de la red con $W^{(l)} \in \mathbb{R}^{(d^{(l-1)}+1) \times d^{(l)}}$ y sea $\phi = \{\theta^{(l)} : l \in \{1, \dots, L\}\}$ el conjunto de funciones de activación siendo $\theta^{(l)}$ la función de activación de la capa l . Definimos una red neuronal prealimentada como la cuaterna $[L, d, w, \phi]$ que implementa la función $h(x; w)$ a través de un conjunto finito de pasos de cálculos partiendo de unas condiciones iniciales:

$$\begin{cases} x^{(0)} = x, & x \in \mathbb{R}^{d^{(0)}} \\ x^{(l)} = \theta^{(l)}(s^{(l)}), & l \in \{1, \dots, L\} \end{cases}$$

donde la salida de cada capa viene dada por:

$$s^{(l)} = (W^{(l)})^T x^{(l-1)}, \quad l \in \{1, \dots, L\}$$

10.2. Optimización basada en el Gradiente

La mayoría de los algoritmos de aprendizaje profundo implican algún tipo de optimización. La optimización se refiere a la tarea de minimizar o maximizar alguna función $f(x)$ mediante la alteración de x , esta función que queremos minimizar o maximizar recibe el nombre de función objetivo. Cuando la minimizamos, también podemos llamarla función de coste, de pérdida o función de error. En esta sección, las técnicas de optimización que vamos a ver son técnicas basadas en el gradiente.

Vamos a suponer que nuestra función de coste es una función derivable, $f : \mathbb{R} \rightarrow \mathbb{R}$, de manera que $y = f(x)$, donde tanto x como y son números reales. La derivada de esta función se denomina $f'(x)$ o $\frac{dy}{dx}$. La derivada $f'(x)$ nos da una medida de la pendiente de $f(x)$ en el punto x . En otras palabras, especifica cómo cambia la función f cuando se produce un pequeño cambio en la entrada x : $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

La derivada es, por tanto, útil para minimizar una función, ya que nos permite saber cómo cambiar x para conseguir una pequeña mejora en y . Por ejemplo, sabemos que $f(x - \epsilon \cdot \text{sign}(f'(x)))$ es menor que $f(x)$ para un valor suficientemente pequeño de ϵ . Así, podemos reducir $f(x)$ moviendo x en pequeños pasos con el signo contrario de la derivada. Esta técnica de optimización recibe el nombre de descenso del gradiente. Vamos a ver en la figura 10.6 un ejemplo de esta técnica.

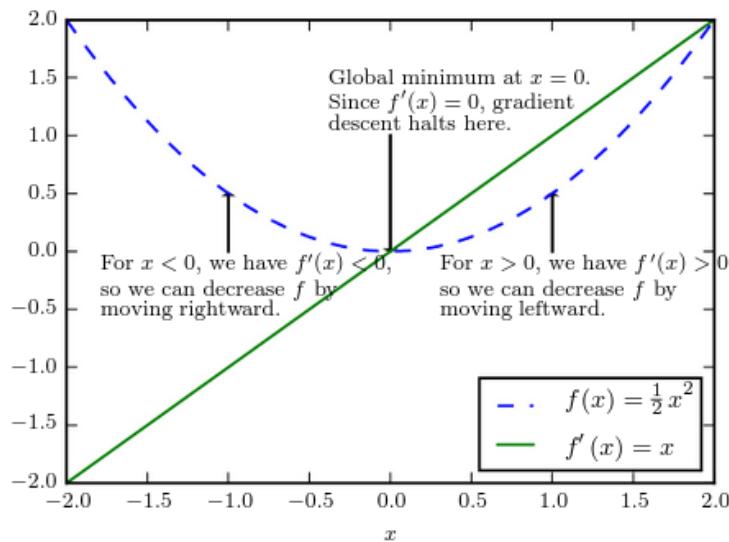


Figura 10.6.: Ejemplo de cómo el algoritmo de descenso del gradiente utiliza las derivadas de una función para optimizarla. [46]

Cuando se cumple que $f'(x) = 0$, es decir, la derivada de la función es 0, esta no proporciona ninguna información sobre en qué dirección debemos mover x . Los puntos donde se verifica que $f'(x) = 0$ se conocen como puntos críticos.

Hay tres tipos de puntos problemáticos:

- *Mínimos locales*: puntos donde $f(x)$ es menor que todos los puntos vecinos, por lo que

ya no es posible disminuir $f(x)$ dando pasos infinitesimales.

- *Máximos locales*: puntos donde $f(x)$ es mayor que cualquier punto cercano, no podemos aumentar el valor de $f(x)$ moviendo x en ninguna dirección
- *Puntos de silla*: puntos que no son ni máximos ni mínimos.

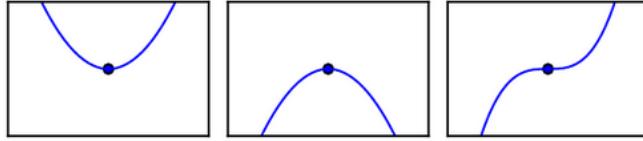


Figura 10.7.: Tipos de puntos críticos. A la izquierda podemos observar una función donde aparece un mínimo local, en el centro una función con un máximo local y a la derecha tenemos un punto de silla. [46]

Un punto en el que se obtiene el menor valor absoluto de $f(x)$ es un mínimo global. En una función puede haber un solo mínimo global o varios mínimos globales. En el contexto del aprendizaje profundo, optimizaremos funciones que pueden tener muchos mínimos locales que no son óptimos, es decir, que no son globales y muchos puntos de sillitas rodeados de regiones planas, que son regiones en las que la derivada de la función con la que trabajamos es muy cercana a 0. Todo esto complica el proceso de optimización, especialmente cuando la entrada de la función es multidimensional. Por tanto, en muchos casos no trataremos de encontrar el mínimo de f en su definición formal, sino que nos conformaremos con encontrar un valor de f que sea muy pequeño. Vamos a ver un ejemplo.

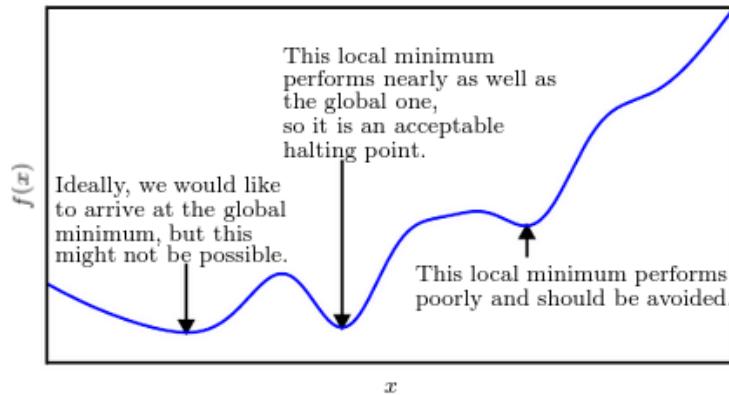


Figura 10.8.: Ejemplo de minimización aproximada. Los algoritmos de optimización pueden no encontrar un mínimo global cuando existen múltiples mínimos locales o mesetas. En el contexto del aprendizaje profundo, generalmente aceptamos esas soluciones aunque no sean realmente mínimas, siempre que correspondan a valores significativamente bajos de la función coste. [46]

Como trabajamos con funciones de varias variables, vamos a extender este concepto a funciones que tienen múltiples entradas, $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Para que el concepto de minimización tenga sentido, debe haber una sola salida (escalar).

Para funciones con múltiples entradas, debemos utilizar el concepto de derivada parcial.

La derivada parcial de una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$ respecto de la variable x_i , la denotaremos como $\frac{\partial}{\partial x_i} f(x)$ y mide cómo cambia f a medida que solo cambia la variable x_i en el punto x . El gradiente generaliza la noción de derivada al caso multidimensional. El gradiente de una función f se define como el vector que contiene las derivadas parciales de la función f respecto de las variables x_i y se denota por $\nabla_x f(x)$. En múltiples dimensiones, los puntos críticos son puntos en los que cada elemento del gradiente es igual a cero, es decir, $\nabla_x f(x)$ es el vector 0.

La derivada direccional en la dirección del vector u , siendo u un vector unitario, es la pendiente de la función f en la dirección de u . Esta derivada se puede calcular como la derivada de la función $f(x + \alpha u)$ con respecto a α , evaluada en $\alpha = 0$. Usando la regla de la cadena se demuestra que dicha derivada es $\frac{\partial}{\partial \alpha} f(x + \alpha u) = u^T \nabla_x f(x)$ cuando $\alpha = 0$.

Para minimizar f , trataremos de encontrar la dirección en la que f disminuye más rápidamente. Podemos hacerlo utilizando la derivada direccional:

$$\min_{u, u^T u=1} u^T \nabla_x f(x) = \min_{u, u^T u=1} \|u\|_2 \|\nabla_x f(x)\|_2 \cos \theta$$

donde θ es el ángulo entre u y el gradiente. Sustituyendo $\|u\|_2 = 1$ e ignorando los factores que no dependen de u , esto se simplifica a $\min_u \cos \theta$. Esto se minimiza cuando u apunta en la dirección opuesta al gradiente. Por tanto, para minimizar f tendremos que movernos en la dirección contraria a la que marca el gradiente. Esto se conoce como gradiente descendente o descenso del gradiente [46].

10.2.1. El Gradiente Descendente y sus variantes

El gradiente descendente fue un método propuesto en 1847 por Cauchy [12] y 100 años más tarde Haskell B. Curry [20] demostró su convergencia.

La intuición geométrica que subyace detrás de esta técnica que acabamos de explicar es la siguiente. Veamos la función que queremos minimizar como una superficie en el espacio. Imaginemos que este espacio en el que nos encontramos en un sistema montañoso y que la misión es llegar a la zona más baja de toda la superficie. Sin embargo, no tenemos acceso al mapa del terreno y ni siquiera podemos observar a nuestro alrededor, vamos con los ojos cerrados. Ante esta situación, lo único que podríamos hacer es tantear con el pie el sitio por el que nos encontramos y como lo que queremos es descender al punto mínimo entonces nos moveremos en la dirección donde la pendiente descienda con mayor intensidad. Por tanto, primero evaluamos la inclinación para encontrar la mayor pendiente, luego caminamos una distancia en esa dirección y nos paramos. En la nueva posición volvemos a repetir el proceso y así iterativamente. Esta es la lógica del algoritmo del descenso del gradiente. Durante este proceso iremos ajustando los parámetros de forma iterativa para minimizar la función de coste. El objetivo es descender desde la superficie hasta un punto donde, nos movamos en la dirección en la que nos movamos, no consigamos descender más.

Como hemos visto, el gradiente descendente es una forma de minimizar una función objetivo $J(w)$ parametrizada por los parámetros de un modelo w actualizando los parámetros en la dirección opuesta al gradiente de la función objetivo $\nabla_w J(w)$ con respecto a los parámetros. Normalmente, la función de coste se puede escribir como una media sobre el conjunto de datos de entrenamiento, como por ejemplo:

$$J(w) = \mathbb{E}_{(x,y) \sim \hat{p}_{data}} L(h(x; w), y)$$

donde L es la función de pérdida asociada a cada ejemplo, $h(x; w)$ es la salida predicha cuando se tiene x como entrada, \hat{p} es la distribución empírica del conjunto de datos. Dicho de otro modo, esta función es la esperanza del error cometido sobre el conjunto de datos cuando para una entrada x se predice como salida $h(x; w)$ cuando el verdadero valor a predecir es y .

El algoritmo del gradiente descendente emplea un factor de proporcionalidad que determina el tamaño de los pasos que damos para alcanzar un mínimo (local), se denota por η y es conocido como tasa de aprendizaje o *learning rate*. Por tanto, el algoritmo consiste en partir de unas condiciones iniciales y de manera iterativa calcular el vector opuesto del gradiente y avanzar en esa dirección una determinada distancia marcada por la tasa de aprendizaje hasta llegar a un valle.

Existen tres variantes del algoritmo del gradiente descendente, que difieren en la cantidad de datos que utilizamos para calcular el gradiente de la función objetivo. Para realizar esta subsección y la siguiente nos hemos basado en [72].

10.2.1.1. Gradiente Descendente por Lotes

El descenso del gradiente por lotes en inglés *batch gradient descent*, calcula el gradiente de la función de coste con respecto a los parámetros w para todo el conjunto de datos de entrenamiento:

$$w = w - \eta \cdot \nabla_w J(w)$$

Al tener que calcular los gradientes para todo el conjunto de datos para realizar una sola actualización, el descenso del gradiente por lotes puede ser muy lento y es intratable para conjuntos de datos que no caben en la memoria. El descenso del gradiente por lotes tampoco nos permite actualizar nuestro modelo en línea, es decir, con nuevos ejemplos sobre la marcha.

Para un número predefinido de épocas, primero se calcula el vector gradiente de la función de pérdida para todo el conjunto de datos con respecto al vector de parámetros. A continuación, se actualizan los parámetros en la dirección de los gradientes con la tasa de aprendizaje que determina la magnitud de la actualización que se realiza. Se garantiza que el descenso del gradiente por lotes converge al mínimo global para superficies de error convexas y a un mínimo local para superficies no convexas.

10.2.1.2. Gradiente Descendente Estocástico

El descenso del gradiente estocástico (SGD) realiza una actualización de los parámetros para cada ejemplo de entrenamiento $x^{(i)}$ y etiqueta $y^{(i)}$:

$$w = w - \eta \cdot \nabla_w J(w; x^{(i)}; y^{(i)})$$

El descenso del gradiente por lotes realiza cálculos redundantes para grandes conjuntos de datos, ya que vuelve a calcular los gradientes de ejemplos similares antes de cada actualización de los parámetros. El SGD elimina esta redundancia al realizar una actualización cada vez. Por tanto, suele ser mucho más rápido y también puede utilizarse para aprender en línea. El SGD realiza actualizaciones frecuentes con una alta varianza que hace que la función objetivo fluctúe fuertemente.

Mientras que el descenso del gradiente por lotes converge al mínimo de la cuenca en la que se encuentran los parámetros, en el SGD estas fluctuaciones que acabamos de comentar le permite saltar a nuevos y potencialmente mejores mínimos locales. Por otra parte, esto complica en última instancia la convergencia al mínimo exacto, ya que el SGD seguirá sobrepasando límites. Sin embargo, se ha demostrado que si disminuimos lentamente la tasa de aprendizaje, el descenso del gradiente estocástico muestra el mismo comportamiento que el descenso del gradiente por lotes, convergiendo casi con seguridad a un mínimo local o global para la optimización no convexa y convexa respectivamente.

10.2.1.3. Gradiente Descendente por Mini-lotes

El descenso del gradiente por mini-lotes es una mezcla del descenso del gradiente por lotes y estocástico. El conjunto de entrenamiento se divide en varios grupos llamados mini-lotes y realiza una actualización para cada uno de ellos de n ejemplos de entrenamiento:

$$w = w - \eta \cdot \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)})$$

Así, reduce la varianza de las actualizaciones de los parámetros, lo que puede conducir a una convergencia más estable. Puede hacer uso de las optimizaciones matriciales altamente optimizadas que se emplean en bibliotecas comunes de aprendizaje automático que hacen que el cálculo del gradiente con un mini-lote sea muy eficiente. Los tamaños comunes de los mini-lotes oscilan entre 50 y 256, pero pueden variar para diferentes aplicaciones. El descenso del gradiente por mini-lotes suele ser el algoritmo elegido cuando se entrena una red neuronal y el término SGD suele emplearse también cuando se utilizan mini-lotes.

10.2.2. Algoritmos de optimización del Gradiente Descendente

Sin embargo, el descenso del gradiente por mini-lotes no garantiza buena convergencia, sino que ofrece algunos desafíos que deben ser abordados:

- Elegir una tasa de aprendizaje adecuada puede ser difícil. Una tasa de aprendizaje demasiado pequeña conduce a una convergencia lenta, mientras que una tasa de aprendizaje demasiado grande puede dificultar la convergencia y hacer que la función de pérdida fluctúe en torno al mínimo o incluso diverja.
- Los programas de tasa de aprendizaje intentan ajustar la tasa de aprendizaje durante el entrenamiento, por ejemplo, reduciendo la tasa de aprendizaje según un programa predefinido o cuando el cambio en el objetivo entre épocas cae por debajo de un umbral. Estos programas y umbrales deben definirse de antemano, luego no pueden adaptarse a las características de un conjunto de datos.

- Además, la misma tasa de aprendizaje se aplica a todas las actualizaciones de los parámetros. Si tenemos un número escaso de datos y nuestras características tienen frecuencias muy diferentes, es posible que no queramos actualizarlas todas en la misma medida, sino que realicemos una actualización mayor para las características que raramente aparecen.
- Otro desafío clave de la minimización de las funciones de error altamente no convexas comunes para las redes neuronales es evitar quedar atrapado en sus numerosos mínimos locales subóptimos. En [26] argumentan que la dificultad no surge del hecho de los mínimos locales, sino de los puntos de silla. Estos puntos suelen estar rodeados por una meseta del mismo error, lo que hace que sea notoriamente difícil para el SGD escapar, ya que el gradiente es cercano a cero en todas las dimensiones.

Vamos a nombrar algunos de los algoritmos que son ampliamente utilizados por la comunidad de aprendizaje profundo para hacer frente a los desafíos que acabamos de comentar. Estos algoritmos son: SGD con Momento, Gradiente acelerado de Nesterov, Adagrad, Ada-delta, RMSProp, Adam, el cual hemos utilizado en nuestro modelo, AdaMax y Nadam.

10.3. Entrenamiento de Redes Neuronales

Para esta parte del capítulo vamos a explicar (con la ayuda de [1], [46] y de [75]) cómo entrena n las rededs neuronales. Cuando utilizamos una red neuronal que acepta una entrada x y produce una salida \hat{y} la información fluye hacia adelante a través de la red. La entrada x proporciona la información inicial que luego se propaga a las unidades ocultas de cada capa y finalmente produce $\hat{y} = h(x; w)$. Esto se conoce como propagación hacia adelante o *forward propagation*. Durante el entrenamiento, este proceso tendrá una función de coste asociada $J(w)$ que dependerá esencialmente de los pesos. El objetivo es calcular los pesos óptimos para minimizar dicha función. El algoritmo de *backpropagation* permite que la información de la función de coste fluya hacia atrás a través de la red para calcular el gradiente. Esta técnica se inventó en el año 1986 y produjo un fuerte cambio en el devenir de las redes neuronales.

Hasta ahora, hemos visto cómo podemos optimizar la función de coste utilizando la técnica del gradiente descendente. El principal inconveniente de esta técnica es que necesitamos calcular el gradiente de la función de coste o pérdida en todos los pasos del algoritmo. El cálculo de una expresión analítica para el gradiente es sencillo pero la evaluación numérica de dicha expresión puede ser costosa desde el punto de vista computacional. Debido a esto surgió el algoritmo de backpropagation, ante la necesidad de calcular el gradiente de manera eficiente. El algoritmo de backpropagation se empleará en el algoritmo del gradiente descendente, ya que nos permite calcular las derivadas parciales de la función de pérdida respecto de los parámetros de la red de forma sencilla, permitiendo una mejoría de la eficiencia en el cálculo del gradiente.

Vamos a comentar algunas consideraciones a tener en cuenta durante el entrenamiento de una red neuronal. Para la aplicación del algoritmo del gradiente descendente es una práctica común inicializar los pesos a valores aleatorios pequeños dados por una distribución normal. Debemos tener en cuenta un criterio que es de vital importancia para no aumentar de manera excesiva la complejidad en tiempo de entrenamiento, el criterio de parada. Hay muchos criterios como imponer un número máximo de iteraciones, tamaño del gradiente,

valor alcanzado en el error, etc. Lo recomendable es emplear una combinación de ambas.

El término backpropagation suele entenderse erróneamente como todo el algoritmo de aprendizaje de las redes neuronales multicapa. En realidad, el algoritmo de backpropagation solo se refiere al método para calcular el gradiente, mientras que otro algoritmo, como el descenso del gradiente, se utiliza para realizar el aprendizaje utilizando este gradiente. Además, otro concepto erróneo es pensar que backpropagation es algo específico de las redes neuronales multicapa, pero en principio puede calcular las derivadas de cualquier función. En particular, calcula el gradiente $\nabla_x f(x, y)$ para una función arbitraria f , donde x es un conjunto de variables cuyas derivadas queremos conocer, e y es un conjunto adicional de variables de las cuales depende la función pero cuyas derivadas no se requieren. En los algoritmos de aprendizaje, el gradiente que requerimos más a menudo es el gradiente de la función de coste con respecto a los parámetros, $\nabla_w J(w)$. Muchas tareas de aprendizaje automático requieren el cálculo de otras derivadas, ya sea como parte del proceso de aprendizaje o para analizar el modelo aprendido, el algoritmo de backpropagation también se puede aplicar a estas tareas. La idea de calcular las derivadas propagando la información a través de una red es muy general y se puede generalizar para múltiples salidas.

El algoritmo de backpropagation consiste en aplicar la regla de la cadena en cada nodo del grafo computacional para calcular el gradiente de manera altamente eficiente. La eficiencia se logra gracias a que no se repiten cálculos ya realizados y a una reordenación de los mismos.

10.3.1. Forward Propagation

La hipótesis de la red neuronal $h(x; w)$ se calcula mediante el algoritmo de propagación hacia delante más conocido como *forward propagation*. En primer lugar, observemos que las entradas y salidas de una capa están relacionadas por la función de activación,

$$x^{(l)} = \begin{bmatrix} 1 \\ \theta(s^{(l)}) \end{bmatrix}$$

donde $\theta(s^{(l)})$ es un vector cuyos componentes son $\theta(s_j^{(l)})$. Para obtener el vector de entrada de la capa l , calculamos la suma ponderada de las salidas de la capa anterior, con pesos especificados por $W^{(l)}$: $s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$. Este proceso se representa de forma compacta mediante la ecuación matricial:

$$s^{(l)} = (W^{(l)})^T x^{(l-1)}$$

Todo lo que queda es inicializar la capa de entrada a $x^{(0)}$ por lo que $d^{(0)} = d$ siendo esta la dimensión de la entrada y usamos las anteriores ecuaciones en la siguiente cadena,

$$x = x^{(0)} \xrightarrow{W^{(1)}} s^{(1)} \xrightarrow{\theta} x^{(1)} \xrightarrow{W^{(2)}} s^{(2)} \xrightarrow{\theta} x^{(2)} \dots \xrightarrow{s^{(L)}} x^{(L)} = h(x; w)$$

Después de realizar la propagación hacia delante, la salida del vector $x^{(l)}$ de todas las capas $l = 0, \dots, L$ han sido calculadas. Ahora tenemos que minimizar nuestra función de coste $J(w)$, para ello aplicaremos el gradiente descendente con un algoritmo especial que calcula el

gradiente de manera eficiente, el algoritmo de backpropagation.

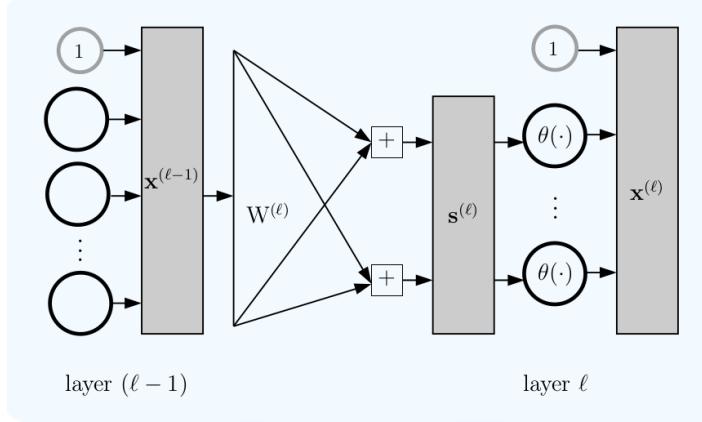


Figura 10.9.: Representación esquemática del algoritmo de forward propagation. [1]

10.3.2. Algoritmo de Backpropagation

El algoritmo de backpropagation nos permite calcular las derivadas parciales con respecto a cada peso de manera eficiente, utilizando un cálculo $O(Q)$, donde Q es el número de pesos. El algoritmo de backpropagation se basa en varias aplicaciones de la regla de la cadena para escribir las derivadas parciales de la capa l utilizando las derivadas parciales de la capa $l+1$. Para describir el algoritmo, definimos el vector de sensibilidad de la capa l , que es la sensibilidad (gradiente) del error $J = J(w)$ con respecto a la señal de entrada $s^{(l)}$ que entra en la capa l , es decir, el error imputado a la capa l . Denotamos la sensibilidad por $\delta^{(l)}$,

$$\delta^{(l)} = \frac{\partial J}{\partial s^{(l)}}$$

La sensibilidad cuantifica cómo cambia J con respecto a $s^{(l)}$. Utilizando la sensibilidad podemos escribir las derivadas parciales con respecto a los pesos $W^{(l)}$ como:

$$\frac{\partial J}{\partial W^{(l)}} = x^{(l-1)} (\delta^{(l)})^T \quad (10.1)$$

Derivaremos esta fórmula más adelante, pero por ahora vamos a examinarla. Las derivadas parciales de la izquierda forman una matriz de dimensiones $(d^{(l-1)} + 1) \times d^{(l)}$ y el producto de los dos vectores de la derecha de la igualdad da exactamente esa matriz. Las derivadas parciales tienen contribuciones de dos componentes:

1. El vector de salida de la capa de la que proceden los pesos; cuanto mayor sea la salida, más sensible será el error J a los pesos de la capa.
2. El vector de sensibilidad de la capa a la que van los pesos; cuanto mayor sea el vector de sensibilidad, más sensible será el error J a los pesos de esa capa.

Las salidas $x^{(l)}$ para cada capa $l \geq 0$ pueden calcularse como hemos visto, mediante una

propagación hacia adelante. Así que para obtener las derivadas parciales, basta con obtener los vectores de sensibilidad $\delta^{(l)}$ para cada capa $l \geq 1$ (recordemos que no hay señal de entrada a la capa $l = 0$). Los vectores de sensibilidad se pueden obtener ejecutando una versión ligeramente modificada de la red neuronal hacia atrás, y de ahí el nombre de backpropagation. La idea básica del algoritmo se ilustra en la siguiente figura.

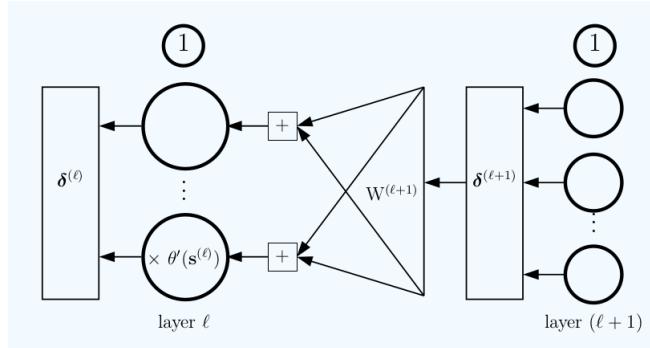


Figura 10.10.: Representación esquemática del algoritmo de backpropagation. [1]

Como se puede ver en la figura, la red neuronal está ligeramente modificada, hemos cambiado la función de activación de los nodos. Ahora tenemos que la función de activación es $\theta'(s^{(l)})$, donde $s^{(l)}$ es la entrada al nodo. Así que ahora la función de activación es diferente para cada nodo y depende de la entrada al nodo x .

En la figura podemos observar que la capa $l + 1$ emite (hacia atrás) el vector de sensibilidad $\delta^{(l+1)}$, que se multiplica por los pesos $W^{(l+1)}$, sumados y pasados a los nodos de la capa l . Los nodos de la capa l se multiplican por $\theta'(s^{(l)})$ para obtener $\delta^{(l)}$. Denotando la multiplicación por componentes mediante \otimes . Una notación abreviada para este paso de retropopagación es:

$$\delta^{(l)} = \theta'(s^{(l)}) \otimes [W^{(l+1)} \delta^{(l+1)}]_1^{d^{(l)}} \quad (10.2)$$

donde el vector $[W^{(l+1)} \delta^{(l+1)}]_1^{d^{(l)}}$ contiene los componentes $1, \dots, d^{(l)}$ del vector $W^{(l+1)} \delta^{(l+1)}$ donde se excluye el componente de sesgo que tiene índice 0. La sensibilidad del error a las entradas de la capa l es proporcional a la pendiente de la función de activación en la capa l (una mayor pendiente significa que un pequeño cambio en $s^{(l)}$ tendrá un mayor efecto en $x^{(l)}$), al tamaño de los pesos que salen de la capa (pesos más grandes significan que un pequeño cambio en $s^{(l)}$ tendrá más impacto en $s^{(l+1)}$) y a la sensibilidad en la siguiente capa (un cambio en la capa l afecta a las entradas de la capa $l + 1$, por lo que si el error J es más sensible a la capa $l + 1$ entonces también será más sensible a la capa l).

Utilizando la propagación hacia adelante, calculamos la salida de cada capa $x^{(l)}$ para $l = 0, \dots, L$ usando la salida de la capa anterior $x^{(l-1)}$ y mediante backpropagation calculamos todas las sensibilidades $\delta^{(l)}$ para $l = 1, \dots, L$ recorriendo el grafo de la red hacia atrás. Esta idea se muestra en el siguiente grafo:

$$\delta^{(1)} \leftarrow \delta^{(2)} \dots \leftarrow \delta^{(L-1)} \leftarrow \delta^{(L)}$$

Por último, obtenemos la derivada parcial del error en un solo punto de datos utilizando

la ecuación 10.1. Notar que si hay un solo nodo de salida tenemos que $s^{(L)}$ es un escalar y también lo es $\delta^{(L)}$.

Vamos a realizar un resumen desde un punto de vista algo más matemático. Para empezar vamos a profundizar en la derivada parcial, $\frac{\partial J}{\partial W^{(l)}}$. La situación se ilustra en la figura 10.11.

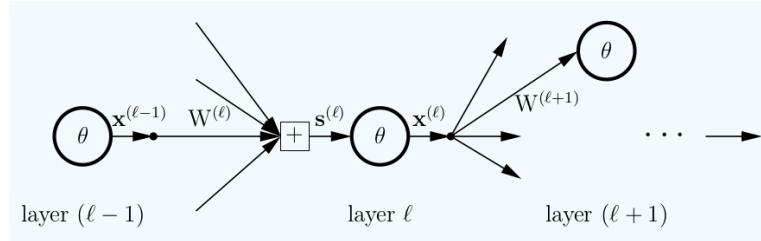


Figura 10.11.: Cadena de dependencias desde $W^{(l)}$ a $x^{(l)}$. Obtenemos como salida la función de coste o error J . [1]

Podemos identificar la siguiente cadena de dependencias por la que $W^{(l)}$ influye en la salida de $x^{(L)}$, y por tanto, en el error de J .

$$W^{(l)} \rightarrow s^{(l)} \rightarrow x^{(l)} \rightarrow s^{(l+1)} \dots \rightarrow x^{(L)} = h$$

Para derivar 10.1 nos limitamos a un solo peso y utilizamos la regla de la cadena. Para un solo peso $w_{ij}^{(l)}$, un cambio en $w_{ij}^{(l)}$ solo afecta a $s_j^{(l)}$ y así por la regla de la cadena tenemos que,

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} \cdot \frac{\partial J}{\partial s_j^{(l)}} = x_i^{(l-1)} \cdot \delta_j^{(l)}$$

donde la última igualdad se deduce porque $s_j^{(l)} = \sum_{\alpha=0}^{d^{(l-1)}} w_{\alpha j}^{(l)} x_{\alpha}^{(l-1)}$ y por la definición de $\delta_j^{(l)}$. Hemos derivado la forma componente de 10.1.

Derivemos ahora la forma componente que aparece en 10.2. Como J depende de $s^{(l)}$ solo a través de $x^{(l)}$, por la regla de la cadena, tenemos

$$\delta_j^{(l)} = \frac{\partial J}{\partial s_j^{(l)}} = \frac{\partial J}{\partial x_j^{(l)}} \cdot \frac{\partial x_j^{(l)}}{\partial s_j^{(l)}} = \theta'(s_j^{(l)}) \cdot \frac{\partial J}{\partial x_j^{(l)}}$$

Este es el error imputado a la neurona j de la capa l . Para obtener la derivada parcial $\frac{\partial J}{\partial x^{(l)}}$ necesitamos extender cómo cambia J debido a los cambios en $x^{(l)}$. De nuevo, a partir de la figura 10.11, vemos que un cambio en $x^{(l)}$ solo afecta a $s^{(l+1)}$ y, por tanto, a J . Como un componente concreto de $x^{(l)}$ puede afectar a todos los componentes de $s^{(l+1)}$, tenemos que sumar estas dependencias utilizando la regla de la cadena:

$$\frac{\partial J}{\partial x_j^{(l)}} = \sum_{k=1}^{d^{(l+1)}} \frac{\partial s_k^{(l+1)}}{\partial x_j^{(l)}} \cdot \frac{\partial J}{\partial s_k^{(l+1)}} = \sum_{k=1}^{d^{(l+1)}} w_{jk}^{(l+1)} \delta_k^{(l+1)}$$

Si juntamos todo esto obtenemos la versión de componentes de [10.2](#).

$$\delta_j^{(l)} = \theta'(s_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{jk}^{(l+1)} \delta_k^{(l+1)}$$

De manera intuitiva, el primer término proviene del efecto de $s^{(l)}$ sobre $x^{(l)}$; la sumatoria es el efecto que produce $x^{(l)}$ sobre $s^{(l+1)}$ y el efecto de $s^{(l+1)}$ sobre h es lo que nos da de vuelta las sensibilidades en la capa $l + 1$, dando lugar a la propagación hacia atrás.

10.4. Funciones de Activación

Para esta parte del capítulo seguiremos principalmente [\[25\]](#) y [\[77\]](#). Una componente clave de las redes neuronales es la función de activación. Una función de activación es una función que transmite la información generada por la combinación lineal de los pesos y las entradas que emplean las neuronas en cada capa, es decir, es la manera de transmitir la información por las conexiones de salida. La información puede transmitirse sin modificaciones, para ello emplearíamos la función identidad, mediante funciones de activación lineales o empleando funciones no lineales. Las funciones de activación suelen ser funciones no lineales permitiendo aportar una mayor variedad y mejores aproximaciones al modelo como veremos a continuación.

El diseño de las funciones de activación es un área de investigación muy activo y todavía no existen una serie de criterios para saber cuál es la más adecuada en un determinado caso, la decisión se suele tomar realizando ensayo y error. Vamos a presentar algunas de las funciones de activación más empleadas junto con sus ventajas y desventajas pero primero mencionaremos una serie de propiedades que esperamos que tengan las funciones de activación.

10.4.1. Propiedades de las funciones de activación

En general, las propiedades que se esperan que tengan las funciones de activación son:

- *No lineales.* Hay dos razones por las que una función de activación debe ser no lineal.
 - Los límites en los problemas del mundo real no siempre son lineales. Una función no lineal puede aproximarse fácilmente a un límite lineal pero una función lineal no puede aproximarse a un límite no lineal. Puesto que, una red neuronal aprende el patrón o límite a partir de unos datos, la no linealidad de la función de activación es necesaria para que la red neuronal artificial pueda aprender fácilmente cualquier límite ya sea lineal o no lineal.
 - Las funciones de activación lineales no tienen mucho sentido, ya que cuando conectamos varias capas de neuronas con este tipo de funciones de activación

obtenemos una nueva función lineal, por lo que toda la red equivaldría a tener una sola capa de neuronas lineales y no tendría sentido disponer de una red neuronal con muchas capas ocultas, la profundidad de la red no tendría ningún efecto.

Luego, la no linealidad en la función de activación es necesaria cuando el límite de decisión es de naturaleza no lineal.

- *Diferenciables.* Durante el empleo del algoritmo de backpropagation, el gradiente de la función de pérdida se calcula mediante el método del descenso del gradiente y para esto es necesario que la función sea diferenciable respecto a su entrada.
- *Continuas.* La diferenciabilidad es una propiedad necesaria de las funciones de activación y para que una función sea diferenciable es necesario que sea continua.
- *Acotadas.* Los datos de entrada pasan por una serie de perceptrones, cada uno de los cuales con su función de activación. Como resultado de esto, si la función no está acotada en un rango puede explotar como explicaremos algo más en profundidad posteriormente. Para controlar esto, es importante, pero no necesario que la función esté acotada.
- *Centradas en cero.* Una función está centrada en cero cuando su rango contiene tanto valores positivos como negativos. Si la función de activación de la red no está centrada en cero significa que es siempre positiva o negativa. Por tanto, la salida de una capa siempre se desplaza hacia los valores positivos o negativos y esto implica que el vector de pesos requiere más actualizaciones para ser entrenado. Como resultado tenemos que el número de épocas necesarias para que la red se entrene aumenta. Por todo esto, esta propiedad es importante pero no necesaria.
- *Coste computacional.* El coste computacional de una función de activación se define como el tiempo necesario para generar la salida de la función de activación cuando se le introduce una entrada. También es importante el coste computacional del gradiente, ya que el gradiente se calcula durante la actualización de los pesos mediante la técnica de backpropagation.

10.4.2. Función de activación sigmoidal

Las funciones sigmoidales son no lineales, estrictamente crecientes, continuas y derivables, lo que las hace especialmente útiles en redes neuronales que se entrena usando backpropagation.

Existen diferentes funciones sigmoidales, todas tienen forma de S. Algunas de ellas son muy empleadas en las redes neuronales artificiales, ya que tienen propiedades matemáticas que las hacen muy interesantes. Si existe una relación sencilla entre el valor de la función en un punto y el valor de su derivada en ese mismo punto, podemos reducir el coste computacional del entrenamiento de la red. Teniendo el valor de activación de la neurona podemos obtener su derivada de manera eficiente con una sencilla expresión aritmética.

Vamos a ver algunos ejemplos de funciones sigmoidales, la función logística, la función tangente hiperbólica y la función exponencial normalizada o softmax que es una generalización de la función logística.

10.4.2.1. Función logística

La función logística es una función continua, diferenciable, tiene como dominio la recta real y está acotada, toma valores en el intervalo $[0, 1]$. La forma matemática de la función logística es:

$$\sigma : \mathbb{R} \rightarrow [0, 1], \sigma(x) = \frac{1}{1 + e^{-x}}$$

Esta función es simétrica,

$$\sigma(-x) = 1 - \sigma(x)$$

La función contiene un término exponencial y como las funciones exponenciales tienen un alto coste computacional esta también lo tiene. A pesar de que tiene un coste computacional elevado, su gradiente no lo es. Su gradiente se puede calcular como

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

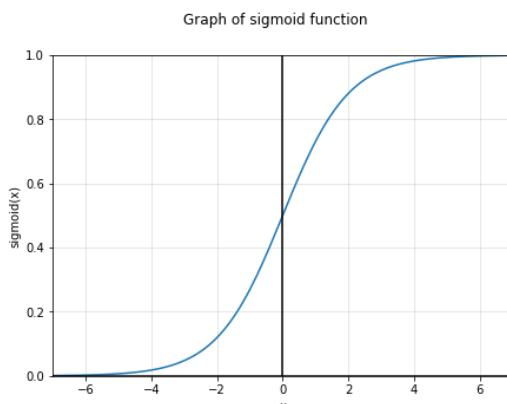


Figura 10.12.: Gráfica de la función sigmoide logística. [25]

La función logística también tiene algunos inconvenientes importantes. Está limitada en el rango $[0, 1]$. Por lo tanto, siempre produce un valor no negativo en la salida, luego no es una función de activación centrada en cero y esto puede originar un comportamiento inadecuado en las actualizaciones del gradiente de los pesos. Un gran cambio en el valor de entrada conduce a un pequeño cambio en el valor de salida dando lugar a valores de gradiente pequeños también. Debido a los pequeños valores del gradiente sufre el problema del desvanecimiento del gradiente que pasamos a explicar de manera breve. Cuando el valor de la variable x es muy positivo, la neurona satura el valor a 1, y cuando es muy negativo, satura a 0, lo que implica que el gradiente de la función de activación sea casi nulo. Durante backpropagation, este gradiente local se utiliza para calcular el gradiente final de la función objetivo. Si un valor cercano a cero se multiplica varias veces por otros valores casi nulos, el valor del gradiente se acerca a cero a medida que se propaga hacia atrás en la red. Por tanto,

las actualizaciones en los pesos serán casi insignificantes y el valor de la función objetivo dejará de disminuir, haciendo que la red no entrene correctamente. Como consecuencia, hay que tener cuidado al inicializar los pesos, ya que si son demasiado grandes la mayoría de las neuronas se saturarán. Por esto, en la actualidad se desaconseja el uso de las funciones sigmoides en las capas ocultas de las redes neuronales prealimentadas.

Esta propiedad es deseable pero no necesaria, por eso se pueden utilizar este tipo de funciones en la capa de salida cuando se elige una función de pérdida adecuada permitiendo mitigar la saturación causada por la sigmoide.

Para tener las ventajas de la función sigmoide junto con la propiedad de que se encuentre centrada en cero, se introdujo la función tangente hiperbólica.

10.4.2.2. Función tangente hiperbólica

La función tangente hiperbólica se emplea en trigonometría esférica, es una función continua, diferenciable, tiene como dominio la recta real y está acotada, toma valores en el intervalo $[-1, 1]$. Tiene como expresión:

$$\tanh : \mathbb{R} \rightarrow [-1, 1], \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

También puede verse como una modificación de la función logística,

$$\tanh(x) = 2\sigma(2x) - 1$$

por lo que presenta los mismos beneficios que la función sigmoide logística junto con la propiedad de que se encuentra centrada en el cero. Es una función impar, $\tanh(-x) = -\tanh(x)$. Su derivada viene dada por:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Al igual que la función logística, tiene un alto coste computacional y presenta también el inconveniente del desvanecimiento del gradiente pero se encuentra centrada en cero, por lo que, aunque la salida de esta función se sature no tendremos el problema que comentamos con la función logística.

10.4.2.3. Función softmax

La función softmax es una combinación de múltiples funciones sigmoides que se suele utilizar en la capa de salida para problemas de clasificación multiclas donde las clases son disjuntas. Representa la probabilidad de que el vector de entrada pertenezca a cada una de las diferentes clases del problema. Se puede expresar como:

$$\text{softmax} : \mathbb{R}^K \rightarrow [0, 1]^K, \text{softmax}(x)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}, \text{ siendo } j = 1, 2, \dots, K$$

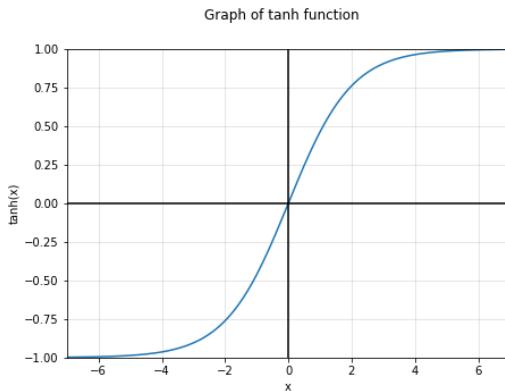


Figura 10.13.: Gráfica de la función tangente hiperbólica [25]

siendo K el número de clases disjuntas. El denominador garantiza que las salidas sumen 1, por lo que la salida de la red dada por la función softmax puede interpretarse como una distribución de probabilidad definida sobre una variable aleatoria discreta con K valores posibles.

Las derivadas parciales de la función softmax son las siguientes:

$$\frac{\partial \text{softmax}(x)_j}{\partial x_i} = \text{softmax}(x)_j (\delta_{ij} - \text{softmax}(x)_i)$$

donde

$$\delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

es la función delta de Kronecker.

La función softmax es una generalización de la función sigmoide logística a múltiples clases y presenta los mismos problemas que la función sigmoide. Cuando constuimos un modelo para la clasificación de múltiples clases, la capa de salida de la red tendrá el mismo número de neuronas que el número de clases que estamos evaluando.

10.4.3. Función ReLU

La unidad lineal rectificada conocida como ReLU es una función que se define de la siguiente manera:

$$\text{ReLU} : \mathbb{R} \rightarrow \mathbb{R}, \text{ReLU}(x) = \max(0, x)$$

La función ReLU es continua, no acotada y no se encuentra centrada en cero. Esta función es diferenciable en todos los puntos menos en $x = 0$. Esto podría parecer un inconveniente para su uso con el algoritmo de descenso del gradiente pero no lo es tanto, ya que en la práctica

es muy poco probable que la entrada tome el valor exacto 0 y si fuera el caso se devolvería una de las derivadas laterales. Su derivada viene dada por:

$$\frac{dReLU(x)}{dx} = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x < 0 \end{cases}$$

Las funciones ReLU son similares a las funciones lineales, siendo lineales en la mitad de su dominio, permitiendo que sean fáciles de optimizar. Además, su derivada es grande siempre que la entrada sea positiva. Como consecuencia permite que aceleren en gran medida la convergencia del descenso del gradiente. Su coste computacional es mucho menor que el de las funciones sigmoides, puesto que simplemente transforman los valores negativos en cero.

Al no estar acotadas puede ocurrir que los gradientes exploten si los pesos tienen valores grandes y nos encontramos frente al problema de la explosión del gradiente. Este problema puede verse como la inversa del desvanecimiento del gradiente y ocurre cuando se acumulan grandes derivadas, el gradiente aumenta exponencialmente y provoca grandes actualizaciones de los pesos. Esto crea un modelo muy inestable e incapaz de aprender y puede acabar con errores de desbordamiento debido a pesos extremadamente grandes. Por tanto, debemos inicializar los pesos con valores muy pequeños. Además, este tipo de funciones pueden sufrir el problema de la ReLU moribunda. Como la función de activación desactiva la neurona cuando la entrada es negativa, existe la posibilidad de que los pesos se actualicen de manera que la neurona dé una salida nula para cualquier entrada. El gradiente en este caso sería cero y los pesos no se actualizarían. Por tanto, estas neuronas permanecerían desactivadas durante todo el entrenamiento provocando que una gran parte de la red quedara inactiva siendo la red incapaz de aprender correctamente. Este escenario es más probable cuando la tasa de aprendizaje es demasiado alta, por lo que se requiere una selección adecuada de la tasa de aprendizaje para aliviar el problema. Otra solución que se suele emplear es inicializar el sesgo a un valor positivo pequeño para asegurarnos de que las neuronas estén inicialmente activas.

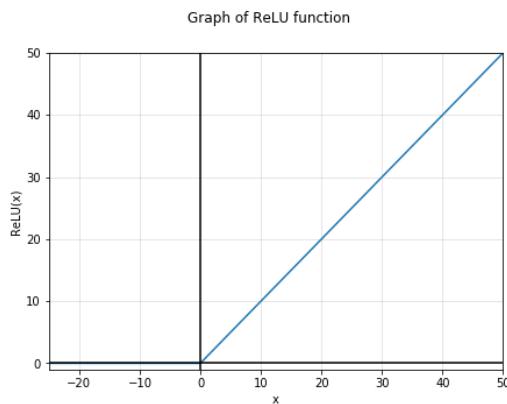


Figura 10.14.: Gráfica de la función ReLU [25]

Para evitar este problema se han introducido algunas modificaciones, no obstante, la función ReLU y sus variantes ofrecen mejor rendimiento y generalización en las redes neuronales que las funciones sigmoides. Debido a esto, son las más empleadas actualmente en las capas

ocultas de las redes neuronales, junto con otra función de activación en la capa de salida. Pasamos a comentar algunas de las modificaciones.

10.4.3.1. Función Leaky ReLU

La función Leaky ReLU o LReLU es una función de activación continua y no está acotada. Es muy poco costosa computacionalmente y está centrada en cero. Viene definida como:

$$LReLU : \mathbb{R} \rightarrow \mathbb{R}, LReLU(x) = \max(\alpha x, x)$$

Su derivada puede calcularse en cualquier punto menos en $x = 0$ y viene dada por:

$$\frac{dLReLU(x)}{dx} = \begin{cases} 1 & \text{si } x > 0 \\ \alpha & \text{si } x < 0 \end{cases}$$

Esta función utiliza una pequeña constante α que es la pendiente, toma valores cercanos a cero, normalmente $\alpha = 0.01$ de manera que $0 < \alpha < 1$ para evitar que el gradiente sea cero cuando la entrada es negativa, intentando solucionar el problema de la función ReLU. Sin embargo, presenta un riesgo de problema del desvanecimiento del gradiente en la parte negativa, ya que la derivada es cercana a cero.

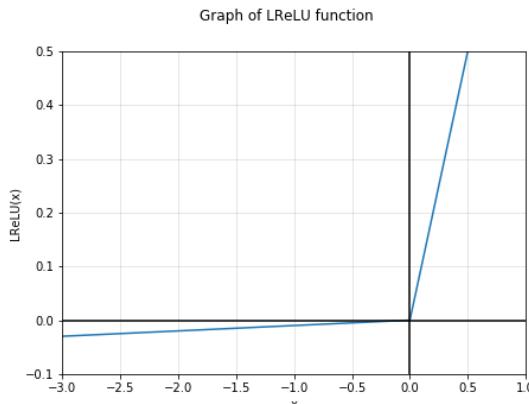


Figura 10.15.: Gráfica de la función LReLU [25]

10.4.3.2. Función ReLU Paramétrica

La función ReLU paramétrica o PReLU es una función continua, no acotada y centrada en cero.

Se define de la misma forma que la función LReLU:

$$LReLU : \mathbb{R} \rightarrow \mathbb{R}, LReLU(x_i) = \max(\alpha_i x_i, x_i)$$

Su derivada puede calcularse en cualquier punto menos en $x = 0$ y se define como en la

función LReLU.

Ahora la pendiente α la interpretan como un parámetro más de la neurona, que hay que entrenar como los pesos y puede ser diferente para cada neurona de la red. Al igual que la función LReLU puede sufrir el problema del desvanecimiento del gradiente en la parte negativa.

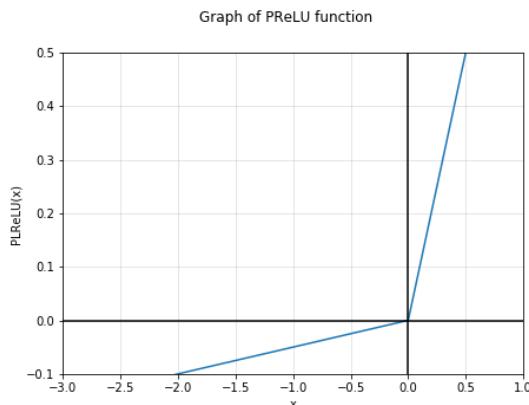


Figura 10.16.: Gráfica de la función PReLU [25]

10.5. Técnicas de Regularización

En aprendizaje automático, la regularización es cualquier modificación que realizamos sobre un algoritmo de aprendizaje con la intención de reducir su error de generalización (sobre el conjunto de prueba) pero no su error de resustitución (sobre el conjunto de entrenamiento). El objetivo de la regularización es prevenir el sobreaprendizaje. Para esta parte del capítulo seguiremos la fuente [4], por lo que la notación será algo distinta a la que venimos usando hasta ahora pero vendrá explicada con detalle.

Hay tres situaciones diferentes en las que puede encontrarse un modelo, como describimos anteriormente. La primera situación es aquella en la que un modelo sufre de sesgo, impide que el modelo sea capaz de representar correctamente el proceso real que da lugar a los datos observados, el modelo no es lo suficientemente flexible, concepto que llamamos underfitting. La segunda es la que buscamos, el modelo se adapta al proceso real del que se obtienen los datos, obteniendo resultados óptimos. La última es aquella en la que el error se debe a la varianza, debido a que el modelo es demasiado flexible y se adapta demasiado bien al conjunto de datos de entrenamiento, overfitting o sobreajuste. El objetivo de la regularización es llevar un modelo del tercer escenario al segundo.

Sabemos por el teorema de Wolpert, que nada es gratis, no existe un algoritmo de aprendizaje que sea consistentemente mejor que otro y lo mismo pasa con las técnicas de regularización. Para cada situación debemos elegir una forma de regularización que se adapte bien al problema particular que deseamos resolver. Existe un conjunto bastante amplio de técnicas genéricas de regularización que resultan aplicables a una amplia gama de problemas.

Podríamos pensar que si elegimos un tipo de modelo adecuado para el problema particular

que nos concierne, entonces no nos tendríamos que preocupar en regularizarlo. Sin embargo, en la práctica, no tenemos la capacidad de determinar cuál es el modelo adecuado y aunque tuviéramos esa capacidad es posible que no existiese ningún modelo realmente preciso para nuestro objetivo.

Para encontrar el mejor modelo, en el sentido de minimizar su error de generalización, debemos entrenar un modelo que sea lo suficientemente grande para evitar posibles sesgos y que haya sido debidamente regularizado. Un modelo de red neuronal será mejor que otro si permite identificar las causas reales que ocasionan variaciones observadas en los datos, sus factores causales, aunque estos son difíciles de identificar en la práctica. En problemas de aprendizaje supervisado, las etiquetas del conjunto de entrenamiento, mediante la definición de funciones de error, coste o pérdida nos servirán para ajustar los parámetros de la red. En problemas de aprendizaje no supervisado, las pistas son más indirectas, creencias previas impuestas anteriormente para guiar el aprendizaje, aunque esas guías no nos aportan nada en cuanto al error de generalización del modelo.

Existen esencialmente dos estrategias generales para ajustar la capacidad efectiva de una red neuronal mediante técnicas de regularización, las cuales son:

- Añadir restricciones adicionales al modelo de red neuronal, en forma de restricciones sobre los valores de sus parámetros. Por ejemplo, que diferentes neuronas o capas de la red comparten parámetros.
- Añadir términos extra en la función de coste o pérdida que se utiliza como objetivo en el proceso de optimización de los parámetros de la red. Esta estrategia puede ser vista como una forma de imponer restricciones débiles sobre los valores de los parámetros de la red, ya que nos permite establecer limitaciones de manera indirecta sobre las configuraciones posibles de la red.

Estas dos estrategias pueden ayudarnos a mejorar el rendimiento de la red neuronal sobre el conjunto de prueba.

10.5.1. Regularización de la función de coste

El método más común de regularizar el entrenamiento de una red neuronal artificial consiste en añadir términos adicionales a la función de coste que intentamos optimizar ajustando sus parámetros. La función de coste no es más que una función de error E que evaluamos sobre el conjunto de entrenamiento con la intención de que la red aprenda, ahora introducimos un término de regularización y la función de coste pasa a ser de la forma:

$$L = L_E + \lambda L_R$$

donde L_E es la función de error, λ es un parámetro de regularización para darle más o menos peso al término de regularización y L_R es el término de regularización. El gradiente de la función coste es ahora:

$$\nabla_x L = \nabla_x L_E + \lambda \nabla_x L_R$$

El resto del proceso de entrenamiento de la red neuronal sigue siendo el mismo. Hay una

diferencia en la señal que se propaga hacia atrás al usar backpropagation. En esa señal, además del error sobre el conjunto de entrenamiento, se incluye información que contribuirá a reducir el error de generalización de la red una vez haya sido entrenada.

Hay que tener en cuenta que hemos añadido un hiperparámetro más al algoritmo de entrenamiento de la red, el hiperparámetro de regularización λ , el cual nos permite controlar la capacidad de la red. Establecer un valor adecuado para el parámetro de regularización λ es esencial para controlar la capacidad de la red, previniendo el sobreaprendizaje y facilitando su capacidad de generalización.

Las regularizaciones de la función de coste tienen su origen en los métodos de contracción que se emplean en Estadística para reducir la varianza de un estimador de mínimos cuadrados por medio de la inclusión de restricciones de los valores de los coeficientes.

Las técnicas de regularización mantienen todas sus variables a diferencia de las técnicas de reducción de la dimensionalidad. Sin embargo, reducen el efecto de algunas de ellas sobre la salida del modelo penalizando los coeficientes asociados a dichas variables. A mayores coeficientes, mayor será la penalización y esto provoca que haya modelos en los que la magnitud de los coeficientes se reduce. En el caso de las redes neuronales, los parámetros ajustados son los pesos sinápticos y la reducción de su magnitud contribuye, además, a prevenir fenómenos como la saturación prematura, que dificulta el proceso de entrenamiento de la red.

La diferencia que reside en la forma que toma el término de penalización L_R en la función de pérdida utilizada para entrenar la red es la diferencia que existe entre las dos regularizaciones que pasamos a describir, la regularización L_1 y la regularización L_2 .

10.5.1.1. Regularización L2

$$L = L_E + \frac{1}{2} \lambda \sum w_k^2$$

La regularización L_2 es la regularización que se emplea de forma más habitual en redes neuronales artificiales y se suele conocer con el término *weight decay*, en español, decaimiento de pesos.

La norma L_2 de un vector w es su norma euclídea, $\|w\|_2$, que se usa entre otras cosas para medir distancias en un espacio euclídeo.

$$L_2 = \|w\|_2^2 = \sum w_k^2$$

Cuando calculamos el gradiente de la función regularizada que se emplea en el gradiente descendente con backpropagation, la actualización de los pesos se realiza de la siguiente forma:

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} = -\eta \left(\frac{\partial E}{\partial w_k} + \lambda w_k \right)$$

donde el único cambio que se introduce es un término adicional que contrae los pesos en un factor $-\eta\lambda$ en cada paso del entrenamiento de la red. El factor $1/2$ se introduce solo en el

término de regularización de la función pérdida L para que el 2 desaparezca al calcular la derivada. Así, el gradiente del término de regularización con respecto al vector de pesos w es, simplemente, λw .

La consecuencia de esta regularización es que el vector de pesos se acerca al origen, al incluir en cada actualización, un pequeño factor que disminuye la magnitud de los pesos. Este mecanismo proporciona una forma de hacer que los pesos tiendan a desaparecer salvo que su valor se refuerce como parte de la actualización debida al gradiente del error con respecto a los pesos. Una vez haya terminado el entrenamiento, los pesos más relevantes en reducir el error tendrán una magnitud significativamente distinta de cero. Así pues, la regularización L_2 penaliza fuertemente los pesos grandes y prima la obtención de vectores de pesos con valores pequeños.

Desde una perspectiva más formal, la regularización L_2 permite que el algoritmo de entrenamiento de una red neuronal perciba los datos de entrada como si tuviesen mayor varianza de la que realmente tiene, haciendo que el algoritmo deba reducir la magnitud de los pesos correspondientes a características cuya covarianza con la función objetivo es baja en comparación con esa varianza añadida. Los pesos que contribuyen a reducir el valor de la función de coste que pretendemos minimizar son los que se preservan relativamente intactos.

Esta regularización se utiliza en aplicaciones adicionales como en resoluciones de ecuaciones, análisis de componentes principales PCA, etc.

10.5.1.2. Regularización L1

$$L = L_E + \lambda \sum |w_k|$$

El nombre de la regularización L_1 se debe al empleo de la norma L_1 , que es la que se utiliza, por ejemplo, para calcular la distancia Manhattan.

$$L_1 = \|w\|_1 = \sum |w_k|$$

Ahora, a diferencia de la regularización L_2 , el gradiente de la función de pérdida hace que la actualización de los pesos usando el gradiente descendente sea:

$$\Delta w_k = -\eta \frac{\partial L}{\partial w_k} = -\eta \left(\frac{\partial E}{\partial w_k} + \lambda \text{sign}(w_k) \right)$$

donde $\text{sign}(w_k)$ es la función signo aplicada al peso w_k . Si el peso es positivo su valor se reducirá, por el contrario, si es negativo, su valor se incrementará. En ambos casos, el cambio en el peso debido al término de regularización será de la magnitud $\eta\lambda$. Ahora, la contribución del término de regularización al gradiente es constante. El gradiente del término de regularización con respecto al vector de pesos w es $\text{sign}(w)$.

En sus orígenes, esta regularización se propuso para darle a las redes neuronales la capacidad de olvidar. Así, las redes obtenidas eran algo menos sensibles a su configuración inicial.

Al igual que sucedía con la regularización L_2 , la regularización L_1 tiende a reducir los valores de los parámetros de un modelo y debido a la presión que hace muchos de ellos acabar siendo cero, conservando aquellos que tienen un mayor impacto en la función objetivo que

queremos minimizar. Los parámetros con valor igual a cero podremos eliminarlos, reduciendo así el tamaño de los modelos obtenidos, esto es, podando la red neuronal. Esta poda será mayor cuanto mayor sea el valor del parámetro de regularización λ . La regularización $L1$ nos ayuda a obtener soluciones más dispersas que la regularización $L2$, permitiendo una mejor generalización hasta cierto punto.

10.5.2. Restricciones sobre los parámetros de la red

Como alternativa a la regularización de la función de coste podemos realizar la regularización directamente sobre los valores de los pesos. Las formas más habituales consisten en imponer una serie de restricciones sobre los valores de los pesos, ya sea limitando su magnitud o vinculando los valores de diferentes pesos de la red.

El método de los multiplicadores de Lagrange permite resolver problemas con restricciones, reduce un problema de optimización de n variables y m restricciones a un problema de optimización sin restricciones de $n + m$ variables.

Supongamos que nuestro problema es optimizar los parámetros de una red neuronal con n pesos, es decir, queremos minimizar la función de error asociada a los pesos, $E(w)$. Además, queremos imponer una restricción sobre la solución que limite la norma euclídea del vector de pesos: $\|w\|_2 \leq k$. Normalizamos esta restricción definiendo la función:

$$\Omega(w) = \|w\|_2 - k \leq 0$$

Supongamos ahora que dibujamos las curvas de nivel de la función de error $E(w)$ y de la función de restricción $\Omega(w)$, estas curvas se intersectarán en una gran cantidad de puntos. Ahora fijamos un valor para $\Omega(w)$, en este caso $\Omega(w) = 0$ y nos movemos a lo largo de la curva para ver cómo varía $E(w)$. Cuando la curva $\Omega(w) = 0$ toca de manera tangencial una curva de nivel de $E(w)$, nos encontramos en un óptimo local de la función de error restringido a los puntos de la curva $\Omega(w) = 0$, es en ese punto donde las tangentes de la función de error y la de restricción son paralelas. Si generalizamos, podemos imponer la condición de tangencialidad de manera que los vectores asociados a los gradientes $E(w)$ y $\Omega(w)$ sean paralelos, puesto que los gradientes son perpendiculares a las líneas de contorno. Por tanto, buscamos puntos de manera que

$$\nabla_w E(w) = \lambda \cdot \nabla_w \Omega(w)$$

donde λ es el multiplicador de Lagrange y $\lambda \neq 0$, ya que los gradientes tienen que ser paralelos pero su magnitud no tiene por qué coincidir.

Definimos una función auxiliar llamada lagrangiano \mathcal{L} para incorporar a una ecuación la restricción anterior:

$$\mathcal{L}(w, \lambda) = E(w) - \lambda \Omega(w)$$

El método de los multiplicadores de Lagrange consiste en resolver el problema de optimización sin restricciones sobre el lagrangiano:

$$\nabla_{w,\lambda} \mathcal{L}(w, \lambda) = \nabla_w [E(w) - \lambda \Omega(w)] = 0$$

Ahora tenemos un problema de optimización sin restricciones con $n + 1$ variables. La función a optimizar tiene la misma forma que la función de coste regularizada, la cual recordamos:

$$L = L_E + \lambda L_R$$

La equivalencia se obtiene si hacemos $L_E = E(w)$ y $L_R = \Omega(w)$ tras cambiar el signo del parámetro λ . Por tanto, si imponemos restricciones sobre los valores de los parámetros de la red durante el entrenamiento obtendremos el mismo efecto que si regularizamos la función de coste.

Al utilizar regularización L_2 sabemos que cuanto mayor sea el parámetro de regularización λ , más restringimos los valores de los pesos, pero no sabemos cuál es el valor adecuado de λ para que los pesos sean de la magnitud que nos interesa, puesto que la relación entre λ y el vector de pesos w depende de $E(w)$. La derivación anterior nos permite obtener efectos similares sin tener que determinar el valor más adecuado de λ sustituyendo la penalización que realizábamos sobre la función de coste por una restricción implícita sobre el vector de pesos.

Para imponer una restricción explícita al vector de pesos lo que tenemos que hacer es ejecutar el algoritmo de entrenamiento y tras cada actualización de los pesos, si no se cumple la restricción $\|w\|_2 \leq k$, reproyectamos el vector de pesos para que verifique la restricción. Estableciendo una cota superior sobre la magnitud del vector de pesos de cada neurona podemos evitar el uso de la regularización L_2 .

Establecer restricciones sobre la norma del vector de parámetros es una forma efectiva que regulariza sus valores pero hay una forma más popular de imponer restricciones sobre los valores de los pesos de una red, forzando que sean iguales. Si vinculamos los valores de unos pesos con los de otros, reducimos la capacidad efectiva de la red, reduciendo así el sobreaprendizaje. Así, conseguimos regularizar la red actuando sobre el valor de sus pesos, esto se puede conseguir de diversas formas.

10.5.2.1. Compresión de parámetros

Comprimir el número de parámetros de la red tiene como finalidad representar el vector de pesos $w \in \mathbb{R}^K$ con menos de K parámetros.

Es posible reducir el número de parámetros de una red multicapa al sustituir una capa de n entradas y m salidas por dos capas, una capa lineal de n entradas y p salidas seguida de una capa con p entradas y m salidas, ambas con la misma función de activación. Ahora tenemos $np + pm = (n + m)p$ parámetros, número que puede ser significativamente menor al número nm si escogemos el valor p de manera adecuada. Al disminuir el número de parámetros de la red podemos facilitar su entrenamiento.

También es posible reducir directamente la matriz de pesos de una capa. Esta reducción es equivalente a comprimir la entrada de una capa empleando una descomposición de dos capas en la que la primera capa tiene conexiones sinápticas preestablecidas, las cuales no se entranan.

10. Aprendizaje Profundo

Por todo esto, podemos interpretar la comprensión de parámetros como una etapa previa de preprocesamiento, permitiendo reducir el coste computacional del proceso de entrenamiento. También puede servir para prevenir el sobreaprendizaje.

10.5.2.2. Parámetros compartidos

Este método de regularización recibe el nombre de compartición de pesos o parámetros ya que permite reducir el número efectivo de parámetros de la red compartiendo estos entre distintos modelos o entre distintos componentes dentro de una misma red.

Un ejemplo destacado de cómo incorporar este método son las redes convolucionales, ya que utilizan múltiples filtros del mismo tipo para diferentes posiciones de la imagen de entrada, reduciendo así el número de parámetros de la red, puesto que utilizamos los mismos parámetros para un filtro que se aplica sobre toda la imagen.

Al compartir parámetros, reducimos el uso de memoria necesario. También facilita a la prevención del sobreaprendizaje y facilita el entrenamiento de la red.

10.5.3. Introducción de ruido

Imponer restricciones sobre los parámetros de la red o regularizar la función de coste son dos de las alternativas de las que disponemos para intentar prevenir los efectos negativos del sobreaprendizaje. Una tercera alternativa consiste en añadir ruido pseudoaleatorio durante el proceso de entrenamiento. El efecto de añadir ruido artificialmente contribuye, de forma genérica, a hacer más robusta la red entrenada.

Hay varias formas de añadir ruido al entrenamiento de una red neuronal. Podemos añadir ruido a las entradas, ampliando el conjunto de entrenamiento para que la red sea robusta frente al tipo de ruido que esperamos encontrar en la práctica. Así permitimos mejorar su capacidad de generalización. En vez de actuar sobre las entradas que recibe la capa de entrada de la red, también podemos actuar sobre los niveles de activación de las capas ocultas, que se puede interpretar como añadir ruido a las entradas a diferentes niveles de abstracción, los correspondientes a las características de los datos de entrada que las neuronas de las capas ocultas son capaces de extraer. Esta estrategia parece ser la más efectiva para prevenir la saturación prematura de las neuronas de la red y, de ese modo, agilizar su entrenamiento. Por último, podemos añadir ruido a los pesos sinápticos de la red, sobre los niveles de activación de las capas ocultas de la red permitiendo que el ruido actúe directamente sobre las características que se extraen de los datos de entrada, características representadas implícitamente por los niveles de activación de las neuronas ocultas de una red multicapa.

10.5.4. Early Stopping

Además de modificar la función de coste, introducir restricciones sobre los parámetros de la red o introducir, de algún modo, ruido en el progreso de entrenamiento, existe una técnica de regularización muy efectiva que es una de las más utilizadas actualmente en deep learning. La técnica se denomina parada temprana, en inglés *early stopping*. Puede verse como una forma de regularización en el tiempo.

El entrenamiento de una red neuronal es un proceso iterativo, en el que vamos ajustando los pesos de la red para resolver el problema planteado. Sin embargo, en el proceso tradicional de entrenamiento de la red carecemos de señales externas, más allá del conjunto de entrenamiento, por tanto, no sabemos cuándo estamos consiguiendo que la red generalice bien y cuándo está sobreaprendiendo. Por este motivo, vamos a utilizar un conjunto de datos independiente del conjunto de entrenamiento, el conjunto de datos de validación, que nos servirá para guiar el proceso de entrenamiento de la red y saber cuándo debemos parar.

El conjunto de validación lo emplearemos para evaluar periódicamente el error de generalización de la red conforme avanza en su entrenamiento. No podemos utilizar el conjunto de entrenamiento, ya que no nos servirá como medida del error de generalización de la red.

Si el algoritmo de entrenamiento de una red neuronal funciona correctamente, la medida de error sobre el conjunto de entrenamiento irá disminuyendo en cada iteración, convergiendo hacia un mínimo de la función de coste o pérdida. Sin embargo, el error sobre el conjunto de validación llega un momento que tiende a aumentar cuando el modelo empieza a dar muestras de sobreaprendizaje. Una vez detectemos esta situación, deberíamos detener el algoritmo iterativo de aprendizaje y devolver los valores de los parámetros para los que el error sobre el conjunto de validación era mínimo.

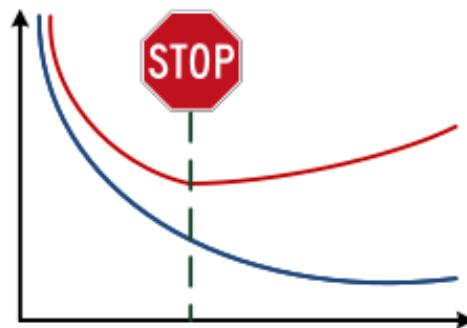


Figura 10.17.: Ejemplo de uso de la técnica early stopping, simulación de entrenamiento de una red, conjunto de validación (en rojo), conjunto de entrenamiento (en azul). [4]

Una de las ventajas de esta técnica es que permite reducir el coste computacional de entrenamiento de la red. Además, proporciona un mecanismo de regularización bastante efectivo sin necesidad de que tengamos que manipular artificialmente la función de coste o introducir restricciones sobre los parámetros de la red. Al igual que cualquier otra técnica de regularización permite contribuir a reducir la necesidad de preprocesar los datos de entrada.

Por otra parte, early stopping tiene algunos efectos secundarios. Tiende a restringir el espacio de búsqueda de valores adecuados para los parámetros de la red a una región, relativamente pequeña, en el entorno de los parámetros iniciales de la red. El momento en el que se detiene el proceso de aprendizaje puede no ser el más adecuado para el conjunto total de datos disponibles para el entrenamiento de la red.

Por todo esto, en la práctica se suele emplear de manera muy habitual early stopping, ya que es una forma sencilla, rápida y económica de regularizar un modelo y prevenir los efectos negativos del sobreaprendizaje.

10.5.5. Dropout

Una de las técnicas de regularización más eficaces y que se utiliza habitualmente en aprendizaje profundo es el método denominado *Dropout* [82]. Durante cada iteración de entrenamiento, las neuronas ocultas individuales de la red se desactivan con una probabilidad $p \in [0, 1)$, de manera que la red resultante tiene un tamaño reducido. De este modo, un porcentaje p de neuronas ocultas elegidas al azar se ignoran temporalmente, no se consideran durante la propagación hacia adelante y hacia atrás. Al final de la iteración de entrenamiento, las neuronas desactivadas se restauran y un nuevo conjunto de neuronas se desactiva durante la siguiente iteración.

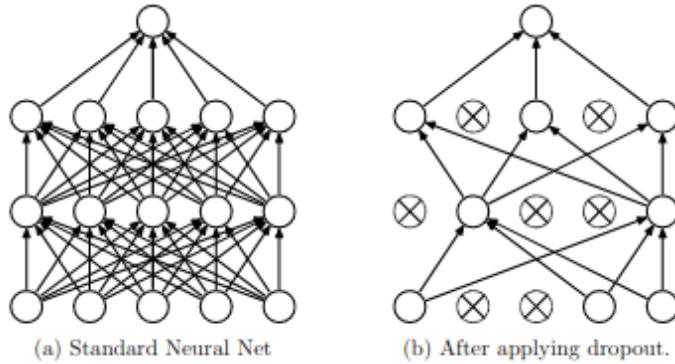


Figura 10.18.: Ejemplo de un modelo de red neuronal dropout con $p = 0,5$. A la izquierda aparece una red neuronal estándar con 2 capas ocultas y a la derecha la misma red producida al aplicar dropout a la red de la izquierda. Las unidades cruzadas se han eliminado. [82]

Como en el momento de la prueba se emplea toda la red, habrá que compensar aquellas neuronas perdidas durante el proceso de entrenamiento multiplicando por p los pesos asociados a las capas en las que se aplicó la desactivación. Esto quiere decir que, si una neurona se desactiva con una probabilidad p en la etapa de entrenamiento entonces tenemos que multiplicar por p los pesos salientes de esa neurona en el momento de la prueba. Así, la salida esperada de cada neurona en el momento de entrenamiento cuando se utilizó el abandono es la misma que la salida en el momento de la prueba o test.

Sin el método dropout las neuronas son propensas a desarrollar codependencia entre ellas, de manera que cada una de ellas se especializa excesivamente en una determinada tarea limitando su potencia, provocando sobreajuste. Por esto, empleamos el método dropout, para desconectar ciertas neuronas durante el entrenamiento y así evitar que haya demasiada codependencia entre ellas. Dropout aumenta el número de iteraciones necesarias para que el algoritmo de optimización converja pero el tiempo requerido para cada iteración disminuye.

10.5.6. Batch normalization

Por último, vamos a explicar una técnica que se emplea para facilitar el entrenamiento de las redes neuronales profundas, conocida como *Batch Normalization* [50], en español, norma-

lización por lotes. Consiste en un método de reparametrización adaptativa cuyo objetivo principal es evitar las diferencias que surgen entre las capas de una red muy profunda, lo que los autores del método denominan desplazamiento interno de covariables. Sin el uso de esta técnica es poco probable que las redes neuronales profundas se entrenen con éxito con el uso del gradiente descendente y el algoritmo de backpropagation. Puesto que las redes neuronales con muchas capas implican la composición de múltiples funciones, pueden surgir resultados inesperados al actualizar los pesos de todas las capas mediante el descenso del gradiente.

La normalización por lotes consiste básicamente en añadir un paso extra, normalmente entre las neuronas y la función de activación, con la idea de normalizar las entradas de cada capa, lo que permite aliviar el problema de coordinar las actualizaciones entre capas. Lo ideal sería que la normalización se hiciera usando la media y la varianza de todo el conjunto de entrenamiento, pero si estamos aplicando el descenso del gradiente estocástico para entrenar a la red se emplea la media y la varianza de cada mini-lote de entrada. Hay que tener en cuenta que cada salida de cada neurona se normalizará de manera independiente, en cada iteración se calcula la media y la varianza de cada salida para el mini-lote en curso.

Tras la normalización se añaden dos parámetros, un sesgo como sumando y otra constante similar a este que aparece multiplicando cada activación. Esto permite ayudar a nuestra red a ajustarse a los datos de entrada y a reducir las oscilaciones de la función de coste. Como consecuencia, podremos aumentar la tasa de aprendizaje y la convergencia hacia el mínimo global se producirá más rápidamente.

La normalización por lotes es una técnica que ayuda al entrenamiento más que una estrategia de regularización en sí misma. Esto se logra gracias a los momentos, permitiendo que al introducir un nuevo mini-lote de entrada no se usen una media y una desviación muy distintas a las de la iteración anterior, se tendrá en cuenta el histórico, y se elegirá una constante que pondere la importancia de los valores del mini-lote actual frente a los valores del anterior. Gracias a esto mejoramos la capacidad de generalización de la red y conseguimos reducir el sobreajuste.

11. Redes Neuronales Convolucionales

Las redes convolucionales, también conocidas como redes neuronales convolucionales o CNN, son un tipo especializado de red neuronal para procesar datos que tienen una topología conocida de tipo reticular, es decir, una topología en forma de cuadrícula, como los sonidos, las imágenes o los vídeos. El nombre de red neuronal convolucional se debe a que la red emplea una operación matemática llamada convolución, que es un tipo especializado de operación lineal. En este capítulo se presentan las redes neuronales convolucionales comenzando por describir qué es la operación de convolución. Tras esto, se explican los tipos de capas que existen y algunas de las arquitecturas más importantes.

Las redes neuronales descritas hasta ahora no se adaptan bien a las imágenes. Sin embargo, las redes neuronales convolucionales [46] aprovechan el hecho de que la entrada es una imagen y restringen la arquitectura de manera más sensata. Así, podemos por ejemplo, reducir de manera notable la cantidad de parámetros de la red. Este tipo de redes son muy similares a las descritas en el capítulo anterior, están formadas por neuronas que tienen pesos y sesgos aprendibles. Cada neurona recibe unas entradas, realiza un producto escalar y obtiene unos valores de salidas a los que se le aplica una función de activación. Además, siguen teniendo una función de coste o pérdida que queremos minimizar.

Las capas de una red neuronal convolucional tienen neuronas dispuestas en 3 dimensiones, anchura, altura y profundidad, siendo la profundidad la tercera dimensión de un volumen de activación. Podemos ver un ejemplo de ello:

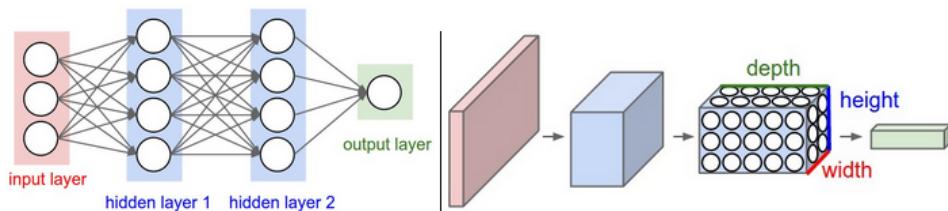


Figura 11.1.: A la izquierda se observa una red neuronal de tres capas. A la derecha una red neuronal convolucional organiza sus neuronas en tres dimensiones (anchura, altura y profundidad). Cada capa transforma el volumen de entrada 3D en un volumen de salida 3D de activaciones neuronales. La capa de entrada roja contiene la imagen, la anchura y la altura serían las dimensiones de la imagen y la profundidad sería tres (que se corresponde con los canales rojo, verde y azul). [36]

Este tipo de redes neuronales recibirán un conjunto de imágenes de entrada de dimensiones $x \times y \times z$ (ancho, alto y profundidad respectivamente). Se procesarán en las capas ocultas donde pronto veremos que las neuronas de una capa solo estarán conectadas a una pequeña

región de la capa anterior, en vez de estar conectadas a todas las neuronas de forma totalmente conectada. Al final de la arquitectura se reduce la imagen completa a un único vector de puntuaciones de clase, dispuesto a lo largo de la dimensión de profundidad que es lo que se recoge en la capa de salida.

11.1. La Operación de Convolución

La operación de convolución que utilizan las redes neuronales convolucionales no se corresponde exactamente con la definición matemática de convolución pero sí que está estrechamente relacionada con ella. En el ámbito matemático, en concreto, en la rama de análisis funcional, la operación de convolución es una operación matemática sobre dos funciones reales de variable real de manera que estas se transforman en una tercera que suele representar la magnitud en la que se superponen una modificación de una de las dos funciones por la otra. Esta operación se define como la integral del producto de dos funciones f y g después de invertir y desplazar una de ellas siempre y cuando exista la integral. Vamos a definir esta operación formalmente, para ello hemos empleado la siguiente bibliografía, [44], [70] y [43].

Definición 11.1. (Convolución) Sean $f, g : \mathbb{R} \rightarrow \mathbb{C}$ dos funciones medibles en \mathbb{R} . Para cada $t \in \mathbb{R}$ tenemos que la función $x \rightarrow f(x)g(t-x)$ es medible en \mathbb{R} y decimos que la convolución de f y g denotada por $f * g$ está definida en el punto t si se cumple la condición

$$\int_{-\infty}^{+\infty} |f(x)g(t-x)|dx < \infty$$

en cuyo caso se define

$$h(t) = (f * g)(t) = \int_{-\infty}^{+\infty} f(x)g(t-x)dx$$

El término convolución se utiliza para referirse tanto a la operación matemática como a la función resultante. Podemos interpretar la convolución de manera geométrica como el área de la función g que queda solapada por la función f cuando g se desplaza por el eje de abcisas, donde t representa el paso del tiempo. La operación de convolución es un operador lineal, se comporta de forma parecida a la multiplicación y verifica una serie de propiedades:

- *Commutativa.* $f * g = g * f$
- *Asociativa.* $(f * g) * h = f * (g * h)$
- *Distributiva.* $(f + g) * h = f * h + g * h$

En el procesamiento digital de señales, se utiliza la convolución discreta, ya que se trabaja con señales discretas y la definición que acabamos de ver es para sistemas continuos. Veamos ahora la convolución discreta para el caso unidimensional y posteriormente extenderemos la definición al caso multidimensional, puesto que en imágenes se suele trabajar con convoluciones de dos dimensiones. La versión discreta usa integración numérica. Si restringimos el conjunto de definición de nuestras funciones a los enteros, \mathbb{Z} , podemos definir dadas dos funciones $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ la operación de convolución discreta como:

$$h(t) = (f * g)(t) = \sum_{n \in \mathbb{Z}} f(n)g(t - n)$$

Para el caso bidimensional como es el caso de las imágenes donde las funciones f y g están definidas de manera que $f, g : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ la operación de convolución se define como:

$$h(s, t) = (f * g)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} f(n, m)g(s - n, t - m)$$

En el contexto en el que nos movemos denotaremos de forma especial a f y a g . La función f se denotará como x y será la entrada. A la función g la denotaremos como w y será el núcleo o filtro. Por tanto, f será nuestra imagen de entrada y g un núcleo que formará parte de la red neuronal. Al trabajar en el contexto de una máquina con memoria finita, nuestras funciones serán idénticamente nulas salvo en un conjunto finito de puntos.

Además, vamos a emplear la propiedad conmutativa de la convolución para reescribir la expresión anterior, ya que nos será más práctico a la hora de hacer cálculos. Esto es debido a que esta expresión nos permite calcular los valores de la convolución como una expresión que actúa sobre todos los puntos de nuestra imagen de entrada. Así, tenemos que la definición de convolución discreta unidimensional quedaría como:

$$h(s, t) = (x * w)(s, t) = (w * x)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} x(s - n, t - m)w(n, m)$$

Aunque la propiedad conmutativa es útil para demostrar sus propiedades no es una propiedad a tener en cuenta cuando se implementa una red neuronal. En su lugar, existe una operación que recibe el nombre de correlación cruzada y tiene como única diferencia que no invierte el núcleo. Así para el caso bidimensional la correlación cruzada sería:

$$h(s, t) = (x * w)(s, t) = \sum_{n \in \mathbb{Z}} \sum_{m \in \mathbb{Z}} x(s + n, t + m)w(n, m)$$

Muchas bibliotecas de aprendizaje profundo implementan la correlación cruzada pero la llaman convolución. En este trabajo seguimos la misma convención, llamamos a ambas operaciones convolución especificando si nos referimos a invertir el núcleo o no en contextos donde sea relevante. Además, nuestros algoritmos de aprendizaje buscarán aprender los valores adecuados del núcleo que les permitirán extraer características relevantes de los datos de entrada. También es extraño que la convolución se utilice sola, en nuestro caso siempre aparecerá compuesta con otra serie de funciones, por lo que la propiedad de conmutatividad se pierde en la mayoría de los casos independientemente de si el núcleo gira o no.

Hasta ahora hemos visto la operación de convolución desde un punto de vista teórico. Vamos ahora a explicar desde un punto de vista práctico en qué consiste y terminaremos con un ejemplo visual. Todas estas explicaciones las veremos aplicando esta operación sobre una imagen de entrada en forma de matriz multidimensional.

En el procesamiento de imágenes, la convolución es el proceso de transformación de una imagen mediante la aplicación de un núcleo sobre cada píxel y sus vecinos locales en toda la imagen. El núcleo es una matriz de valores, cuyo tamaño y valores determinan el efecto

11. Redes Neuronales Convolucionales

de transformación del proceso de convolución. El proceso de convolución implica una serie de pasos. En primer lugar, se coloca la matriz del núcleo sobre cada píxel de la imagen (hay que asegurarse que el tamaño del núcleo no sea mayor que el de la imagen). Tras esto, se multiplica cada valor del núcleo con el píxel correspondiente sobre el que está. Posteriormente se suman los valores multiplicados resultantes y se devuelve el valor obtenido como el nuevo valor del píxel central. Tras esto, se desliza el kernel por la entrada y se repite el proceso en toda la imagen.

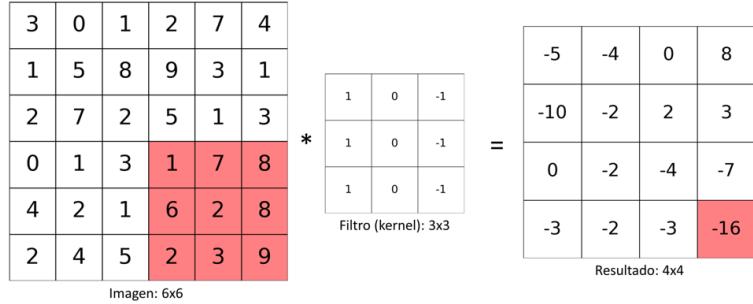


Figura 11.2.: Ejemplo de operación de convolución. [81]

Si aplicamos determinados filtros a una imagen real podemos obtener los siguientes resultados:

<i>Original</i>	<i>Gaussian Blur</i>	<i>Sharpen</i>	<i>Edge Detection</i>
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

Figura 11.3.: Ejemplo práctico sobre una imagen utilizando diferentes filtros. [18]

Los filtros de la figura 11.3 sirven para resaltar diferentes características, el desenfoque gaussiano permite suavizar la imagen antes del procesamiento, el filtro Sharpen permite mejorar la profundidad de los bordes y el último filtro se emplea para la detección de bordes. Una vez vista la operación de convolución explicaremos los tipos de capas que existen en una red neuronal convolucional.

11.2. Tipos de capas

Una red neuronal convolucional como hemos dicho previamente transforma un volumen de activaciones en otro a través de una función diferenciable. Utilizamos tres tipos principales

de capas para construir arquitecturas de este tipo, capas convolucionales, capas de agrupación y capas totalmente conectadas o densas. Estas capas las apliaremos para formar una arquitectura de red neuronal convolucional completa.

11.2.1. Capas Convolucionales

Las capas convolucionales son el bloque central de una red neuronal convolucional y realizan la mayor parte del trabajo computacional.

En primer lugar vamos a hablar de lo que calcula la capa de convolución. Una capa de convolución está formada por un conjunto de filtros que la red va aprendiendo. Cada filtro es pequeño espacialmente, pero se extiende por toda la profundidad del volumen de entrada. Como ejemplo de un filtro típico de la primera capa tenemos un filtro de tamaño $5 \times 5 \times 3$, que quiere decir que hay 5 píxeles tanto de ancho como de alto y el 3 se refiere a la profundidad, en el caso de imágenes es 3 debido a los canales de color. Durante la convolución deslizamos (convolucionamos) cada filtro a lo largo de la anchura y de la altura del volumen de entrada y calculamos los productos escalares entre las entradas del filtro y la entrada de cualquier posición. Al deslizar el filtro por la anchura y la altura del volumen de entrada, producimos un mapa de activación bidimensional que da las respuestas de ese filtro en cada posición espacial. De manera intuitiva, la red aprenderá filtros que se activan cuando ven algún tipo de característica visual como una mancha de algún color, un borde de alguna orientación y cuando nos adentramos en capas superiores de la red estos patrones se vuelven más complejos. Ahora, tendremos un conjunto completo de filtros en cada capa de la red convolucional y cada uno de ellos producirá un mapa de activación bidimensional independiente. Estos mapas de activación se apilarán a lo largo de la dimensión de profundidad y así produciremos el volumen de salida.

Ahora vamos a discutir los detalles de la conectividad de las neuronas, su disposición en el espacio y su esquema de compartición de parámetros.

- *Conectividad local.* Esta es la principal motivación para la creación de las redes neuronales convolucionales. Si trabajamos con entradas de alta dimensión, como por ejemplo con imágenes, es poco práctico conectar todas las neuronas de dos capas entre sí. En su lugar, conectaremos cada neurona solo a una región local del volumen de entrada. La extensión espacial de esta conectividad es un hiperparámetro llamado campo receptivo de la neurona, que equivale al tamaño del filtro. La extensión de la conectividad a lo largo del eje de profundidad es siempre igual a la profundidad del volumen de entrada. Conviene volver a destacar la asimetría en el tratamiento de las dimensiones espaciales (anchura y altura) y la dimensión de profundidad. Las conexiones son locales a lo largo de la anchura y la altura y son completas a lo largo de toda la profundidad del volumen de entrada.

Por ejemplo, supongamos que el volumen de entrada tiene un tamaño de $[32 \times 32 \times 3]$. Si el tamaño del filtro (o el campo receptivo) es de tamaño 5×5 entonces cada neurona de la capa convolucional tendrá pesos para una región de $[5 \times 5 \times 3]$ en el volumen de entrada, para un total de $5 \cdot 5 \cdot 3 = 75$ pesos, más 1 parámetro de sesgo. Obsérvese que la extensión de la conectividad a lo largo del eje de profundidad debe ser 3, ya que esta es la profundidad del volumen de entrada.

- *Disposición espacial.* Acabamos de explicar la conectividad de cada neurona de la capa

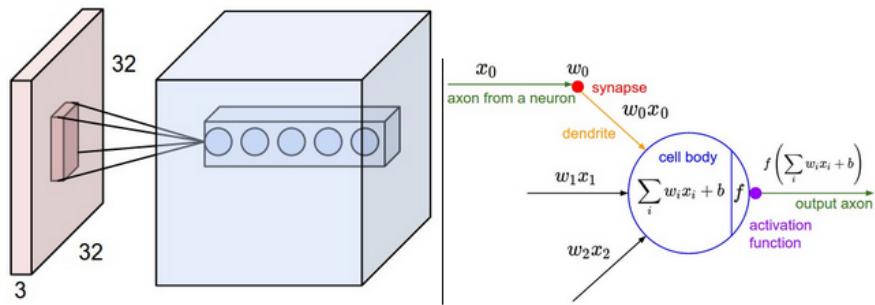


Figura 11.4.: A la izquierda tenemos un ejemplo de volumen de entrada en rojo de tamaño $32 \times 32 \times 3$ y un ejemplo de capa convolucional con un volumen de neuronas. Cada neurona de la capa convolucional está conectada a una sola región local del volumen de entrada, pero a toda la profundidad, es decir, a todos los canales de color. En este caso tenemos 5 neuronas a lo largo de la profundidad, todas ellas mirando a la misma región de entrada, comparten el mismo campo receptivo pero no los mismos pesos. A la derecha podemos ver que las neuronas de las redes previamente estudiadas no son distintas de las redes neuronales convolucionales. [36]

convolucional con el volumen de entrada. Ahora vamos a hablar de cuántas neuronas hay en el volumen de salida y de cómo están dispuestas. Existen tres hiperparámetros que controlan el tamaño del volumen de salida: la profundidad, la zancada y el *zero-padding* o relleno cero.

- *Profundidad*. La profundidad del volumen de salida es un hiperparámetro que se corresponde con el número de filtros que queremos utilizar, cada uno de los cuales aprende a buscar algo diferente en la entrada. Por ejemplo, si la primera capa convolucional recibe como entrada la imagen en bruto, entonces diferentes neuronas a lo largo de la dimensión de profundidad pueden activarse en presencia de una serie de características como bordes orientados, manchas de color, etc. Nos referiremos a un conjunto de neuronas que miran todas a la misma región de la entrada como una columna de profundidad.
- *Zancada*. En segundo lugar deberemos especificar la zancada con la que deslizaremos el filtro, esta se mide por *strides*. Cuando el stride es 1 movemos los filtros un píxel a la vez, si son 2 los filtros se mueven 2 píxeles a la vez mientras los deslizamos, a veces el stride puede ser de tamaño 3 o mayor. Así, conseguimos volúmenes de salida más pequeños al reducir la dimensión espacial.
- *Relleno cero*. A veces será conveniente llenar el volumen de entrada con ceros alrededor del borde. El tamaño de este relleno de ceros es un hiperparámetro. Gracias a esto podremos controlar el tamaño espacial de los volúmenes de salida, lo más habitual es utilizarlo para preservar exactamente el tamaño espacial del volumen de entrada, de manera que la anchura y la altura de la entrada y la salida sean las mismas.

Es posible calcular el tamaño espacial del volumen de salida en función del tamaño del volumen de entrada (W), del tamaño del campo receptivo de las neuronas de la capa,

es decir, de la profundidad (F), de la zancada (S) y de la cantidad de relleno cero (P) utilizado en el borde. La fórmula correcta para calcular el número de neuronas que caben viene dada por $(W - F + 2P)/S + 1$. Por ejemplo, dada una entrada de tamaño 7×7 y un filtro 3×3 con zancada 1 y zero-padding 0 obtendríamos una salida de 5×5 y con zancada 2 una salida de 3×3 . Veamos un ejemplo gráfico:

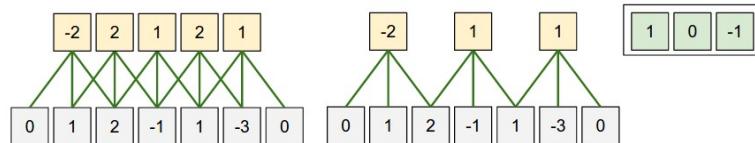


Figura 11.5.: Ilustración de la disposición espacial. En este ejemplo solo contamos con una dimensión espacial, el tamaño de entrada es $W = 5$, una neurona con un tamaño de campo receptivo de $F = 3$, y un relleno cero de $P = 1$. A la izquierda la neurona recorre la entrada en zancadas de $S = 1$ dando una salida de tamaño $(5 - 3 + 2)/1 + 1 = 5$. A la derecha la neurona emplea zancadas de $S = 2$ y la salida tiene tamaño $(5 - 3 + 2)/2 + 1 = 3$. Obsérvese que la zancada $S = 3$ no se puede utilizar, debido a que no cabría en el volumen. En términos de la ecuación, tenemos que $(5 - 3 + 2) = 4$ y no es divisible por 3. Los pesos de la neurona en este ejemplo son $[1, 0, -1]$, y su sesgo es cero. Estos pesos los comparten todas las neuronas amarillas. [36]

En el ejemplo de la izquierda vemos que la dimensión de entrada y la de salida son iguales, de tamaño 5. Esto es debido a que nuestros campos receptivos eran 3 y utilizamos el relleno cero de 1. Si no se emplea el relleno cero, entonces el volumen de salida habría tenido una dimensión espacial de solo 3, ya que ese hubiera sido el número de neuronas que hubieran cabido en la entrada original. En general, establecer el relleno cero para que sea $P = (F - 1)/2$ cuando la zancada es $S = 1$ asegura que el volumen de entrada y el de salida tengan el mismo tamaño espacial.

- *Parámetros compartidos.* Ya vimos en el capítulo anterior en concreto en 10.5.2.2 este método de regularización y ahora vamos a desarrollarlo un poco más a fondo en el caso de redes neuronales convolucionales. Este esquema se emplea para controlar el número de parámetros.

Podemos reducir drásticamente el número de parámetros si realizamos una suposición razonable: si una característica es útil para calcular en alguna posición espacial (x_1, y_1) , entonces también debería ser útil para calcular en una posición diferente (x_2, y_2) . En otras palabras, vamos a restringir las neuronas en cada corte de profundidad para utilizar los mismos pesos y sesgos. En la práctica, durante la retropropagación, cada neurona del volumen calculará el gradiente de sus pesos, pero estos gradientes se sumarán en cada corte de profundidad y solo actualizarán un único conjunto de pesos por corte.

Si todas las neuronas de un mismo corte de profundidad utilizan el mismo vector de pesos, entonces la propagación hacia delante de la capa convolucional puede calcularse en cada corte de profundidad como una convolución de los pesos de la neurona con el volumen de entrada. Debido a esto es común referirse a los conjuntos de pesos como un filtro o un núcleo, que se convoluciona con la entrada.

A veces la suposición de compartir parámetros puede no tener sentido. Este es el caso especial cuando las imágenes de entrada a una red convolucional tienen alguna estructura centrada específica, en la que deberíamos esperar, por ejemplo, que se aprendieran características completamente diferentes en un lado de la imagen que en otro. Un ejemplo práctico son los rostros que han sido centrados en la imagen, para los que se debería esperar que características específicas como los ojos o el cabello se aprendan en diferentes ubicaciones espaciales. En estos casos, es común relajar el esquema de compartición de parámetros y en su lugar, tener una capa totalmente conectada.

11.2.1.1. Convolución 1x1

Vamos a dedicar un espacio a este tipo de filtros, a los filtros de tamaño 1×1 , que han sido empleados en dos de las arquitecturas que hemos utilizado para realizar este trabajo.

En el caso de datos bidimensionales, una convolución 1×1 simplemente escala la entrada por el valor del filtro. Sin embargo, como las capas convolucionales trabajan con datos multicanal, se puede emplear una convolución de este tipo para reducir la dimensionalidad de la entrada. En efecto, dado un volumen de entrada de tamaño $W \times H \times D$, tras aplicar una convolución 1×1 con un filtro de tamaño $1 \times 1 \times D$, obtenemos una salida bidimensional de tamaño $W \times H \times 1$. Aplicando K de estos filtros y concatenando los resultados, podemos obtener un volumen de salida con profundidad $K < D$. Así, las convoluciones 1×1 combinan diferentes canales de la entrada utilizando los valores del núcleo y nos permiten aprender de las interacciones entre canales, permitiendo en nuestro caso como veremos, que puedan combinar los diferentes tableros de entrada. Este tipo de filtros se suelen emplear en las últimas capas de la red, ya que las primeras capas necesitan utilizar filtros más grandes para poder capturar la información espacial local.

11.2.2. Capas de Agrupación

Es habitual insertar una capa de agrupación, denominada en inglés capa de *pooling* entre las sucesivas capas convolucionales en una arquitectura de redes neuronales. La función de esta capa consiste en reducir progresivamente el tamaño espacial de la representación, es decir, reduce el tamaño de los mapas de características, de manera que también se reduce la cantidad de parámetros y cálculos que efectúa la red, y así, evitamos el sobreajuste y reducimos el tiempo de cálculo necesario.

La operación de pooling puede utilizarse para reducir el tamaño espacial de los datos, esto se conoce como pooling espacial, que es el caso habitual. También se puede emplear para reducir la profundidad de los volúmenes, es decir, reducir el número de canales, lo que se denomina pooling de canales cruzados y es lo que hacen las convoluciones 1×1 que acabamos de explicar. Nos vamos a centrar en el pooling espacial. En concreto, lo que hace una función de pooling es sustituir la salida de una capa convolucional en una determinada posición por una estadística resumen de los valores vecinos de la salida en esa posición. Las funciones de pooling más populares son las siguientes:

- *Max Pooling*: esta es la más común dentro de las capas de pooling, suele emplear filtros de tamaño (2×2) . Se encarga de calcular el valor máximo dentro de una vecindad rectangular de la salida en cada posición.

- *Average Pooling*: esta es la segunda técnica más conocida y empleada dentro de las capas de pooling. Se encarga de calcular en cada posición de la salida la media de los valores dentro de una vecindad rectangular.
- *L2 norm Pooling*: se calcula la norma L2 de los valores dentro de una vecindad rectangular de la salida en cada posición.

En la práctica se ha observado que la técnica de max pooling es la que funciona mejor. Al igual que sucede con las capas convolucionales las capas de pooling dependen de hiperparámetros que deben ser ajustados:

- *Extensión espacial, F*. Tamaño del vecindario considerado para producir cada valor de la salida de la capa de pooling. Normalmente es la misma vertical y horizontalmente.
- *Zancada, S*. El tamaño del deslizamiento sobre la entrada de la capa de pooling para obtener valores consecutivos de la salida. Normalmente, se elige el desplazamiento para que sea el mismo que la extensión espacial para evitar que haya solapamiento.

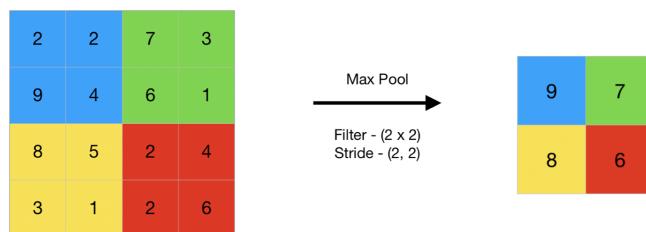


Figura 11.6.: La imagen muestra el empleo de la técnica de max pooling con extensión espacial 2×2 y stride 2. [39]



Figura 11.7.: La imagen muestra el empleo de la técnica de average pooling con extensión espacial 2×2 y stride 2. [39]

En una red neuronal, normalmente las capas de agrupación se insertan entre capas convolucionales consecutivas. Por tanto, tenemos que una capa de agrupación recibe un volumen de entrada de tamaño $W \times H \times D$, que es el volumen de salida de una capa convolucional anterior. Si la extensión espacial de la operación de pooling viene dada por $F \times F$ y utilizamos la zancada S en ambas dimensiones, el volumen de salida de la capa de pooling estará formado por D canales. Cada uno de estos canales tendrá una anchura $(W - F)/S + 1$ y una altura $(H - F)/S + 1$. La profundidad del volumen no cambia porque la operación de pooling espacial se aplica de forma independiente en cada mapa de características 2D. Puesto que el objetivo de las capas de agrupación es reducir las dimensiones de la entrada, no tiene sentido emplear el relleno cero en este tipo de capas.

11. Redes Neuronales Convolucionales

Cabe destacar que solo hay dos variaciones de la capa de agrupación máxima que se encuentran en práctica. Una capa de pooling con $F = 3$ y $S = 2$ (a veces llamada *overlapping pooling*, pooling de solapamiento), y más comúnmente, $F = 2$ y $S = 2$. Los tamaños de pooling con extensiones espaciales mayores son demasiado destructivas, por lo que no son frecuentes en la práctica. A diferencia de otras capas, las capas de pooling no introducen ningún parámetro aprendible, puesto que solo calculan una función fija de la entrada elegida a mano.

Una desventaja que producen las capas de agrupación es la pérdida de información sobre la posición exacta de las características de la imagen de entrada. Después de algunas capas de agrupación, la red pierde por completo la capacidad de localizar las características en la señal de entrada. Por esto, las capas de agrupación se utilizan con mayor frecuencia en contextos en los que es más importante determinar si una determinada característica está presente en la entrada que su localización exacta.

Actualmente hay controversia sobre el empleo de este tipo de capas, puesto que algunos piensan que puede lograrse una reducción sin el empleo de este tipo de capas e incluso se ha demostrado de manera empírica que ciertos modelos rinden mejor sin usar las capas de agrupación.

11.2.3. Capas Totalmente Conectadas o Densas

Las capas totalmente conectadas o densas de una red neuronal, en inglés *fully-connected* (FC), son aquellas en las que cada una de las salidas de la capa previa está conectada con un peso a cada una de las neuronas de la capa totalmente conectada. Por eso, sus salidas se pueden calcular con una multiplicación matricial seguida de una compensación de sesgo.

A diferencia del resto de capas de una red neuronal convolucional que actúan de manera independiente de las dimensiones del volumen de entrada a la red, como las capas convolucionales que están conectadas a una sola región local de la entrada; el número de pesos de las capas totalmente conectadas incrementa rápidamente cuando el número de entradas o el número de neuronas de la capa incrementa. Debido a esto, estas capas son las que normalmente determinan el volumen de entrada a la red para que pueda funcionar correctamente. En las redes neuronales convolucionales este tipo de capas suelen encontrarse al final de la arquitectura.

11.3. Arquitecturas

En la sección anterior hemos visto que las redes convolucionales suelen estar formadas por tres tipos de capas: las capas convolucionales (CONV), las capas de pooling (POOL) y las capas totalmente conectadas (FC). En esta sección discutiremos cómo se apilan comúnmente estas capas para formar redes neuronales convolucionales completas, vamos a escribir de manera explícita la función de activación ReLU como una capa que aplica la no linealidad elemental.

La forma más común de una arquitectura de redes convolucionales está formada por unas cuantas capas CONV-RELU apiladas, a estas se le apilan capas POOL y se repite este patrón hasta que la imagen se fusiona espacialmente a un tamaño pequeño. Una vez hecho esto, se suele realizar una transición a capas totalmente conectadas, FC. La última capa totalmente

conectada contiene la salida. La arquitectura de redes convolucionales más común sigue el siguiente patrón:

$$\text{INPUT} \rightarrow [[\text{CONV} \rightarrow \text{RELU}]^N \rightarrow \text{POOL?}]^M \rightarrow [\text{FC} \rightarrow \text{RELU}]^K \rightarrow \text{FC}$$

donde POOL? indica una capa de agrupación opcional. Además, los valores de las constantes suelen ser, $N \geq 0$ (y normalmente $N \leq 3$), $M \geq 0$, $K \geq 0$ (y normalmente $K < 3$).

Algunas de las arquitecturas más comunes que se pueden ver y que siguen este patrón son:

- $\text{INPUT} \rightarrow \text{FC}$, implementa un clasificador lineal con $N = M = K = 0$.
- $\text{INPUT} \rightarrow \text{CONV} \rightarrow \text{RELU} \rightarrow \text{FC}$
- $\text{INPUT} \rightarrow [\text{CONV} \rightarrow \text{RELU} \rightarrow \text{POOL}]^2 \rightarrow \text{FC} \rightarrow \text{RELU} \rightarrow \text{FC}$. Aquí podemos observar que hay una sola capa de convolución entre cada capa de POOL.
- $\text{INPUT} \rightarrow [[\text{CONV} \rightarrow \text{RELU}]^2 \rightarrow \text{POOL}]^3 \rightarrow [\text{FC} \rightarrow \text{RELU}]^2 \rightarrow \text{FC}$. Aquí tenemos dos capas CONV apiladas antes de cada capa POOL. Esto suele ser una buena idea para redes más grandes y profundas, debido a que múltiples capas CONV apiladas pueden desarrollar características más complejas del volumen de entrada antes de la operación de pooling.

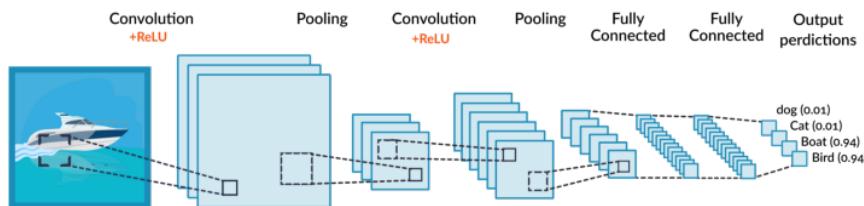


Figura 11.8.: Ejemplo de una arquitectura de red neuronal convolucional. [18]

La investigación acerca de las arquitecturas de redes convolucionales avanza muy rápido y cada pocas semanas o meses se anuncia una nueva arquitectura mejor para un punto de referencia determinado. Las mejores arquitecturas que existen actualmente se han compuesto sistemáticamente de los bloques de construcción descritos aquí. Entre las arquitecturas que existen en el campo de las redes neuronales que tienen un nombre destacamos las siguientes: LeNet, AlexNet, ZF Net, GoogLeNet, VGGNet, ResNet.

Parte IV.

Clasificación de Posiciones de Ajedrez

En esta parte del trabajo se explicará en qué consiste el problema de clasificación de posiciones de ajedrez propuesto y también se presentarán los avances que han habido en este campo. Posteriormente, se mostrarán una serie de modelos que han sido desarrollados para resolver el problema planteado gracias a los conocimientos adquiridos a lo largo del trabajo. Por último, se realizará una comparativa y análisis de los modelos propuestos, y se mostrará el beneficio que puede suponer en el mundo del ajedrez.

12. Descripción del problema

En este capítulo se pretende introducir el problema que se debe abordar, la clasificación de posiciones de ajedrez. En primer lugar se describirá el problema, donde se explicará el motivo y la importancia de este. Se presentará lo que es una posición de ajedrez para familiarizar al lector en lo que sigue. En la última parte del capítulo se describirá la base de datos que se ha empleado para el desarrollo del problema.

12.1. Descripción del problema

El ajedrez es uno de los juegos más antiguos y más fascinantes que se conocen y como tal, los seres humanos están interesados en encontrar mecanismos para incrementar el nivel de juego. Actualmente, existen los módulos de ajedrez pero no se sabe muy bien cómo evalúan las diferentes posiciones que se dan en una partida de ajedrez. Lo que sí se conoce es que emplean una unidad, los centipeones, pero no se entiende muy bien qué significa que por ejemplo, las blancas tengan 1,25 peones de ventaja (o puntos) con respecto a las negras. En resumidas cuentas, se desconoce de manera exacta cómo los motores miden estas ventajas y por eso, se ha realizado este trabajo.

El problema a tratar consiste en clasificar una serie de características de diferentes posiciones de ajedrez para ser capaces de analizarlas y extraer aquellas que se repiten con más frecuencia. Estas posiciones tienen la particularidad de que han sido escogidas justo antes de que se haya producido una mala jugada. Las posiciones de ajedrez escogidas se pueden dar en las diferentes fases del juego que existen (explicadas en la sección 3.5). En este caso, las posiciones seleccionadas se dan en partidas entre la jugada veinte y la jugada cincuenta.

En una posición de ajedrez hay muchas características que se pueden extraer, como por ejemplo, el bando que tiene ventaja de espacio, si un bando tiene la pareja de alfiles, si alguno tiene uno o varios peones pasados, si se encuentran ligados, etc. El objetivo consiste en contribuir para que un jugador mejore en su estudio y aprendizaje del ajedrez permitiendo que se centre en aquellas posiciones donde se equivoca y que entienda a qué se deben los errores producidos. Para ayudarlo a incrementar su nivel se extraen una serie de características de cada una de las posiciones de la base de datos y estas son las etiquetas que se han utilizado para entrenar las redes neuronales, permitiendo así ayudar al jugador a que sepa ciertas peculiaridades de posiciones donde escoge una mala decisión. De esta forma podrá profundizar su entrenamiento en este tipo de posiciones que son aquellas en las que pierde ventajas decisivas o termina perdiendo la partida.

Para enfrentarse al problema que se acaba de describir se ha recurrido a un subconjunto del aprendizaje automático que ha sido descrito a lo largo de este trabajo, el aprendizaje profundo, donde se han empleado redes neuronales de varios tipos. Esto permite que, a pesar de no ser un jugador profesional, en concreto un GM¹, ya que no se ha adquirido

¹En ajedrez GM significa Gran Maestro.

12. Descripción del problema

el suficiente conocimiento como para saber analizar de manera exhaustiva una posición de ajedrez, cualquier persona pueda desenvolverse y enfrentarse al problema sin tener la sabiduría de un experto.

12.2. Posición de Ajedrez

La posición de ajedrez describe cómo están colocadas las piezas en el tablero de ajedrez, tal como se imprime en un diagrama de ajedrez, en una imagen o en una fotografía de una partida de ajedrez. Un jugador no profesional al observar una posición verá una serie de piezas sobre el tablero pero no podrá darse cuenta de todas las conexiones que existen entre las diferentes piezas del tablero ni será capaz de percibir si hay alguna amenaza. Sin embargo, los grandes maestros sí son capaces en la mayoría de las ocasiones de ver más allá de lo que podemos observar los jugadores no profesionales y se percatan de la mayor parte de las peculiaridades de la posición, ya que en ocasiones la han analizado previamente con la ayuda de los módulos de ajedrez. La información de cualquier posición de ajedrez arbitraria que ocurra dentro del juego de ajedrez se puede determinar a partir de la posición inicial 3.4.4 y siguiendo una secuencia de movimientos, que conduce a esta posición. Cada posición tendrá unas particularidades y de todas ellas se pueden extraer una serie de características ya sean básicas o de más alto nivel. En 1996 Shirish Chinchalkar determinó el número 10^{46} como límite superior para el número de posiciones de ajedrez alcanzables [17]. El número máximo de movimientos por posición de ajedrez parece ser 218.



Figura 12.1.: Ejemplo de una posición de ajedrez. El último movimiento que se produjo en la posición fue el movimiento de la torre negra de a8 a c8 marcado en amarillo.

En la figura 12.1 se puede observar una posición de ajedrez. Esta posición es una de las diez mil que se han utilizado para desarrollar la aplicación web de nuestro trabajo. Ahora se va a explicar cómo se han extraído las diferentes posiciones para conformar la base de datos que se ha utilizado para la realización del trabajo y las 38 características que se han empleado para clasificar estas posiciones. Las 38 características serán las etiquetas que se necesitan para entrenar los modelos propuestos.

12.3. Base de Datos

En sus inicios, la base de datos estaba formada por las diez mil posiciones de ajedrez que aparecen en la página web, se puede encontrar en el primer apéndice A, y la idea consistía en que jugadores de todo el mundo pudieran clasificar estas posiciones para así poder obtener las etiquetas con las que entrenar los modelos. Debido a la falta de respuestas por parte de la comunidad se optó por otra vía que fue la creación de una serie de algoritmos que se describen en el apéndice B.1. También se aumentó la base de datos, ya que se consideró que con diez mil posiciones los modelos no serían capaces de aprender lo suficiente como para lograr buenos resultados.

La base de datos con la que se ha trabajado ha sido obtenida a partir de la base de partidas de Lichess [55] de la cual se han extraído más de tres millones de posiciones en formato CBV, que es un archivo creado por ChessBase [15]. Estas posiciones han sido obtenidas de más de ciento cincuenta mil partidas en las que los jugadores tienen más de 2200 puntos de Elo. Una vez extraídas estas posiciones se realizó una conversión de formato CBV a formato PGN, para ello fue necesario abrir las diferentes posiciones en formato CBV en *ChessBase Reader 2017* y ya se pudieron obtener las posiciones en formato PGN. Posteriormente, se convirtió el formato PGN al sistema de Forsyth-Edwards (FEN) explicado en la sección 3.7 y estas posiciones fueron almacenadas en un archivo CSV.

Una vez se consiguió tener las posiciones en el formato adecuado se realizaron una serie de cribas. El primer filtro que se llevó a cabo fue no tener en cuenta posiciones que se han dado antes de la jugada veinte y aquellas posteriores a la jugada cincuenta donde se producen a alto nivel la mayoría de los errores. Una vez estaban cribadas se seleccionaron las posiciones donde la diferencia entre la evaluación de la posición actual y la evaluación de la posición una vez se ha producido el movimiento es de medio punto (medio peón). De todas las posiciones que se obtuvieron, que fueron más de trescientas mil, se barajaron y se seleccionaron cincuenta mil posiciones y estas últimas serán las posiciones de nuestra base de datos. El resto de posiciones se reservaron para futuros experimentos. Estas posiciones se pasaron a formato con extensión *txt* para poder emplear los algoritmos que se han desarrollado para obtener las etiquetas y las codificaciones necesarias para entrenar los diferentes modelos. En la figura 12.2 viene un ejemplo de una posición de la base de datos que se ha utilizado con su correspondiente código FEN.

Por tanto, las cincuenta mil posiciones de la base de datos son posiciones que han sido extraídas de partidas entre la jugada veinte y la jugada cincuenta donde la siguiente jugada que se realizó fue una mala jugada pero sin ser una jugada nefasta. Las piezas de estas partidas fueron dirigidas por jugadores de más de 2200 puntos de Elo y se escogieron posiciones donde se produjeron errores tanto conduciendo las piezas blancas como las negras.

Para saber si se ha efectuado una mala jugada se ha evaluado la posición antes de realizar el movimiento y una vez se ha ejecutado. Para llevar a cabo esta evaluación se ha empleado el motor de ajedrez Stockfish y se ha estimado que una mala jugada se produce cuando la evaluación de la posición antes de efectuar el movimiento y la evaluación de la posición una vez se ha realizado se diferencian en más de medio punto en detrimento del bando que realizó el movimiento.

Para cada una de las diferentes posiciones de ajedrez se han seleccionado 38 características

12. Descripción del problema



Figura 12.2.: Ejemplo de una posición de ajedrez cuyo código FEN es:
 $3b1r1k/RQnq4/2p1n3/NpPp1p2/1P1PpPp1/2B1P1Pp/4B2P/7K$ b - - 4
 43; f8f7

y estas son las que se han empleado para entrenar nuestros modelos. Todas las etiquetas han sido extraídas gracias a los algoritmos que se han desarrollado. Las etiquetas se han clasificado en tres tipos, aquellas que son comunes, las que corresponden a las piezas blancas y las que corresponden a las negras. Hay que tener en cuenta que al ser una base de datos propia no está bien distribuida puesto que hay algunos casos en esas cincuenta mil posiciones que solo aparecen en unas pocas ocasiones y esto provocará que haya dificultades a la hora de entrenar las diferentes redes neuronales. Debido a esto, se ha intentado seleccionar un subconjunto de aquellas posiciones que más se suelen frecuentar en las partidas a este nivel, teniendo en cuenta por ejemplo que es muy poco frecuente que un bando tenga tres damas sobre el tablero. Una vez fueron seleccionadas las cincuenta mil posiciones había una única posición en la que las piezas negras tenían siete peones aislados. Por este motivo, al solo haber una con esta característica se optó por sustituir esa posición por otra del conjunto de posiciones que no se habían escogido, una de las más de trescientas mil posiciones que se habían reservado. Se van a presentar las etiquetas que se han utilizado, las cuales han sido extraídas de la página [83], así como, los respectivos valores que pueden tomar para la base de datos escogida y lo que significan.

- *Etiquetas comunes.*

- *Ventaja de espacio:* un bando tiene ventaja de espacio cuando tiene sus peones más avanzados. Posibles clases:
 - -1: ventaja de espacio negra.
 - 0: ningún bando tiene ventaja de espacio.
 - 1: ventaja de espacio blanca.
- *Columnas abiertas:* una columna se considera abierta cuando no hay ningún peón en ella. Contabilizamos el número de columnas abiertas que hay en la posición:
 - 0: no hay columnas abiertas.
 - 1-7: número de columnas abiertas, ya sea una que se codifica con un 1, dos, tres, y así hasta ocho, 8, que representa que todas las columnas están abiertas. En la base de datos que hemos considerado no aparecen posiciones con las

8 columnas abiertas, siempre hay al menos un peón en alguna de ellas. Esto se debe a que no hemos estudiado muchos finales de partida.

- *Alfiles del mismo color*: hay un alfil por bando y ambos se encuentran en las casillas blancas o en las negras. Posibles clases:
 - 0: no hay alfiles del mismo color.
 - 1: en la posición sí hay alfiles del mismo color.
- *Alfiles de distinto color*: cada bando cuenta con un alfil y uno es de casillas blancas y el otro de casillas negras. Posibles clases:
 - 0: no hay alfiles de distinto color.
 - 1: en la posición sí hay alfiles de distinto color.

■ *Etiquetas por bando*.

- *Columnas semiabiertas*: una columna está semiabierto si no hay peones de un bando en ella. Distinguiremos el número de columnas abiertas por bando:
 - 0: no hay columnas semiabiertas.
 - 1-6: número de columnas semiabiertas, puede haber una, dos, hasta seis. No hay ninguna posición en nuestra base de datos con siete u ocho columnas semiabiertas por parte de un bando.
- *Peones*: número de peones. Posibles clases:
 - 0: el bando no tiene peones en esa posición.
 - 1-8: en la posición hay uno, dos o hasta ocho peones de ese bando.
- *Caballos*: número de caballos. Posibles clases:
 - 0: el bando no tiene caballos en esa posición.
 - 1: en la posición hay un caballo.
 - 2: el bando correspondiente dispone de dos caballos.
- *Alfiles*: número de alfiles. Posibles clases:
 - 0: si ese bando no tiene alfiles.
 - 1: si dispone de uno.
 - 2: si tiene la pareja de alfiles.
- *Torres*: número de torres. Posibles clases:
 - 0: si ese bando no tiene torres.
 - 1: si dispone de una.
 - 2: si conserva ambas torres.
- *Damas*: número de damas. Posibles clases:
 - 0: si ese bando ya no tiene dama.
 - 1: si conserva su dama.
 - 2: cuando el bando tiene dos damas en juego.

12. Descripción del problema

- *Torre en séptima*: si una torre se encuentra en la séptima fila para las blancas o en la segunda para las negras. Posibles clases:
 - 0: si no hay ninguna torre en séptima.
 - 1: si hay alguna torre en séptima.
- *Torres dobladas*: torres que se encuentran conectadas en la misma columna. Posibles clases:
 - 0: si las torres no se encuentran dobladas.
 - 1: si las torres están dobladas.
- *Torres ligadas*: torres conectadas en la misma fila sin que haya ninguna pieza entre ellas. Posibles clases:
 - 0: si las torres no se encuentran ligadas.
 - 1: si las torres están ligadas.
- *Pistola de Alekhine*: batería de tres piezas en vertical, formada por dos torres y la dama. Posibles clases:
 - 0: si no está conformada la estructurada.
 - 1: si las torres y la dama forman la pistola de Alekhine.
- *Peones doblados*: dos peones del mismo color en la misma columna. Las clases se corresponden con el número de peones doblados que tiene cada bando:
 - 0: si no hay ningún peón doblado.
 - 1: si hay un peón doblado.
 - 2: si hay dos peones doblados.
 - 3: si las tres peones doblados.
- *Peones aislados*: peón que no tiene peones de su mismo color en columnas adyacentes. Las clases se corresponden con el número de peones aislados que tiene cada bando:
 - 0: si no hay peones aislados.
 - 1-6: si hay un peón aislado, dos, tres, hasta seis. Comentar que se dan cinco o seis peones aislados cuando se encuentran doblados.
- *Peones retrasados*: un peón retrasado es aquel que no puede ser protegido por peones de su bando, puesto que los peones que lo flanquean han sido avanzados más allá del peón retrasado. Las clases que puede tomar la etiqueta es el número de peones retrasados que tiene cada bando:
 - 0: si no hay peones retrasados.
 - 1-4: si hay un peón retrasado, dos, tres, hasta cuatro.
- *Peones pasados*: peones que no pueden ser detenidos por peones del otro bando. Las clases se corresponden con el número de peones pasados que tiene cada bando:
 - 0: si no hay peones pasados.

- 1-6: si hay un peón peón pasado, dos, tres, hasta seis.
- *Islas de peones*: cada grupo de peones (al menos uno), separados por columnas sin peones propios. Las posibles clases son el número de islas de peones que tiene cada bando:
 - 0: si un bando no tiene islas de peones significa que no conserva peones.
 - 1-4: si hay una isla, dos, tres o cuatro.
- *Falanges de peones*: dos o más peones del mismo bando adyacentes en la misma fila. Las clases se corresponden con el número de grupos de falanges de peones que tiene cada bando:
 - 0: si no hay falanges de peones.
 - 1: si hay una falange.
 - 2: si hay dos falanges.
 - 3: si hay tres falanges.
- *Peones conectados*: dos o más peones del mismo bando en columnas adyacentes. Contabilizamos el número de grupos de peones conectados por bando:
 - 0: si no hay peones conectados.
 - 1: si hay un grupo de peones conectados.
 - 2: si hay dos grupos de peones conectados.
 - 3: si hay tres grupos de peones conectados.

Se puede observar que todas las etiquetas adquieren datos enteros, esto es así porque se va a clasificar cada uno de los valores de las etiquetas como datos categóricos para que sea más fácil realizar la clasificación. Antes de pasar con las tablas resumen se va a presentar una tabla donde se pone de manifiesto lo que se ha descrito sobre el desbalanceo de clases. Este desbalanceo se podría haber intentado mejorar pero se dejará para un trabajo futuro. Por ejemplo, para el caso de la etiqueta del número de damas blancas:

Tipo	Cantidad de filas en la base de datos
Ninguna dama	24558
Una dama	25428
Dos damas	14

Tabla 12.1.: Datos de la etiqueta de damas blancas.

En la tabla que se acaba de presentar el número de posiciones donde aparecen dos damas blancas es insignificante y con total seguridad se tendrán problemas a la hora de realizar la clasificación, esto se ha tenido en cuenta a la hora de realizar el análisis. Por último, una vez hemos expuesto las diferentes etiquetas empleadas para entrenar los modelos, así como un ejemplo de tabla donde se recogen los datos de una de ellas, se van a presentar dos tablas resumen con cada una de las etiquetas y sus correspondientes valores. En la primera tabla aparecen las etiquetas comunes y en la segunda las etiquetas de las piezas blancas y

12. Descripción del problema

negras que son las mismas pero hay dos de cada tipo, una por bando. Las posibles clases de estas últimas etiquetas son las mismas independientemente del color de las piezas.

Etiqueta	Clase
Ventaja de espacio	-1, 0, 1
Columnas abiertas	0-7
Alfiles del mismo color	0,1
Alfiles de distinto color	0,1

Tabla 12.2.: Tabla con las etiquetas comunes y sus respectivos valores.

Etiqueta	Clase
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

Tabla 12.3.: Tabla con las etiquetas de las piezas blancas y negras con sus respectivos valores.
Estas etiquetas aparecen una vez por cada bando.

13. Estado del Arte

En este capítulo se presenta cómo ha ido evolucionando este campo a lo largo de la historia para que el lector pueda saber y conocer cómo se encuentra actualmente. En primer lugar, se expondrá un caso experimental que se asemeja en cierta medida con el estudio que se ha realizado. Posteriormente, se repasará la evolución que se ha realizado desde principios del siglo XX hasta nuestros días en diferentes áreas relacionadas con el ajedrez, se verá su conexión con la inteligencia artificial y se destacará la parte donde se interconectan el ajedrez y el aprendizaje profundo. El capítulo finalizará con una breve reflexión.

13.1. Estado del Arte

Para finalizar este capítulo se va a dar un repaso breve por la historia para comprobar cómo ha avanzado este campo y cómo de desarrollado se encuentra. Se va a realizar una revisión de trabajos e investigaciones sobre el ajedrez. En primer lugar, se presentarán las primeras investigaciones que se realizaron que tratan de la psicología cognitiva y posteriormente se revisarán trabajos más actuales que emplean técnicas de aprendizaje profundo y visión por computador para reconocer y clasificar piezas sobre el tablero, evaluar posiciones, analizar posibles movimientos, etc. Para poder llevar a cabo esta búsqueda se ha utilizado la base de datos bibliográfica Scopus y también Google Scholar, para poder ampliar el campo de búsqueda y encontrar documentos académicos.

Buscando en la literatura existente aparecen una gran cuantía de artículos y documentos relacionados con el ajedrez, pero los documentos del tema que a tratar son escasos. En primer lugar, se han examinado aquellos artículos que pueden tener un fin similar o sean próximos a este estudio. En concreto se describirán cuatro artículos.

Linhares y Brum presentan en 2007 [57] un experimento realizado durante seis meses en Río de Janeiro debido a que existía un debate sobre la naturaleza de las posiciones de ajedrez. Una posibilidad afirmaba que las posiciones se construían codificando combinaciones de piezas en las diferentes casillas, lo que se conoce en inglés como *POSs* que son piezas en cuadrados, y querían comprobar cómo los jugadores de ajedrez son capaces de percibir una gran similitud entre posiciones muy diferentes.

Este experimento consistió en seleccionar un conjunto diverso de jugadores de ajedrez que se dividió en dos grupos, uno formado por novatos y otro por jugadores con mayor nivel para que emparejaran veinte posiciones de ajedrez. Estas veinte posiciones tenían la particularidad de que la disposición de las piezas según Linhares y Brum eran muy distintas pero se podían agrupar formando diez emparejamientos. Para familiarizarse con las posiciones los jugadores tuvieron que clasificar las posiciones en función de si las blancas ganaban, empataban o perdían y de los movimiento sugeridos.

Los resultados que obtuvieron fueron que el 53 % coincidieron en las veinte posiciones que se esperaba según la teoría, aunque sí que es cierto que se observó mucha diferencia

ya que en el grupo de principiantes, aproximadamente el 20 % realizó el emparejamiento de forma adecuada frente al casi 75 % de los expertos. Concluyeron que existen múltiples niveles de codificación de posiciones de ajedrez desde representaciones superficiales de relaciones de piezas concretas hasta representaciones semánticas o conceptuales abstractas. Además, destacaron que los jugadores expertos prefieren agrupar las posiciones de ajedrez por similitudes abstractas que no se pueden explicar de manera visual.

Bilalić y Gobet en 2009 [5] contestan a Linhares y Brum exponiendo que la metodología que utilizaron en su experimento usando similitud abstracta es inadecuada para sacar conclusiones sobre la naturaleza de la percepción de los expertos de las posiciones de ajedrez. Ellos argumentaron que en el experimento se les pidió a los jugadores que igualaran posiciones de ajedrez en parejas, a través de instrucciones explícitas para buscar similitudes de visión estratégica. Lo que querían demostrar era que las instrucciones explícitas sesgaron a los jugadores y estos respondieron como quería el experimentador.

En este artículo Bilalić y Gobet presentaron un experimento en el que usan las mismas veinte posiciones que usaron previamente Linhares y Brum en [57] y en el que participaron doce expertos y diez principiantes. La diferencia con respecto al anterior experimento es que para agrupar posiciones deben usar una similitud concreta. Así, todos los jugadores para clasificar las posiciones tenían, además de buscar similitudes en un nivel abstracto y estratégico como hicieron Linhares y Brum una segunda condición que era buscar emparejamientos basados en similitudes concretas de superficie, es decir, buscar similitudes a nivel superficial o de apariencia.

Los resultados que obtuvieron mostraron que los expertos sí son mejores que los novatos para encontrar analogías profundas pero cuando relacionan problemas con base a una similitud superficial la diferencia que hay entre expertos y novatos fue mínima.

En 2009 Linhares y Brum [58] contestaron a Bilalić y Gobet poniendo de manifiesto la asimetría básica entre expertos y novatos, ya que los novatos no se pueden comportar como expertos pero los expertos sí como novatos. Además, argumentaron que el experimento original estaba destinado a demostrar que los expertos son capaces de agrupar las posiciones por similitud abstracta, mientras que los novatos solo se fijan en la similitud superficial y así mostraban la gran diferencia entre unos y otros. Terminaron mencionando que los expertos adquieren la habilidad de codificar posiciones en un nivel abstracto que incluye analogías y la percepción fluida de los roles abstractos que juegan las piezas.

Pablo y Frank replican en 2013 [54] el experimento llevado a cabo por Linhares y Brum en 2007 [57] con instrucciones menos explícitas. Así podrían comprobar si se fijaban más en la similitud abstracta, en la similitud visual o en otra noción de similitud y Linhares y Brum tendrían una evidencia más sólida de sus afirmaciones. Además, recolectan etiquetas (algunas de las cuales han sido empleadas en nuestro estudio) para categorizar las posiciones y explorar los principios abstractos que usaron los participantes.

En este artículo los autores mencionan estar de acuerdo con Linhares y Brum y piensan que Bilalić y Gobet tienen un punto válido. También destacan que el experimento original solo muestra que los jugadores expertos pueden agrupar por similitud abstracta si se les indica que lo hagan pero esto no prueba que notar similitudes abstractas y hacer analogías sea crucial en ajedrez como afirmaban Linhares y Brum.

Para realizar el experimento se reclutaron a treinta participantes dividiéndolos en dos grupos según su calificación DWZ (adaptación del sistema de clasificación Elo para la federación

alemana de ajedrez) aunque esta división no fue la ideal al haber en ambos grupos jugadores alrededor del promedio alemán. A los participantes, en una primera fase, les presentaron las veinte posiciones de manera aleatoria y se les pidió que dieran una evaluación de la posición y si eran las blancas o las negras las que disponían de una mejor posición, si la posición era igualada o si no era posible ninguna evaluación. Además, se les solicitó que efectuaran el próximo movimiento de las blancas. En la segunda fase debían agrupar las veinte posiciones de ajedrez de la anterior fase. Algunos jugadores preguntaron los criterios para interpretar la similitud de las posiciones a lo que se le respondió con que eso dependía de ellos mismos. Los jugadores debían nombrar un tema para cada pareja seleccionada, así como atributos o características que les había hecho seleccionar el par y calificar dicha similitud.

Para comparar las etiquetas tuvieron que idear un sistema de clasificación jerárquico y posteriormente asignaron las etiquetas y agregaron algunas clases adicionales según las etiquetas de los participantes.

A la conclusión que llegaron es que los jugadores más experimentados eligieron pares considerablemente más abstractos aunque en menor medida que en los estudios anteriores y que los pares elegidos por los novatos fueron agrupados más frecuentemente por similitud visual. Asimismo, aunque los novatos agruparon alguna pareja por similitud abstracta pudieron comprobar que los expertos tenían una tendencia a usar categorías mejor diferenciadas.

Una vez presentados estos artículos que se asemejan en cierta medida con el trabajo que se ha realizado aunque distan mucho del propio, se van a introducir los inicios de la investigación relacionada con el ajedrez. Históricamente, el ajedrez ha sido uno de los principales campos en el estudio de la experiencia. Esto es debido, entre otras cosas, a las ventajas que ofrece el ajedrez para estudiar los procesos cognitivos, al carácter bien definido de la tarea, a la presencia de una escala cuantitativa para clasificar a los ajedrecistas y a la fertilización cruzada con investigación sobre juegos en informática e inteligencia artificial.

Muchos de los conceptos y mecanismos clave que se desarrollaron más tarde en la psicología cognitiva fueron avanzados por Adriaan de Groot en su libro cuyas dos versiones son: [29] y [30].

De Groot realizó experimentos de ajedrez con jugadores de diversas categorías, desde principiantes a grandes maestros como Alekhine, Max Euwe, Reuben Fine, etc. El objetivo del estudio consistía en explicar cómo los expertos son capaces de entender en unos pocos segundos la situación actual en el tablero, encontrar ideas constructivas para llevar a cabo y descubrir buenas jugadas.

En los experimentos que realizó, los participantes miraban una posición de ajedrez y comentaban en voz alta sus pensamientos. De Groot descubrió que era durante los primeros segundos cuando los grandes maestros tienen las mejores ideas según una posición concreta, el proceso de pensamiento lo definió en cuatro fases:

- *Fase de orientación*: donde los jugadores capturan la posición y formulan ideas generales sobre lo que hacer.
- *Fase de exploración*: se analizan las posibles variaciones.
- *Fase de investigación*: deciden qué movimiento creen que es el mejor.
- *Fase de control*: fase en la que el jugador comprueba la validez de la elección que realizó en la fase anterior.

De Groot se centró en el papel de la percepción visual y la memoria en las fases descritas y cómo los jugadores, en general grandes maestros, pueden echar mano de posiciones vistas o estudiadas anteriormente para facilitar el proceso descrito previamente.

También realizó otro tipo de experimentos con otros autores como el que llevó a cabo junto a Gobet en 1996 [31] en el que mostraron posiciones del tablero durante pocos segundos y vieron si los jugadores de diferentes niveles podían memorizarlas. Descubrieron que los maestros y grandes maestros conseguían memorizar un 93 % de las posiciones, los expertos un 72 % y los principiantes tan solo un 51 %.

Las ideas teóricas de De Groot que se basaron en la psicología de Otto Seilz no tuvieron tanta relevancia como sus técnicas y resultados empíricos. Unos veinticinco años más tarde la investigación del ajedrez con la teoría de la fragmentación, obra de Chase y Simon en 1973 [14], produjo un fuerte impacto en el estudio de la pericia y de la psicología.

Posteriormente se fueron realizando experimentos entre jugadores no profesionales y jugadores de alto nivel donde se quería comprobar qué llevaba a los jugadores expertos a tener mayor capacidad para reproducir posiciones tras haberlas observado durante unos pocos segundos o de qué manera percibían estas posiciones, todo para seguir investigando y avanzando en el conocimiento de la percepción visual y la memoria. Algunos de estos trabajos son: [80], [37], [59], [73], [40], [41].

Uno de los resultados clave que se obtuvo con el paso de los años fue que los ajedrecistas expertos cuando están frente a posiciones que no han visto anteriormente son capaces de reconocer patrones en la posición y relacionarlos con otros patrones de posiciones que sí conocen. Esto les permite poder tener cierto conocimiento de la posición a diferencia de jugadores principiantes e incluso pueden clasificar posiciones que a un jugador normal le parecen muy diferentes pero sin embargo tienen patrones que las hacen realmente similares. Este tipo de patrones son los que queremos obtener clasificando las posiciones de nuestra base de datos para saber en qué tipo de posiciones debemos profundizar para mejorar nuestro nivel; en nuestro trabajo estos patrones no serán muy complejos.

Ahora se mostrarán algunos de los avances que se han producido en este campo al emplear inteligencia artificial. La mayoría de los trabajos que se han llevado a cabo en el área de la investigación relacionada con el ajedrez empleando métodos de aprendizaje automático o profundo se basan en evaluar una posición para saber si un bando tiene ventaja, si es así cuánta ventaja tiene, cuál es la mejor jugada a realizar, si se puede realizar mate forzado, etc. Así, se han podido construir y crear máquinas capaces de jugar al ajedrez, es decir, se han entrenado redes neuronales sin conocimiento previo de ajedrez y han conseguido ser capaces de jugar, lo que se conoce como un programa para jugar al ajedrez. Ya se mencionó en el capítulo 5 los elementos que necesita un motor de ajedrez para ser capaz de aprender a jugar e ir mejorando su nivel. Estos avances emplean en la mayoría de los casos aprendizaje profundo que se apoyan en el uso de redes neuronales y, sobre todo, redes neuronales convolucionales.

Ahora se presentan algunas referencias y artículos que han sido encontrados en la literatura que han servido para progresar en este campo y que se pueden buscar para aprender a crear una red que juegue al ajedrez. Una de las primeras publicaciones fue escrita por Shannon [76] donde se buscaba crear una computadora que era capaz de jugar al ajedrez. Con el paso de los años fueron apareciendo otros artículos y libros, entre la literatura existente destaca el artículo [86] donde se muestra cómo entrenar una red neuronal para jugar al ajedrez o [61]

donde se presenta un nuevo enfoque de aprendizaje supervisado para el entrenamiento de redes neuronales artificiales para evaluar posiciones de ajedrez. También se destacan estas otras referencias, [27] donde se presenta un método de aprendizaje integral para el ajedrez basado en redes neuronales profundas y el artículo [68] que usa una red neuronal de tres capas para predecir movimientos en ajedrez. Además, existen documentos como el libro [51] donde se ofrece una completa introducción al funcionamiento técnico de algunos motores de ajedrez, el trabajo [84] presenta NeuroChess, un programa que aprende a jugar al ajedrez a partir del resultado final de las partidas. Con el paso de los años, incluso se han desarrollado algoritmos genéticos para evolucionar estos programas como el que aparece en [28]. Gracias a estos progresos, entre otros tantos, actualmente se puede jugar contra máquinas y aprender de ellas, ya que estas han adquirido un conocimiento el cual no puede ser alcanzado por el ser humano.

A lo largo de los años también se ha abierto una vía de investigación para reconocer el tablero y las piezas que se sitúan sobre el tablero en tiempo real utilizando una rama de la inteligencia artificial, la visión por computador. Algunos de los trabajos que se han realizado y los que han ayudado a avanzar y a profundizar en este campo son: [89] o [66]. Otros trabajos han empleado redes neuronales para ayudar al reconocimiento de los cuales destacamos uno realizado hace unos años [22], donde se diseña un método de reconocimiento del tablero y de detección de piezas que es resistente a las condiciones de iluminación y al ángulo de captura de las imágenes, y que funciona correctamente con numerosos estilos de tablero de ajedrez.

Una vez se ha realizado un repaso por las principales áreas de investigación relacionadas con el ajedrez que se han desarrollado a lo largo de la historia, se puede apreciar que no hay una línea de investigación clara a partir de la cual se pueda continuar para realizar este trabajo, ya que la literatura existente es prácticamente nula por lo que se puede decir que es algo bastante novedoso. Sí es cierto que, se busca realizar una clasificación de patrones que se podría asemejar en cierta medida con algunas investigaciones que se realizaron durante el siglo XX y a principios del siglo XXI en el tema de localización de patrones, pero ahora se va a utilizar para ello aprendizaje profundo. Por tanto, se va a abrir una nueva línea empleando estas dos vertientes, la cual tiene todavía mucho camino por recorrer.

14. Metodología

En este capítulo se ha descrito en detalle la metodología que se ha empleado para desarrollar un software basado en el aprendizaje profundo para la clasificación de ciertas características que aparecen en una posición de ajedrez y facilitar el análisis de esta. Este software se puede encontrar en GitHub (ver apéndice B). En primer lugar, se van a explicar las herramientas que se han utilizado para desarrollar el software. Después, se presentan las métricas que se han manejado y cómo se han seleccionado los modelos. Posteriormente, se explicarán tres formas diferentes que se han utilizado para representar una posición de ajedrez y por cada representación se mostrarán los modelos escogidos para resolver el problema. Por último, se explicará el proceso de entrenamiento que se ha llevado a cabo.

14.1. Herramientas Software

El software que se ha desarrollado ha sido implementado gracias a Python, en concreto, se ha empleado la versión 3.9.12. Python es un lenguaje de alto nivel de programación interpretado, es un lenguaje orientado a objetos que destaca por su flexibilidad, sencillez y legibilidad. Además, dispone de multitud de bibliotecas para el análisis de datos con las que trabajar. Se ha utilizado TensorFlow junto con Keras para la implementación de los modelos. Keras es una API diseñada para trabajar con redes neuronales artificiales y es de código abierto. Está desarrollada en Python y se puede ejecutar sobre diferentes plataformas, la plataforma escogida ha sido Tensorflow. Keras está diseñado para ir construyendo por bloques la arquitectura de cada red neuronal. Tensorflow es una plataforma de código abierto que ayuda a los desarrolladores a crear, lanzar y gestionar aplicaciones de aprendizaje automático, mediante distintas herramientas, bibliotecas de código integradas y recursos comunitarios. La versión de Tensorflow que se ha utilizado es la versión 2.4.1.

Otras bibliotecas de Python que se han utilizado para el desarrollo de nuestro software han sido:

- *Numpy* 1.21.5: está especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos.
- *SciPy* 1.7.3 : está compuesta de herramientas y algoritmos matemáticos.
- *Matplotlib* 3.5.2: sirve para crear visualizaciones estáticas, animadas e interactivas en Python.
- *Scikit-learn* 1.1.1: es un módulo de Python para el aprendizaje automático construido sobre SciPy.
- *Pandas* 1.4.4: es una herramienta especializada en el manejo y análisis de estructuras de datos. Es de código abierto, rápida, potente, flexible y fácil de usar, construida sobre el lenguaje de programación Python.

14.2. Métricas

En esta sección se van a presentar las métricas utilizadas para evaluar el rendimiento de los distintos modelos que se han empleado para clasificar las etiquetas extraídas de las posiciones de ajedrez. En general, las métricas son un número que se encuentra en el rango de 0 a 1 donde un valor más alto representa un mejor resultado. Existen una gran cantidad de métricas que evalúan el rendimiento de los modelos y unas son más adecuadas que otras dependiendo del problema. En este caso, son varios los problemas de clasificación que hay que resolver en los cuales no es más importante una clase que otra sea cual sea la etiqueta con la que se esté trabajando. Se han empleado dos métricas, la métrica *Accuracy* que es de las más utilizadas en los problemas de clasificación y la métrica *F1-score* también conocida como *F1*. Antes de explicar estas métricas se presenta la matriz de confusión. Esta matriz tiene dimensión $n \times n$ donde n es el número de clases de la etiqueta que estamos evaluando. La celda (i, j) representa el número de casos de la clase i que han sido predichos como elementos de la clase j . Por tanto, es en la diagonal de la matriz donde aparecen los elementos perfectamente clasificados y es lo que buscamos maximizar en la práctica. Por el contrario, en el resto de posiciones tendremos casos mal clasificados, un ejemplo aparece en la figura 15.4.

La primera métrica que se explicará será la métrica *Accuracy*. Esta métrica es muy intuitiva y natural, es la medida más directa de la calidad de los clasificadores, generalmente describe el rendimiento del modelo en todas las clases. Es útil cuando todas las clases tienen la misma importancia como ocurre en este caso. Se calcula como la relación entre el número de predicciones correctas y el número total de predicciones, se trata de un valor entre 0 y 1 y cuanto más alto sea mejor será el modelo. Por tanto, la métrica *Accuracy* se calcula de la siguiente forma:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

donde

- *TP (Verdadero Positivo)*: son valores que el algoritmo clasifica como positivos que son realmente positivos.
- *TN (Verdadero Negativo)*: son valores que el algoritmo clasifica como negativos y que realmente son negativos.
- *FP (Falso Positivo)*: son valores que el algoritmo clasifica como positivos cuando realmente son negativos.
- *FN (Falso Negativo)*: son valores que el algoritmo clasifica como negativos cuando realmente son positivos.

El problema que tiene esta métrica es que cuando la base de datos con la que trabajamos está desbalanceada como sucede en este caso, se producen resultados que aparentemente son muy buenos cuando realmente no lo son. Ahora se presenta la segunda métrica utilizada pero antes se deben introducir dos métricas:

- *Precision*: esta métrica permite conocer la relación entre las observaciones positivas predichas correctamente y el total de observaciones positivas predichas, es decir, qué

porcentaje de valores que se han clasificado como positivos son realmente positivos.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- *Recall*: esta métrica se conoce como el ratio de verdaderos positivos, es utilizada para saber cuantos valores positivos son correctamente clasificados.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Una vez han sido explicadas estas dos métricas se procede a explicar la métrica F1-score que es la media ponderada de Precision y Recall. Por tanto, esta puntuación tiene en cuenta tanto los falsos positivos como los falsos negativos, la fórmula queda como sigue:

$$F1 = \frac{2TP}{2TP + FP + FN} = 2 \frac{\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}$$

El motivo por el que se ha escogido esta métrica es que en la base de datos hay etiquetas cuyas clases están muy desbalanceadas, como es el caso del número de damas ya sean blancas o negras. Esta métrica permite subsanar este desbalanceo entre clases otorgando la misma importancia a cada una de ellas. Ahora bien, con esta métrica se obtienen múltiples puntuaciones F1 por clase y sería mejor promediarlas para obtener un único número que describa el rendimiento global.

Ahora se presentan los métodos para promediar esta métrica que han sido utilizados:

- *Macro Average*: este método es quizás el más sencillo entre los numerosos métodos de promediación que existen. Esta puntuación se calcula utilizando la media aritmética (también conocida como media no ponderada) de todas las puntuaciones F1 por clase. Este método trata a todas las clases por igual independientemente de si las clases están desbalanceadas.
- *Weighted Average*: esta puntuación es la media ponderada de F1. Se calcula tomando la media de todas las puntuaciones F1 por clase y teniendo en cuenta el soporte de cada clase. El soporte se refiere al número de ocurrencias reales de la clase en el conjunto de datos. Así, la media ponderada tiene en cuenta la contribución de cada clase ponderada por el número de ejemplos de esa clase.

Hay otro método para promediar la métrica F1, *Micro Average*, esta calcula esencialmente la proporción de observaciones correctamente clasificadas de entre todas las observaciones siendo esta la definición de la métrica Accuracy que se ha explicado previamente. Por tanto, se podría decir que la métrica considerada ha sido la métrica F1 la cual la vamos a promediar de tres formas distintas donde una de ellas se corresponde con la métrica Accuracy.

14.3. Selección de los Modelos

Hay diferentes métodos para seleccionar un modelo, uno de los más preferidos es el método de validación cruzada, más conocido como *cross-validation*. En este caso, se ha empleado otro

14. Metodología

método que se conoce como *hold-out*. La elección de este método por delante de validación cruzada se debe a una serie de consideraciones, en primer lugar se cuenta con una gran cantidad de etiquetas que se deben clasificar, donde cada etiqueta por separado no supone mucho problema pero al tener clases desbalanceadas las clasificaciones de estas se complican. Si se emplea el método de validación cruzada como se ha probado para algún caso particular el proceso se complica bastante, ya que se necesitan muchos más recursos y mucho más tiempo para obtener los resultados. Estos resultados que se obtendrían no serían mucho mejores que los obtenidos mediante *hold-out* donde como se verá son muy buenos para algunas etiquetas y muy pobres para etiquetas donde las clases se encuentran muy desbalanceadas.

El método de *hold-out* para el entrenamiento de un modelo es el proceso que consiste en dividir los datos en diferentes conjuntos y utilizar una parte para entrenar el modelo, otra para validar y otra para probarlo. Este método se utiliza tanto para la evaluación como para la selección del modelo. Cuando se emplean todos los datos para entrenar el modelo utilizando diferentes algoritmos queda el problema de evaluar los modelos y seleccionar el más óptimo. La tarea principal es averiguar qué modelo de entre todos los modelos tiene el menor error de generalización, en otras palabras, qué modelo tiene mejor rendimiento en datos no vistos. Aquí es donde entra en escena *hold-out*, ante la necesidad de contar con algún mecanismo en el que el modelo sea entrenado en un conjunto de datos y probado en otro conjunto. Veamos el proceso que se lleva a cabo para seleccionar el modelo:

1. Dividir el conjunto de datos en tres partes: conjunto de datos de entrenamiento, conjunto de datos de validación y conjunto de datos de prueba.
2. Entrenar diferentes modelos utilizando diferentes algoritmos de aprendizaje.
3. Para los modelos entrenados con diferentes algoritmos ajustar los hiperparámetros y obtener diferentes modelos.
4. Probar el rendimiento de cada uno de estos modelos en el conjunto de datos de validación.
5. Seleccionar el modelo más óptimo de los modelos probados en el conjunto de datos de validación. El modelo más óptimo tendrá la configuración de hiperparámetros más óptima para el problema en cuestión.
6. Probar el rendimiento del modelo más óptimo en el conjunto de datos de prueba

En este caso, se ha seguido lo que recomiendan en la literatura, y el conjunto de entrenamiento estará formado por el 80 % del conjunto total de posiciones que disponemos y el 20 % restante será el que conforma el conjunto de test. El conjunto de entrenamiento se dividirá en dos conjuntos, uno que será el que utilicen los modelos para entrenar, que estará formado por el 80 % de los datos de entrenamiento y el 20 % restante es el que formará el conjunto de validación.

Una vez se ha expuesto cómo se han seleccionado los modelos se van a presentar los modelos propuestos. Se han propuesto tres modelos distintos, uno por cada representación del tablero que se ha considerado. Cada modelo se ha empleado para entrenar las 38 etiquetas, por lo que para cada una de ellas contará con una distribución de pesos distinta. El primer modelo tiene como datos de entrada siete canales de matrices 8x8, el segundo un vector unidimensional con los siete tableros y el tercero es el que emplea una única matriz donde se representa la posición dándole valores positivos a las piezas blancas, negativos a las negras y el valor neutro a las casillas vacías.

14.4. Modelo Primera Representación (Representación Tridimensional)

Para cada modelo se presentarán los hiperparámetros utilizados. Estos en todos los casos fueron escogidos en base a un proceso experimental. Las arquitecturas de los modelos escogidos las presentaremos en formato de tabla una vez se haya explicado cada conjunto de modelos con sus respectivos hiperparámetros. En estas tablas se podrá observar el tipo de capas que componen el modelo (capas convolucionales o densas), el número de unidades que componen las capas de convolución y las capas densas, el tamaño de entrada que dependerá del modelo y el tamaño de salida que variará dependiendo de la etiqueta seleccionada. Cada una de las tablas que se presentan por modelo representa el caso de etiquetas con tres clases como es el caso de la ventaja de espacio, para el resto de casos como la clasificación de peones blancos la única diferencia aparecerá en la última de las capas de cada arquitectura variando por tanto el número de parámetros totales.

Las funciones de activación que se han utilizado en los tres modelos han sido dos, la más usada ha sido la función de activación ReLU (es una de las más aconsejadas por la literatura) que será la que se emplee en todas las capas de los tres modelos menos en las capas de salida. En las capas de salida se ha utilizado la función softmax, ya que soporta sistemas de clasificación multinomial, siendo la más empleada en las capas de salida de un clasificador.

14.4. Modelo Primera Representación (Representación Tridimensional)

14.4.1. Primera Representación

Esta representación proviene de la literatura, en concreto, se puede encontrar en numerosos artículos y libros, algunos de ellos son [68] o [27]. Se sabe que en una posición de ajedrez hay seis tipos de piezas, reyes, damas, torres, alfiles, caballos y peones, para cada una de ellas se crea un tablero de tamaño 8×8 , en total seis. En cada tablero se representarán únicamente las piezas correspondientes a ese tablero, donde las piezas blancas vendrán codificadas con el valor 1 y las piezas negras con -1, el resto de casillas del tablero aparecerán codificadas con el valor cero. Por ejemplo, si se representan las torres en su correspondiente tablero, si en la posición aparecen torres blancas entonces las casillas donde se encuentren toman el valor 1, las casillas donde estén situadas las torres negras se codifican como -1 y el resto de casillas sin importar si hay piezas o no reciben el valor 0. Por tanto, la representación está formada por seis tableros de tamaño 8×8 pero tiene en cuenta otro tablero adicional del mismo tamaño para representar el turno. Si le toca mover a las blancas entonces las sesenta y cuatro casillas del tablero tienen como valor 1 y si le toca mover a las negras entonces reciben el valor -1. Este último tablero es muy importante, sobre todo para evaluar la posición. Podemos ver un ejemplo en la figura 14.1.

En resumen, la idea es utilizar un canal independiente para cada tipo de pieza, por tanto, cada posición está representada por seis canales en una cuadrícula 8×8 donde cada canal solo codifica información sobre las piezas de un tipo determinado. El color de las piezas se codifica con un signo. Así que, los canales son matrices dispersas de 8×8 con valores en el conjunto $\{-1, 0, 1\}$. Esta representación se amplia añadiendo un séptimo canal que codifica la información sobre quién hace el siguiente movimiento.

Por tanto, la primera representación que se ha utilizado queda formada por siete matrices

de tamaño 8×8 (una matriz tridimensional de tamaño $8 \times 8 \times 7$) donde aparecen seis canales de cuadrículas 2D de 8×8 con valores en el conjunto $\{-1, 0, 1\}$. El primero de los canales estará formado por el tablero de los reyes, el segundo por el de las damas, el tercero por el de las torres, el cuarto por el tablero de los alfiles, el quinto por el de los caballos y el sexto por el de los peones. A estos canales se le añade uno adicional formado por unos si las blancas mueven o menos unos si mueven las negras. Esta representación es la más popular y es la que utilizan los motores de ajedrez, aunque estos emplean además otros muchos tableros como el de posibles movimientos, casillas ocupadas por piezas propias, por piezas contrarias, etc. Para la realización del presente documento no hacen falta más tableros y la representación descrita es más que suficiente.

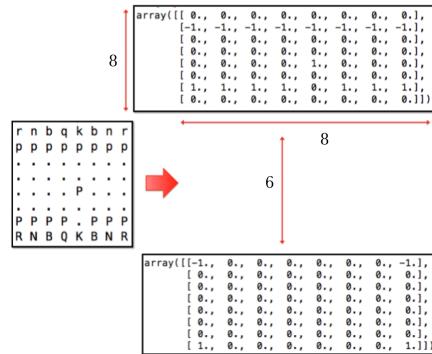


Figura 14.1.: Ejemplo de la primera codificación. En la parte izquierda aparece la posición de ajedrez y en la parte derecha la codificación con los tableros. La primera matriz se corresponde con el tablero de peones y la segunda con el tablero de torres. [68]

14.4.2. Modelo

Una vez hemos explicada la primera forma que vamos a utilizar para representar una posición de ajedrez pasamos a presentar el modelo que va a hacer uso de ella, se trata de una red neuronal convolucional. Los parámetros utilizados han sido el tamaño del kernel, el número de filtros y el porcentaje de *drop out*, en este caso al contar con tableros de tamaño 8×8 se considera oportuno no utilizar capas de agrupación pero sí el uso de una capa convolucional con tamaño de kernel 1×1 para reducir la dimensionalidad de la entrada. Es necesario un estudio previo para obtener estos parámetros, así como el número de capas y neuronas por capa. El valor de *drop out* escogido ha sido de 0.4, ya que es el recomendado por la literatura. En la siguiente tabla aparece la arquitectura del modelo para una etiqueta con tres clases como la ventaja de espacio:

14.5. Modelo Segunda Representación (Representación Unidimensional)

Layer	Output Shape	Param
Conv2D	(None,6,6,64)	4096
Conv2D	(None,6,6,64)	4160
Drop Out	(None,6,6,64)	0
Flatten	(None,2304)	0
Dense	(None,64)	147520
Batch Normalization	(None,64)	256
Dense	(None,32)	2080
Batch Normalization	(None,32)	128
Dense	(None,64)	2112
Batch Normalization	(None,64)	256
Dense	(None,16)	1040
Batch Normalization	(None,16)	64
Dense	(None,3)	51
Total Parámetros	161763	

Tabla 14.1.: Arquitectura del modelo (primera representación). Caso de etiquetas con tres clases donde las entradas son matrices tridimensionales de tamaño 8x8x7.

14.5. Modelo Segunda Representación (Representación Unidimensional)

14.5.1. Segunda Representación

La segunda representación que se ha utilizado es una modificación de la representación que se acaba de explicar. La diferencia es que ahora en vez de tener siete tableros como una matriz tridimensional aparecen uno a continuación del otro en un vector unidimensional, de manera que las primeras sesenta y cuatro componentes corresponden al tablero de reyes, las siguientes sesenta y cuatro al de damas y así hasta las últimas sesenta y cuatro que se corresponden con el tablero de turno. Esto supondrá un problema, ya que se pierde la localidad espacial. Esta representación no es recomendable pero ha sido propuesta a modo de experimento.

14.5.2. Modelo

Presentamos ahora el modelo que ha empleado como datos de entrada las posiciones del tablero representadas mediante un tablero unidimensional. Este modelo está formado por capas densas donde no aparece ninguna capa convolucional. Esto es así, ya que trabajamos con un vector unidimensional como dato de entrada. Por tanto, al no haber capas de convolución, solo se ha tenido que escoger el porcentaje de *drop out* que en este caso será de 0.4 y el número de capas y neuronas que hay en cada capa. La arquitectura de este modelo para una etiqueta con tres clases es:

Layer	Output Shape	Param
Dense	(None,128)	57472
Batch Normalization	(None,128)	512
Dense	(None,64)	8256
Batch Normalization	(None,64)	256
Dense	(None,128)	8320
Batch Normalization	(None,128)	512
Dense	(None,256)	33024
Batch Normalization	(None,256)	1024
Dense	(None,128)	32896
Batch Normalization	(None,128)	512
Drop Out	(None,128)	0
Dense	(None,64)	8256
Batch Normalization	(None,64)	256
Dense	(None,16)	1040
Batch Normalization	(None,16)	64
Dense	(None,3)	51
Total Parámetros	152451	

Tabla 14.2.: Arquitectura del modelo (segunda representación). Caso de etiquetas con tres clases donde las entradas son vectores unidimensionales de tamaño 448.

14.6. Modelo Tercera Representación (Representación Bidimensional)

14.6.1. Tercera Representación

La tercera representación tiene en cuenta que hay doce piezas distintas sobre el tablero distinguiendo el bando de cada una y también pueden haber casillas vacías. Una vez se tiene esto en consideración se le va a dar a cada pieza blanca un valor positivo, a cada pieza negra un valor negativo y las casillas vacías tendrá el valor neutro, esta es la representación conocida como lista cuadrada (ver sección 5.2.2.2). Por tanto, se va a representar cada pieza con la siguiente notación:

- *Casilla vacía* → 0
- *Rey blanco* → 1
- *Dama blanca* → 2
- *Torre blanca* → 3
- *Alfil blanco* → 4
- *Caballo blanco* → 5
- *Peón blanco* → 6
- *Rey negro* → -1
- *Dama negra* → -2

14.6. Modelo Tercera Representación (Representación Bidimensional)

- Torre negra → -3
- Alfil negro → -4
- Caballo negro → -5
- Peón negro → -6

Una vez se ha establecido la notación ya se puede representar el tablero, esta representación cuenta con un único canal de tamaño 8x8 y no se añadirá otro para saber al bando al que le toca mover, ya que para este problema no es necesario. Se presenta un ejemplo con una posición de ajedrez y la codificación explicada.

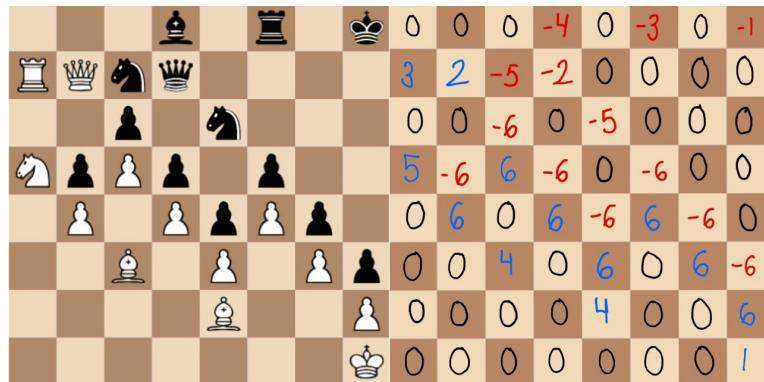


Figura 14.2.: A la izquierda aparece la posición real y a la derecha la posición codificada.

Esta tercera representación al igual que la primera y a diferencia de la segunda sí mantiene la localidad espacial.

14.6.2. Modelo

El tercer modelo, el que se ha utilizado para la tercera representación, está formado únicamente por redes neuronales convolucionales. Al tener solamente una dimensión de profundidad y como el tablero tiene dimensiones 8x8 solo aparecerá una única capa de convolución. Por tanto, se tiene que realizar un pequeño estudio para escoger el tamaño del kernel y el número de filtros más adecuado. También será necesario establecer el porcentaje de *drop out* que en este caso será de 0.2. En estos modelos no se emplea una capa de convolución 1x1 para reducir la profundidad de la entrada, puesto que esta tiene dimensión uno. Se va a presentar la última arquitectura para el caso de una etiqueta que tiene tres clases:

Layer	Output Shape	Param
Conv2D	(None,6,6,32)	320
Drop Out	(None,6,6,32)	0
Flatten	(None,1152)	0
Dense	(None,64)	73792
Batch Normalization	(None,64)	256
Dense	(None,128)	8320
Batch Normalization	(None,128)	512
Dense	(None,256)	33024
Batch Normalization	(None,256)	1024
Dense	(None,128)	32896
Batch Normalization	(None,128)	512
Dense	(None,64)	8256
Batch Normalization	(None,64)	256
Dense	(None,32)	2080
Batch Normalization	(None,32)	128
Dense	(None,64)	2112
Batch Normalization	(None,64)	256
Dense	(None,16)	1040
Batch Normalization	(None,16)	64
Dense	(None,3)	51
Total Parámetros	164899	

Tabla 14.3.: Arquitectura del modelo (tercera representación). Caso de etiquetas con tres clases donde las entradas son matrices de tamaño 8x8.

14.7. Proceso de Entrenamiento

En esta sección se va a explicar el proceso de entrenamiento que se ha llevado a cabo para que los modelos aprendan. Para realizar el entrenamiento se ha hecho uso de Colab, también conocido como Colaboratory, una herramienta de Google que permite programar y ejecutar Python en el navegador con acceso a GPUs sin coste durante cierto tiempo. A continuación, se van a explicar los hiperparámetros utilizados en el entrenamiento.

El optimizador empleado para realizar el entrenamiento de todos los modelos es el optimizador *Adam* con una tasa de aprendizaje inicial de 10^{-3} , $\beta_1 = 0.9$ y $\beta_2 = 0.999$ y el tamaño de batch ha sido de 32 debido a que suele ser un múltiplo de 2 y 32 es uno de los tamaños estándar. La función de pérdida utilizada para cada modelo es la entropía cruzada categórica, que en general es una medida de la distancia entre funciones de probabilidad. Esta suele ser adecuada en modelos de redes donde se realiza una clasificación categórica con función de activación softmax como sucede en este caso. Las métricas utilizadas ya han sido explicadas en la sección 14.2 y son: la métrica Accuracy y la métrica F1-score.

El número de épocas para entrenar cada conjunto de modelos ha variado, para los modelos que conforman el primer conjunto (modelos tridimensionales) se han empleado 80 épocas, para los modelos de la segunda representación 150 épocas y para el tercer conjunto de modelos 100 épocas. También se ha hecho uso de *early stopping* 10.5.4 para evitar el problema

de overfitting al imponer el número máximo de épocas con las que puede estar entrenando el modelo sin que haya mejora, tras las cuales se para el entrenamiento. Este parámetro recibe el nombre de paciencia y su valor ha variado según el modelo entrenado, pero en todos los casos se ha seguido el mismo criterio, aproximadamente el 20 % del número de épocas totales impuestas de antemano.

Una vez explicados los hiperparámetros se explica el proceso que se ha seguido para entrenar los modelos. En primer lugar, se han importado los archivos CSV donde se encuentran las etiquetas y los datos de entrada. Las etiquetas han sido convertidas a datos categóricos, puesto que resultaba más sencillo trabajar con este tipo de datos con la función de pérdida que se ha escogido. Posteriormente, se ha aplicado el método *hold-out* para seleccionar los modelos dividiendo el conjunto de datos en tres grupos, conjuntos de entrenamiento, validación y test, y con estos conjuntos y con los hiperparámetros descritos son con los que se han seleccionado, validado y evaluado los modelos.

15. Resultados Experimentales y Análisis

En esta parte del trabajo se presentan y analizan los resultados obtenidos habiendo utilizado los modelos descritos en el capítulo previo. En primer lugar, se muestran una serie de curvas de aprendizaje y algunas matrices de confusión para un par de etiquetas. Posteriormente, se presentan los resultados y una serie de gráficas para poder compararlos y facilitar así, el análisis posterior. Tras esto, se exponen una serie de tablas donde se muestra la utilidad de nuestro estudio. En la última parte de este capítulo se analizan detenidamente los resultados obtenidos.

15.1. Resultados

En esta sección se presentan los resultados obtenidos gracias a los modelos comentados anteriormente. Las gráficas y tablas donde se recogen los resultados serán analizadas en la segunda parte del capítulo. En primer lugar, se muestran una serie de gráficas donde aparecen las curvas de entrenamiento (curvas de aprendizaje y curvas de pérdida) de los modelos para dos etiquetas. Estas gráficas se corresponden con el entrenamiento de las etiquetas ventaja de espacio y número de columnas semiabiertas de las blancas. En color azul se representa la curva de entrenamiento y en naranja la de validación, en el eje X tendremos las épocas y en el eje Y la métrica, en este caso, la métrica Accuracy.

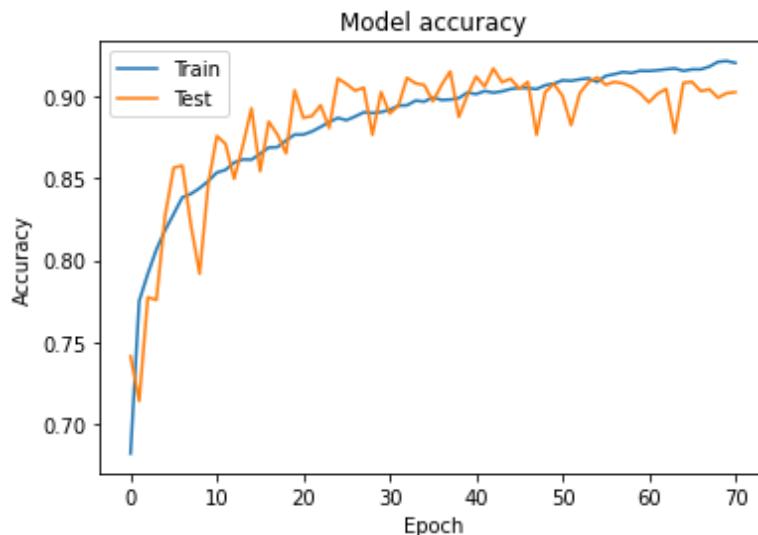


Figura 15.1.: Etiqueta: *Ventaja de Espacio*. Curva de aprendizaje para el modelo de la Primera Representación (3D).

15. Resultados Experimentales y Análisis

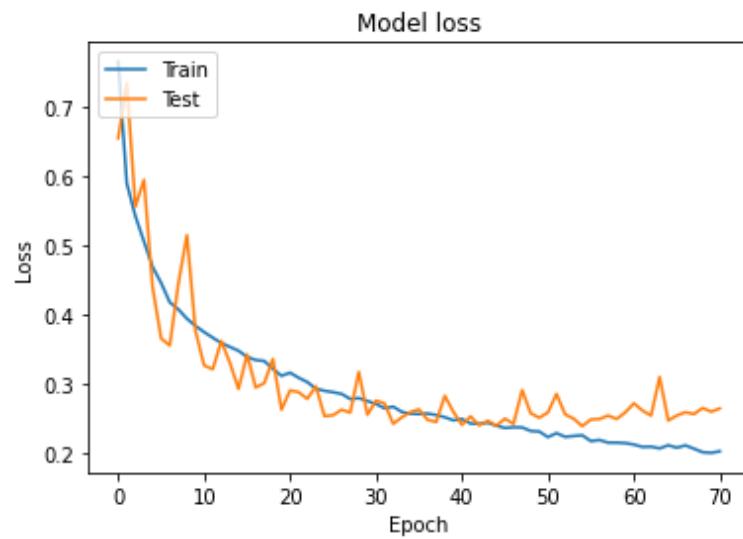


Figura 15.2.: Etiqueta: *Ventaja de Espacio*. Curva de pérdida para el modelo de la Primera Representación (3D).

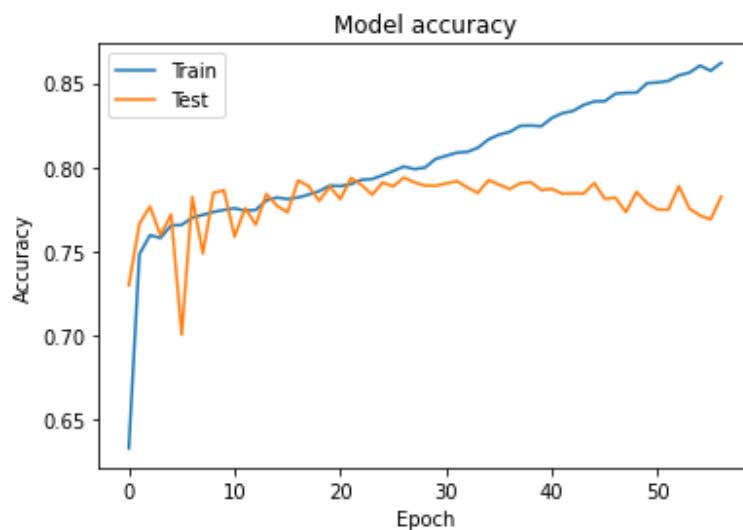


Figura 15.3.: Etiqueta: *Ventaja de Espacio*. Curva de aprendizaje para el modelo de la Segunda Representación (1D).

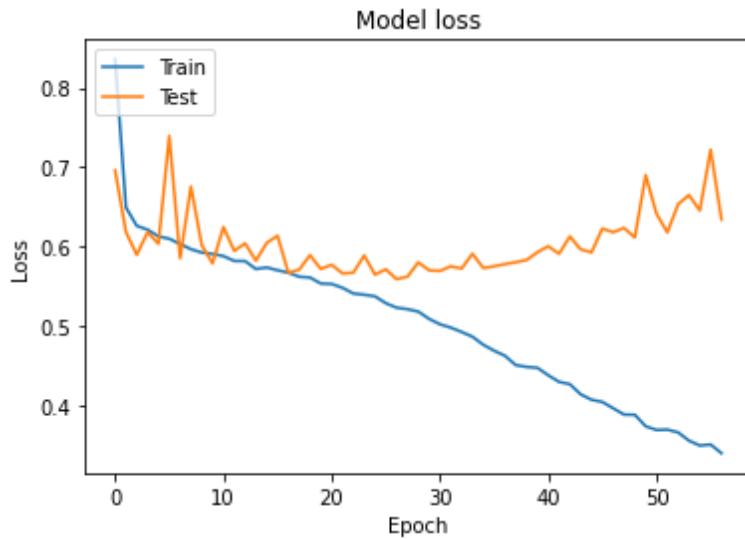


Figura 15.4.: Etiqueta: *Ventaja de Espacio*. Curva de pérdida para el modelo de la Segunda Representación (1D).

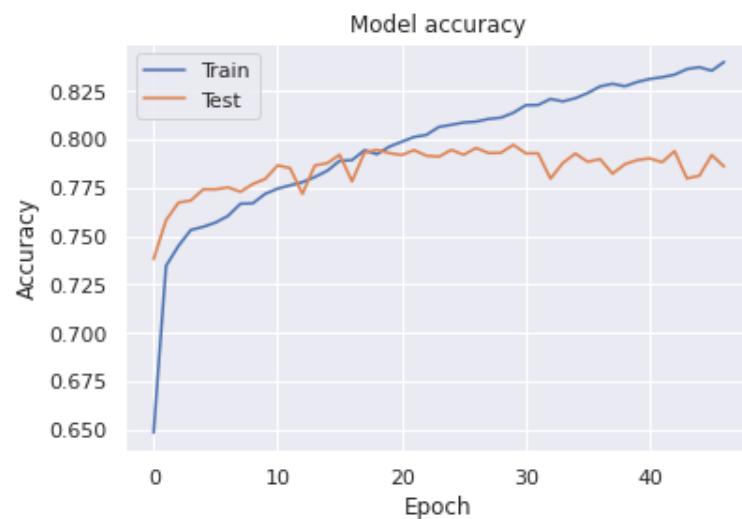


Figura 15.5.: Etiqueta: *Ventaja de Espacio*. Curva de aprendizaje para el modelo de la Tercera Representación (2D).

15. Resultados Experimentales y Análisis

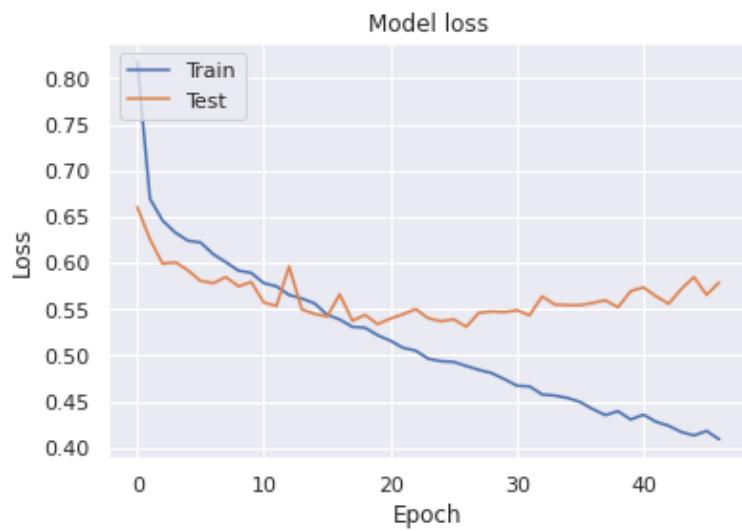


Figura 15.6.: Etiqueta: *Ventaja de Espacio*. Curva de pérdida para el modelo de la Tercera Representación (2D).

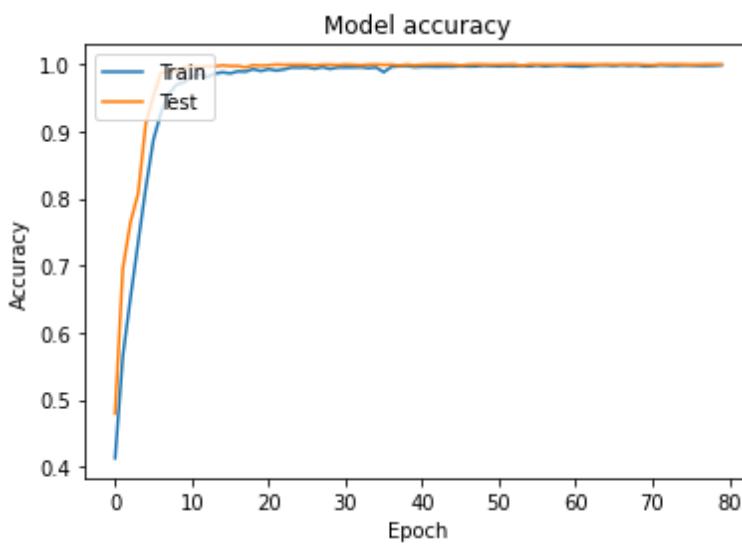


Figura 15.7.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de aprendizaje para el modelo de la Primera Representación (3D).

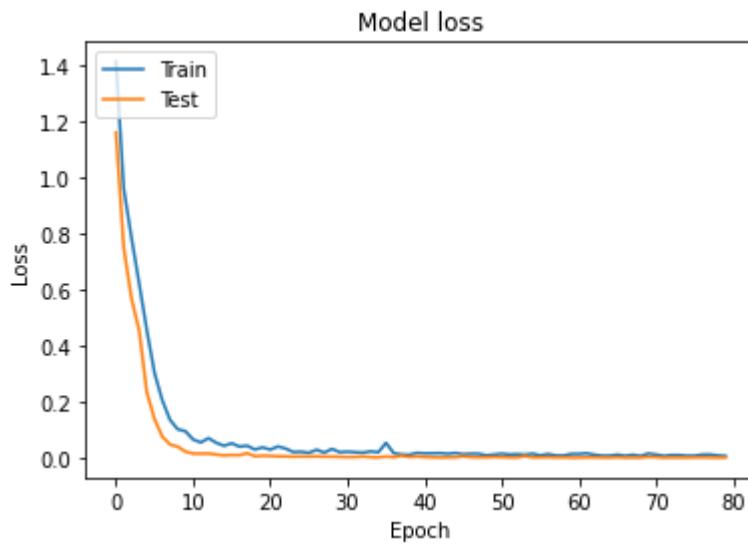


Figura 15.8.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de pérdida para el modelo de la Primera Representación (3D).

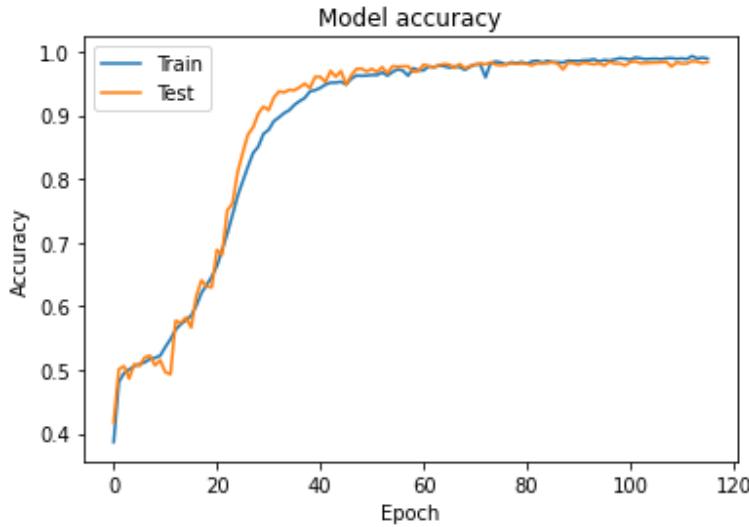


Figura 15.9.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de aprendizaje para el modelo de la Segunda Representación (1D).

15. Resultados Experimentales y Análisis

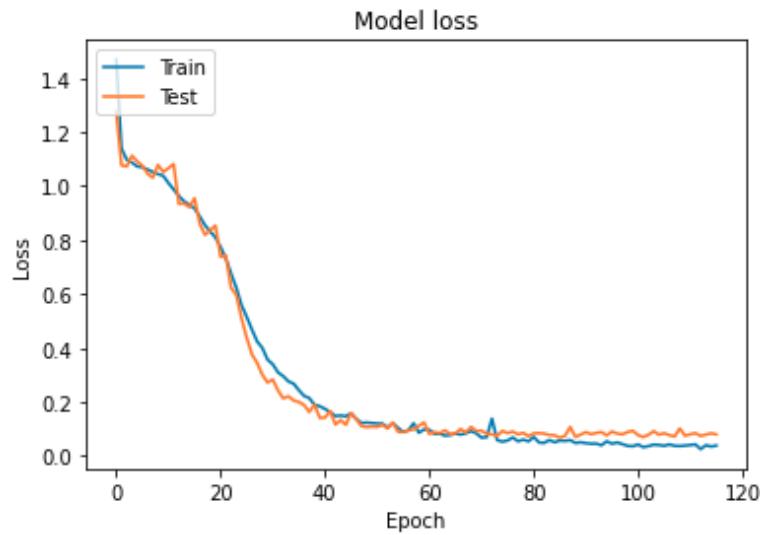


Figura 15.10.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de pérdida para el modelo de la Segunda Representación (1D).

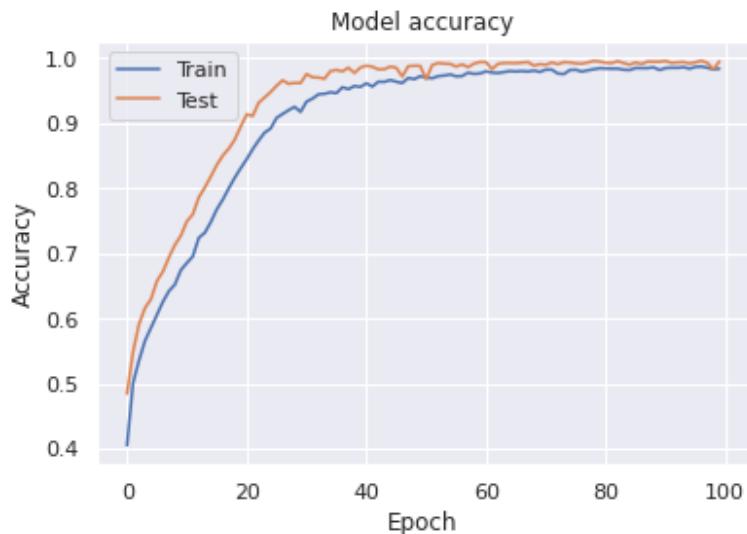


Figura 15.11.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de aprendizaje para el modelo de la Tercera Representación (2D).

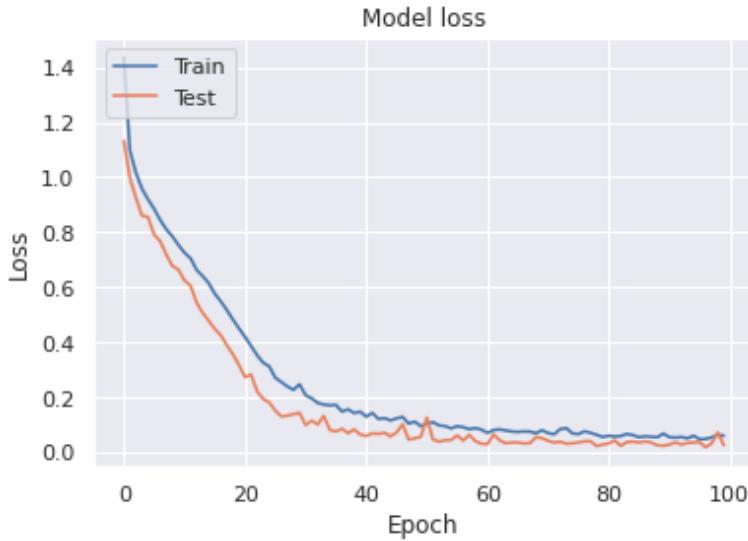


Figura 15.12.: Etiqueta: *Número de Columnas Semiabiertas para las Blancas*. Curva de pérdida para el modelo de la Tercera Representación (2D).

Las gráficas del resto de etiquetas no aparecerán en este documento pero se representarán de la misma manera que estas y gracias a las tablas donde aparecen los resultados alcanzados se puede tener una idea de cada una de ellas. A continuación, se presentan algunas de las matrices de confusión que se han obtenido al evaluar los modelos, se muestran seis matrices, tres de ellas para la etiqueta ventaja de espacio y las otras tres para la etiqueta peones blancos.

Ventaja Espacio	Igualdad	Blancas	Negras
Igualdad	483	234	257
Blancas	135	4604	88
Negras	106	82	4011

Tabla 15.1.: Etiqueta: *Ventaja de Espacio*. Matriz de Confusión obtenida con el modelo de la Primera Representación (3D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

15. Resultados Experimentales y Análisis

Ventaja Espacio	Igualdad	Blancas	Negras
Igualdad	0	450	524
Blancas	0	4196	631
Negras	0	433	3766

Tabla 15.2.: Etiqueta: *Ventaja de Espacio*. Matriz de Confusión obtenida con el modelo de la Segunda Representación (1D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

Ventaja Espacio	Igualdad	Blancas	Negras
Igualdad	36	489	449
Blancas	28	4300	499
Negras	27	499	3673

Tabla 15.3.: Etiqueta: *Ventaja de Espacio*. Matriz de Confusión obtenida con el modelo de la Tercera Representación (2D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

Peones Blancos	Cero	Uno	Dos	Tres	Cuatro	Cinco	Seis	Siete	Ocho
Cero	18	0	0	0	0	0	0	0	0
Uno	0	116	0	0	0	0	0	0	0
Dos	0	0	384	0	0	0	0	0	0
Tres	0	0	0	992	0	0	0	0	0
Cuatro	0	0	0	0	1648	0	0	0	0
Cinco	0	0	0	0	0	2490	0	0	0
Seis	0	0	0	0	0	0	2711	0	0
Siete	0	0	0	0	0	0	0	1476	0
Ocho	0	0	0	0	0	0	0	0	165

Tabla 15.4.: Etiqueta: *Peones Blancos*. Matriz de Confusión obtenida con el modelo de la Primera Representación (3D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

Peones Blancos	Cero	Uno	Dos	Tres	Cuatro	Cinco	Seis	Siete	Ocho
Cero	2	16	0	0	0	0	0	0	0
Uno	0	94	22	0	0	0	0	0	0
Dos	0	7	345	32	0	0	0	0	0
Tres	0	0	24	889	79	0	0	0	0
Cuatro	0	0	0	39	1543	66	0	0	0
Cinco	0	0	0	0	80	2382	28	0	0
Seis	0	0	0	0	0	112	2568	31	0
Siete	0	0	0	0	0	1	82	1371	22
Ocho	0	0	0	0	0	0	0	11	154

Tabla 15.5.: Etiqueta: *Peones Blancos*. Matriz de Confusión obtenida con el modelo de la Segunda Representación (1D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

Peones Blancos	Cero	Uno	Dos	Tres	Cuatro	Cinco	Seis	Siete	Ocho
Cero	18	0	0	0	0	0	0	0	0
Uno	0	116	0	0	0	0	0	0	0
Dos	0	0	384	0	0	0	0	0	0
Tres	0	0	0	992	0	0	0	0	0
Cuatro	0	0	0	0	1648	0	0	0	0
Cinco	0	0	0	0	6	2484	0	0	0
Seis	0	0	0	0	0	8	2703	0	0
Siete	0	0	0	0	0	0	9	1467	0
Ocho	0	0	0	0	0	0	0	0	165

Tabla 15.6.: Etiqueta: *Peones Blancos*. Matriz de Confusión obtenida con el modelo de la Tercera Representación (2D). Las filas representan los valores reales observados y las columnas los valores que han sido predichos.

Antes de presentar los resultados se muestran unas gráficas donde se podrán apreciar y comparar de manera visual algunos de los resultados que se presentarán en las tablas. En estas gráficas aparecerán los resultados obtenidos habiendo empleado los diferentes modelos con las distintas métricas. Las gráficas se corresponden con los resultados de cuatro etiquetas: ventaja de espacio, peones blancos, damas blancas y peones pasados blancos.

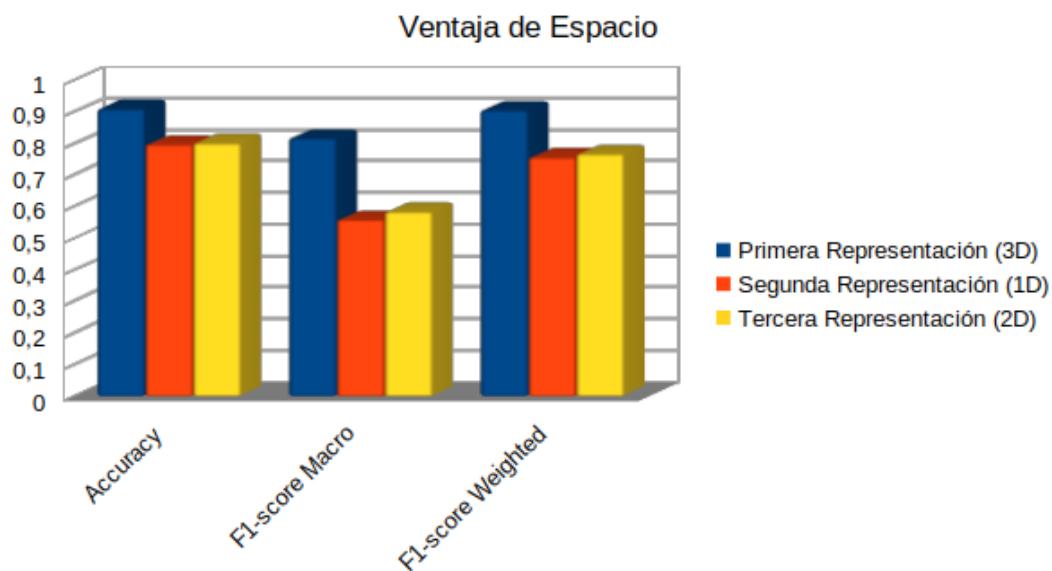


Figura 15.13.: Etiqueta: *Ventaja de Espacio*. Gráfica donde se representan de manera visual los resultados obtenidos.

15. Resultados Experimentales y Análisis

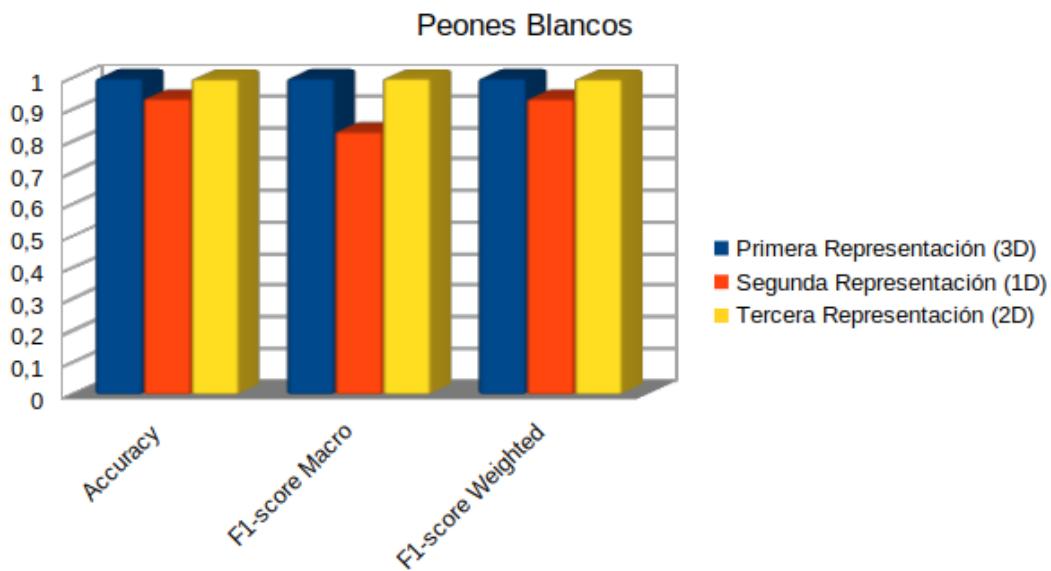


Figura 15.14.: Etiqueta: *Peones Blancos*. Gráfica donde se representan de manera visual los resultados obtenidos.

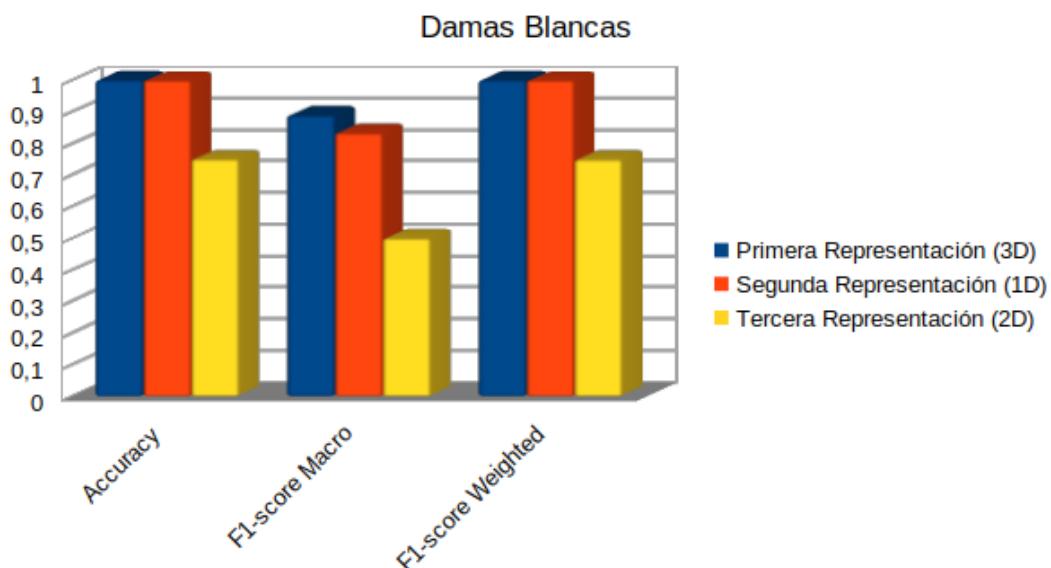


Figura 15.15.: Etiqueta: *Damas Blancas*. Gráfica donde se representan de manera visual los resultados obtenidos.

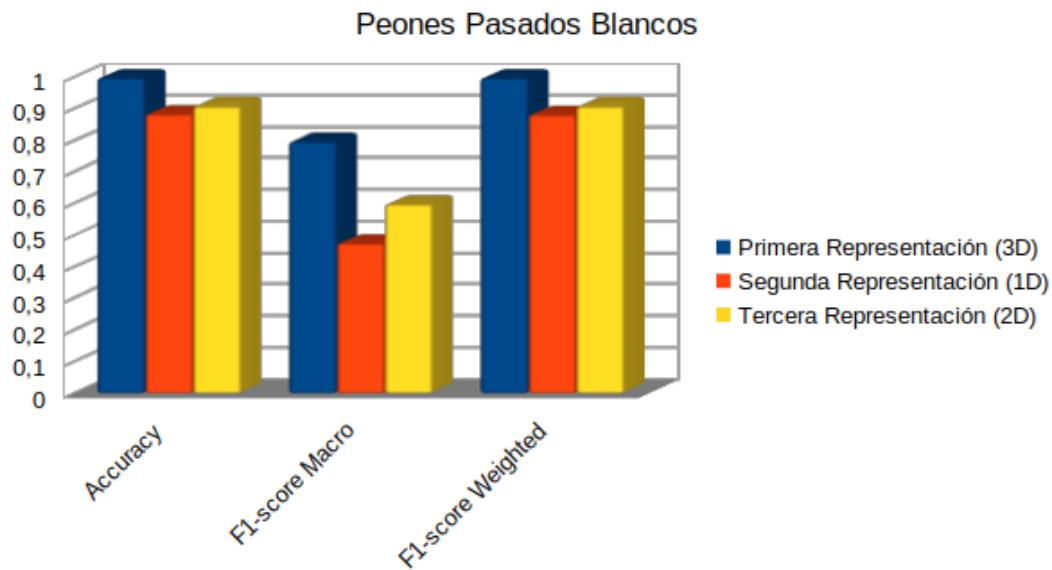


Figura 15.16.: Etiqueta: *Peones Pasados Blancos*. Gráfica donde se representan de manera visual los resultados obtenidos.

Llega el momento de presentar los resultados que se han obtenido al evaluar los modelos viendo cómo de bien se comportan frente a datos nuevos de prueba. Los resultados se han dividido en cuatro tablas, en la primera aparecerán los resultados obtenidos empleando el primer modelo, es decir, el que utiliza la primera representación. En la segunda aparecen los resultados obtenidos al emplear el modelo que hace uso de la segunda representación y en la tercera aparecerán los resultados pertenecientes al tercer modelo, el de la tercera representación. En estas tres tablas aparecen las 38 etiquetas en la primera columna, en la segunda columna se muestran los resultados de la función de pérdida con la métrica Accuracy, y en el resto de columnas se presentan los resultados alcanzados según las diferentes métricas que se han utilizado, tanto Accuracy como la metrica F1-score con ambos promedios. Los resultados se muestran redondeados a cuatro cifras significativas. La cuarta tabla está formada por los resultados obtenidos por los diferentes modelos empleando la métrica F1-score con promedio macro. Esta métrica es la más importante y la que se ha utilizado para analizar los modelos y gracias a esta tabla se pueden comparar los resultados de manera más sencilla.

Modelo Primera Representación (3D)				
Etiquetas	Loss (Acc)	Accuracy	F1-score Macro	F1-score Weighted
Ventaja Espacio	0.2408	0.9098	0.8171	0.9052
Columnas Abiertas	0.001	0.9997	0.9872	0.9997
Alfiles Mismo Color	0.0009	0.9999	0.9998	0.9999
Alfiles Distinto Color	0.0003	0.9999	0.9996	0.9999
Columnas Semiabiertas B	0.0019	0.9997	0.9918	0.9997
Peones B	0.0019	1.0	1.0	1.0
Caballos B	0.0008	0.9998	0.9996	0.9998
Alfiles B	0.0005	0.9999	0.9999	0.9999
Torres B	0.0001	1.0	1.0	1.0
Damas B	0.0014	0.9999	0.8889	0.9999
Torre en Séptima B	0.0	1.0	1.0	1.0
Torres Dobladas B	0.0157	0.9952	0.9785	0.9952
Torres Ligadas B	0.0094	0.9976	0.9963	0.9976
Pistola de Alekhine B	0.0022	0.9997	0.4999	0.9996
Peones Doblados B	0.0282	0.9935	0.6762	0.9931
Peones Aislados B	0.0077	0.9982	0.9522	0.9982
Peones Retrasados B	0.0101	0.998	0.9951	0.998
Peones Pasados B	0.0108	0.9966	0.7956	0.9966
Islas de Peones B	0.0001	1.0	1.0	1.0
Falanges de Peones B	0.0007	0.9998	0.9953	0.9998
Peones Conectados B	0.0019	0.9996	0.9995	0.9996
Columnas Semiabiertas N	0.0019	0.9996	0.8496	0.9996
Peones N	0.0015	0.9998	0.9989	0.9998
Caballos N	0.0001	1.0	1.0	1.0
Alfiles N	0.0	1.0	1.0	1.0
Torres N	0.0	1.0	1.0	1.0
Damas N	0.0021	0.9997	0.6666	0.9997
Torre en Séptima N	0.0	1.0	1.0	1.0
Torres Dobladas N	0.0173	0.9967	0.9831	0.9967
Torres Ligadas N	0.0101	0.9978	0.9966	0.9978
Pistola de Alekhine N	0.003	0.9997	0.4999	0.9996
Peones Doblados N	0.035	0.9924	0.6755	0.9921
Peones Aislados N	0.0068	0.9983	0.841	0.9982
Peones Retrasados N	0.0168	0.9957	0.9917	0.9957
Peones Pasados N	0.0201	0.9943	0.7854	0.9943
Islas de Peones N	0.0001	1.0	1.0	1.0
Falanges de Peones N	0.0023	0.9996	0.991	0.9996
Peones Conectados N	0.0036	0.9989	0.9977	0.9989

Tabla 15.7.: Resultados obtenidos con el modelo de la Primera Representación donde B representa piezas blancas y N piezas negras.

Modelo Segunda Representación (1D)				
Etiquetas	Loss (Acc)	Accuracy	F1-score Macro	F1-score Weighted
Ventaja Espacio	0.5435	0.7962	0.5577	0.7557
Columnas Abiertas	0.02	0.995	0.9807	0.995
Alfiles Mismo Color	0.0393	0.9937	0.9895	0.9937
Alfiles Distinto Color	0.0001	1.0	1.0	1.0
Columnas Semiabiertas B	0.0742	0.983	0.9737	0.983
Peones B	0.204	0.9348	0.8309	0.9343
Caballos B	0.0567	0.9896	0.9837	0.9895
Alfiles B	0.0464	0.9887	0.9853	0.9887
Torres B	0.0447	0.9881	0.9888	0.9881
Damas B	0.0038	0.999	0.8327	0.999
Torre en Séptima B	0.0081	0.9984	0.9944	0.9984
Torres Dobladas B	0.1189	0.9718	0.8644	0.9708
Torres Ligadas B	0.0645	0.9822	0.9723	0.9821
Pistola de Alekhine B	0.0032	0.9997	0.4999	0.9996
Peones Doblados B	0.1933	0.9451	0.5853	0.9422
Peones Aislados B	0.0847	0.9802	0.7835	0.9799
Peones Retrasados B	0.4114	0.8541	0.7918	0.8537
Peones Pasados B	0.2973	0.883	0.4737	0.8807
Islas de Peones B	0.0412	0.9911	0.9894	0.9911
Falanges de Peones B	0.1475	0.9736	0.9402	0.9734
Peones Conectados B	0.1905	0.944	0.9257	0.9438
Columnas Semiabiertas N	0.0703	0.9838	0.8076	0.9838
Peones N	0.2406	0.9161	0.8351	0.9156
Caballos N	0.049	0.9876	0.9791	0.9876
Alfiles N	0.0396	0.9923	0.9907	0.9923
Torres N	0.048	0.9848	0.9859	0.9848
Damas N	0.008	0.9977	0.8874	0.9977
Torre en Séptima N	0.0081	0.9984	0.9941	0.9984
Torres Dobladas N	0.1196	0.9718	0.8429	0.9705
Torres Ligadas N	0.0527	0.987	0.9795	0.987
Pistola de Alekhine N	0.004	0.9997	0.4999	0.9996
Peones Doblados N	0.1926	0.9413	0.5207	0.9372
Peones Aislados N	0.0583	0.9874	0.9128	0.9873
Peones Retrasados N	0.4213	0.8555	0.8184	0.8551
Peones Pasados N	0.3008	0.8855	0.4958	0.8854
Islas de Peones N	0.0401	0.992	0.9922	0.992
Falanges de Peones N	0.0963	0.9769	0.9441	0.9768
Peones Conectados N	0.2236	0.9349	0.9081	0.9346

Tabla 15.8.: Resultados obtenidos con el modelo de la Segunda Representación donde B representa piezas blancas y N piezas negras.

Modelo Tercera Representación (2D)				
Etiquetas	Loss (Acc)	Accuracy	F1-score Macro	F1-score Weighted
Ventaja Espacio	0.5208	0.8009	0.5836	0.7667
Columnas Abiertas	0.0909	0.9815	0.9603	0.9814
Alfiles Mismo Color	0.4617	0.8169	0.4496	0.7346
Alfiles Distinto Color	0.2522	0.9255	0.4912	0.8917
Columnas Semiabiertas B	0.0272	0.9944	0.9954	0.9944
Peones B	0.0088	0.9977	0.9988	0.9977
Caballos B	0.7631	0.6322	0.5399	0.6197
Alfiles B	0.8382	0.5918	0.5263	0.5761
Torres B	0.6575	0.6773	0.5969	0.6676
Damas B	0.5138	0.7491	0.4992	0.7487
Torre en Séptima B	0.0119	0.9961	0.9865	0.9961
Torres Dobladas B	0.197	0.939	0.5276	0.9164
Torres Ligadas B	0.0831	0.975	0.9605	0.9747
Pistola de Alekhine B	0.0027	0.9997	0.4999	0.9996
Peones Doblados B	0.0112	0.9965	0.7194	0.9965
Peones Aislados B	0.0112	0.997	0.8938	0.997
Peones Retrasados B	0.0059	0.9988	0.9957	0.9988
Peones Pasados B	0.2846	0.9077	0.5978	0.9079
Islas de Peones B	0.001	1.0	1.0	1.0
Falanges de Peones B	0.002	0.9996	0.9995	0.9996
Peones Conectados B	0.0238	0.9926	0.9906	0.9926
Columnas Semiabiertas N	0.0141	0.9977	0.9981	0.9977
Peones N	0.0139	0.9968	0.9919	0.9968
Caballos N	0.7587	0.6305	0.5647	0.6251
Alfiles N	0.8374	0.5885	0.559	0.5845
Torres N	0.6108	0.7017	0.6619	0.7031
Damas N	0.515	0.7456	0.497	0.7453
Torre en Séptima N	0.0172	0.9947	0.9804	0.9947
Torres Dobladas N	0.175	0.9484	0.4868	0.9233
Torres Ligadas N	0.0764	0.9778	0.9648	0.9777
Pistola de Alekhine N	0.0029	0.9997	0.4999	0.9996
Peones Doblados N	0.0083	0.9973	0.7235	0.9973
Peones Aislados N	0.0044	0.9993	0.947	0.9993
Peones Retrasados N	0.0194	0.9951	0.9913	0.9951
Peones Pasados N	0.282	0.9103	0.5343	0.909
Islas de Peones N	0.0003	1.0	1.0	1.0
Falanges de Peones N	0.0051	0.9993	0.9868	0.9993
Peones Conectados N	0.021	0.9938	0.9897	0.9938

Tabla 15.9.: Resultados obtenidos con el modelo de la Tercera Representación donde B representa piezas blancas y N piezas negras.

Métrica F1-score Macro Average			
Etiquetas	Primera Repr. (3D)	Segunda Repr. (1D)	Tercera Repr. (2D)
Ventaja Espacio	0.8171	0.5577	0.5836
Columnas Abiertas	0.9872	0.9807	0.9603
Alfiles Mismo Color	0.9998	0.9895	0.4496
Alfiles Distinto Color	0.9996	1.0	0.4912
Columnas Semiabiertas B	0.9918	0.9737	0.9954
Peones B	1.0	0.8309	0.9988
Caballos B	0.9996	0.9837	0.5399
Alfiles B	0.9999	0.9853	0.5263
Torres B	1.0	0.9888	0.5969
Damas B	0.8889	0.8327	0.4992
Torre en Séptima B	1.0	0.9944	0.9865
Torres Dobladas B	0.9785	0.8644	0.5276
Torres Ligadas B	0.9963	0.9723	0.9605
Pistola de Alekhine B	0.4999	0.4999	0.4999
Peones Doblados B	0.6762	0.5853	0.7194
Peones Aislados B	0.9522	0.7835	0.8938
Peones Retrasados B	0.9951	0.7918	0.9957
Peones Pasados B	0.7956	0.4737	0.5978
Islas de Peones B	1.0	0.9894	1.0
Falanges de Peones B	0.9953	0.9402	0.9995
Peones Conectados B	0.9995	0.9257	0.9906
Columnas Semiabiertas N	0.8496	0.8076	0.9981
Peones N	0.9989	0.8351	0.9919
Caballos N	1.0	0.9791	0.5647
Alfiles N	1.0	0.9907	0.559
Torres N	1.0	0.9859	0.6619
Damas N	0.6666	0.8874	0.497
Torre en Séptima N	1.0	0.9941	0.9804
Torres Dobladas N	0.9831	0.8429	0.4868
Torres Ligadas N	0.9966	0.9795	0.9648
Pistola de Alekhine N	0.4999	0.4999	0.4999
Peones Doblados N	0.6755	0.5207	0.7235
Peones Aislados N	0.841	0.9128	0.947
Peones Retrasados N	0.9917	0.8184	0.9913
Peones Pasados N	0.7854	0.4958	0.5343
Islas de Peones N	1.0	0.9922	1.0
Falanges de Peones N	0.991	0.9441	0.9868
Peones Conectados N	0.9977	0.9081	0.9897

Tabla 15.10.: Tabla que muestra una comparativa de los resultados obtenidos con los diferentes modelos empleando la métrica F1-score con el promedio *Macro Average*, B representa piezas blancas y N piezas negras. Los mejores resultados han sido destacados.

15. Resultados Experimentales y Análisis

Para terminar esta sección se presentan una serie de tablas donde se muestra, con un ejemplo, la utilidad de este proyecto. En cada una de las tablas aparecerá la etiqueta o etiquetas con sus correspondientes clases y para cada una de estas clases la frecuencia con la que aparece en las diferentes posiciones examinadas. La información de estas tablas se ha extraído de 3000 posiciones de diferentes partidas no vistas anteriormente, donde en todas ellas, el jugador que se ha analizado juega con blancas y realizó una mala jugada. En total se presentan 21 tablas, las cuatro primeras son las correspondientes a las etiquetas comunes, están formadas por dos columnas, en la primera columna aparecen las clases de la etiqueta y en la segunda la frecuencia con la que aparece cada clase. Las otras diecisiete tablas cuentan con tres columnas, en la primera aparecen las clases, en la segunda, la frecuencia con la que aparece cada una de las clases para la etiqueta de las piezas blancas y en la tercera, la frecuencia de las clases para la etiqueta de las piezas negras, la frecuencia con la que aparece cada clase. La información de estas tablas se ha extraído empleando para cada etiqueta el modelo que mejores resultados obtuvo según la tabla 15.10.

Ventaja de Espacio	
Igualdad	9 % de las posiciones
Ventaja Propia	46 % de las posiciones
Ventaja Rival	45 % de las posiciones

Tabla 15.11.: Tabla con información relevante sobre la etiqueta *Ventaja de Espacio* para un conjunto de 3000 partidas jugadas con blancas.

Columnas Abiertas	
Ninguna	15 % de las posiciones
Una	32 % de las posiciones
Dos	27 % de las posiciones
Tres	17 % de las posiciones
Cuatro	6.5 % de las posiciones
Cinco	2 % de las posiciones
Seis	0.5 % de las posiciones
Siete	0 % de las posiciones
Ocho	0 % de las posiciones

Tabla 15.12.: Tabla con información relevante sobre la etiqueta *Columnas Abiertas* para un conjunto de 3000 partidas jugadas con blancas.

Alfiles Mismo Color	
No	82.5 % de las posiciones
Sí	17.5 % de las posiciones

Tabla 15.13.: Tabla con información relevante sobre la etiqueta *Alfiles Mismo Color* para un conjunto de 3000 partidas jugadas con blancas.

Alfiles Distinto Color	
No	93 % de las posiciones
Sí	7 % de las posiciones

Tabla 15.14.: Tabla con información relevante sobre la etiqueta *Alfiles Distinto Color* para un conjunto de 3000 partidas jugadas con blancas.

Columnas Semiabiertas		
	Blancas/Propia	Negras/Rival
Ninguna	19 % de las posiciones	20.5 % de las posiciones
Una	42 % de las posiciones	44 % de las posiciones
Dos	28.5 % de las posiciones	27.5 % de las posiciones
Tres	9 % de las posiciones	7 % de las posiciones
Cuatro	1.5 % de las posiciones	1 % de las posiciones
Cinco	0 % de las posiciones	0 % de las posiciones
Seis	0 % de las posiciones	0 % de las posiciones

Tabla 15.15.: Tabla con información relevante sobre las etiquetas *Columnas Semiabiertas Blancas* y *Columnas Semiabiertas Negras* para un conjunto de 3000 partidas jugadas con blancas.

Número de Peones		
	Blancas/Prop	Negras/Riv
Ninguno	0 % de las posiciones	0 % de las posiciones
Uno	1 % de las posiciones	1 % de las posiciones
Dos	4 % de las posiciones	3 % de las posiciones
Tres	10 % de las posiciones	8 % de las posiciones
Cuatro	17 % de las posiciones	16 % de las posiciones
Cinco	25 % de las posiciones	26.5 % de las posiciones
Seis	26.5 % de las posiciones	28 % de las posiciones
Siete	15 % de las posiciones	15.5 % de las posiciones
Ocho	1.5 % de las posiciones	2 % de las posiciones

Tabla 15.16.: Tabla con información relevante sobre las etiquetas *Peones Blancos* y *Peones Negros* para un conjunto de 3000 partidas jugadas con blancas.

Número de Caballos		
	Blancas/Prop	Negras/Riv
Ninguno	47.5 % de las posiciones	45.2 % de las posiciones
Uno	42 % de las posiciones	43.5 % de las posiciones
Pareja de Caballos	10.5 % de las posiciones	11.3 % de las posiciones

Tabla 15.17.: Tabla con información relevante sobre las etiquetas *Caballos Blancos* y *Caballos Negros* para un conjunto de 3000 partidas jugadas con blancas.

15. Resultados Experimentales y Análisis

Número de Alfiles		
	Blancas/Prop	Negras/Riv
Ninguno	38 % de las posiciones	37 % de las posiciones
Uno	45.8 % de las posiciones	46.3 % de las posiciones
Pareja de Alfiles	16.2 % de las posiciones	16.7 % de las posiciones

Tabla 15.18.: Tabla con información relevante sobre las etiquetas *Alfiles Blancos* y *Alfiles Negros* para un conjunto de 3000 partidas jugadas con blancas.

Número de Torres		
	Blancas/Prop	Negras/Riv
Ninguna	15.5 % de las posiciones	14.3 % de las posiciones
Una	36.7 % de las posiciones	36.7 % de las posiciones
Dos	47.8 % de las posiciones	49 % de las posiciones

Tabla 15.19.: Tabla con información relevante sobre las etiquetas *Torres Blancas* y *Torres Negras* para un conjunto de 3000 partidas jugadas con blancas.

Número de Damas		
	Blancas/Prop	Negras/Riv
Ninguna	50.5 % de las posiciones	49.5 % de las posiciones
Una	49.5 % de las posiciones	50.5 % de las posiciones
Dos	0 % de las posiciones	0 % de las posiciones

Tabla 15.20.: Tabla con información relevante sobre las etiquetas *Damas Blancas* y *Damas Negras* para un conjunto de 3000 partidas jugadas con blancas.

Torre en Séptima		
	Blancas/Prop	Negras/Riv
Ninguna	93.7 % de las posiciones	91.5 % de las posiciones
Alguna	6.3 % de las posiciones	8.5 % de las posiciones

Tabla 15.21.: Tabla con información relevante sobre las etiquetas *Número de Torres Blancas en Séptima* y *Número de Torres Negras en Séptima* para un conjunto de 3000 partidas jugadas con blancas.

Torres Dobladadas		
	Blancas/Prop	Negras/Riv
No Dobladadas	94.3 % de las posiciones	94.3 % de las posiciones
Dobladas	5.7 % de las posiciones	5.7 % de las posiciones

Tabla 15.22.: Tabla con información relevante sobre las etiquetas *Torres Blancas Dobladadas* y *Torres Negras Dobladadas* para un conjunto de 3000 partidas jugadas con blancas.

Torres Ligadas		
	Blancas/Prop	Negras/Riv
No Ligadas	79 % de las posiciones	80 % de las posiciones
Ligadas	21 % de las posiciones	20 % de las posiciones

Tabla 15.23.: Tabla con información relevante sobre las etiquetas *Torres Blancas Ligadas* y *Torres Negras Ligadas* para un conjunto de 3000 partidas jugadas con blancas.

Pistola de Alekhine		
textbf{Blancas}/Prop	Negras/Riv	
No Aparece	100 % de las posiciones	100 % de las posiciones
Aparece	0 % de las posiciones	0 % de las posiciones

Tabla 15.24.: Tabla con información relevante sobre las etiquetas *Pistola de Alekhine Blanca* y *Pistola de Alekhine Negra* para un conjunto de 3000 partidas jugadas con blancas.

Número de Peones Doblados		
	Blancas/Prop	Negras/Riv
Ninguno	84.9 % de las posiciones	83 % de las posiciones
Uno	14.7 % de las posiciones	16.5 % de las posiciones
Dos	0.4 % de las posiciones	0.5 % de las posiciones
Tres	0 % de las posiciones	0 % de las posiciones

Tabla 15.25.: Tabla con información relevante sobre las etiquetas *Número de Peones Blancos Doblados* y *Número de Peones Negros Doblados* para un conjunto de 3000 partidas jugadas con blancas.

Número de Peones Aislados		
	Blancas/Prop	Negras/Riv
Ninguno	46.2 % de las posiciones	44.2 % de las posiciones
Uno	31.3 % de las posiciones	33.3 % de las posiciones
Dos	15.2 % de las posiciones	15.1 % de las posiciones
Tres	5.6 % de las posiciones	5.8 % de las posiciones
Cuatro	1.5 % de las posiciones	1.3 % de las posiciones
Cinco	0.2 % de las posiciones	0.3 % de las posiciones
Seis	0 % de las posiciones	0 % de las posiciones

Tabla 15.26.: Tabla con información relevante sobre las etiquetas *Número de Peones Blancos Aislados* y *Número de Peones Negros Aislados* para un conjunto de 3000 partidas jugadas con blancas.

Número de Peones Retrasados		
	Blancas/Prop	Negras/Riv
Ninguno	27.5 % de las posiciones	23.7 % de las posiciones
Uno	40.6 % de las posiciones	39.3 % de las posiciones
Dos	25.6 % de las posiciones	27.7 % de las posiciones
Tres	6 % de las posiciones	8.5 % de las posiciones
Cuatro	0.3 % de las posiciones	0.8 % de las posiciones

Tabla 15.27.: Tabla con información relevante sobre las etiquetas *Número de Peones Blancos Retrasados* y *Número de Peones Negros Retrasados* para un conjunto de 3000 partidas jugadas con blancas.

15. Resultados Experimentales y Análisis

Número de Peones Pasados		
	Blancas/Prop	Negras/Riv
Ninguno	70.5 % de las posiciones	66.5 % de las posiciones
Uno	22.8 % de las posiciones	23.7 % de las posiciones
Dos	5.5 % de las posiciones	8.1 % de las posiciones
Tres	1.1 % de las posiciones	1.5 % de las posiciones
Cuatro	0.1 % de las posiciones	0.2 % de las posiciones
Cinco	0 % de las posiciones	0 % de las posiciones
Seis	0 % de las posiciones	0 % de las posiciones

Tabla 15.28.: Tabla con información relevante sobre las etiquetas *Número de Peones Blancos Pasados* y *Número de Peones Negros Pasados* para un conjunto de 3000 partidas jugadas con blancas.

Número de Islas de Peones		
	Blancas/Prop	Negras/Riv
Ninguno	0.2 % de las posiciones	0.1 % de las posiciones
Uno	11.1 % de las posiciones	11.1 % de las posiciones
Dos	60.7 % de las posiciones	57.7 % de las posiciones
Tres	26 % de las posiciones	29.2 % de las posiciones
Cuatro	2.0 % de las posiciones	1.9 % de las posiciones

Tabla 15.29.: Tabla con información relevante sobre las etiquetas *Número de Islas de Blancos* y *Número de Islas de Peones Negros* para un conjunto de 3000 partidas jugadas con blancas.

Número de Falanges de Peones		
	Blancas/Prop	Negras/Riv
Ninguno	47.6 % de las posiciones	49.9 % de las posiciones
Uno	42.5 % de las posiciones	40 % de las posiciones
Dos	9.6 % de las posiciones	9.8 % de las posiciones
Tres	0.3 % de las posiciones	0.3 % de las posiciones

Tabla 15.30.: Tabla con información relevante sobre las etiquetas *Número de Islas de Blancos* y *Número de Islas de Peones Negros* para un conjunto de 3000 partidas jugadas con blancas.

Número de Grupos de Peones Conectados		
	Blancas/Prop	Negras/Riv
Ninguno	10 % de las posiciones	9.3 % de las posiciones
Uno	45.5 % de las posiciones	45.6 % de las posiciones
Dos	40.9 % de las posiciones	41.5 % de las posiciones
Tres	3.6 % de las posiciones	3.6 % de las posiciones

Tabla 15.31.: Tabla con información relevante sobre las etiquetas *Número de Grupos de Peones Blancos Conectados* y *Número de Grupos de Peones Negros Conectados* para un conjunto de 3000 partidas jugadas con blancas.

15.2. Análisis

Una vez se han presentado todos los resultados toca analizarlos. En primer lugar, se cuenta con un total de 38 etiquetas y para cada una de ellas se deben de entrenar los diferentes modelos. Si se observan las tablas 15.7, 15.8, 15.9 y 15.10 y, más concretamente, la primera de las columnas donde aparecen las etiquetas, se pueden hacer una serie de distinciones en base a estas. Por un lado, se analizarán las etiquetas comunes sin tener en cuenta el bando y luego el resto, que están formadas por diecisiete etiquetas por bando. Antes de pasar a comparar los resultados se va a analizar cada conjunto por separado.

Los resultados obtenidos con el modelo de la primera representación se recogen en la tabla 15.7, donde se puede observar que, en general, se han obtenido resultados muy buenos a excepción de unos pocos casos. En la segunda columna se puede apreciar que el error es prácticamente insignificante si se utiliza la métrica Accuracy y en la tercera aparece el tanto por uno de acierto para cada una de las etiquetas donde se aprecian que todas rozan la perfección. Sin embargo, los resultados que aparecen en la cuarta columna hacen cambiar algo la opinión respecto a los obtenidos en la tercera, la mayoría son muy buenos, pero algunos, como la pistola de Alekhine o el número de damas negras, no son tan buenos como los de la columna previa. Esto se debe a que la métrica Accuracy no tiene en cuenta el desbalanceo de clases y los resultados que obtiene son ficticios y no se asemejan con la realidad. Por eso, destacando el caso de la pistola de Alekhine ya sea blanca o negra, se puede apreciar que una clase la clasifica bien, que es aquella a la que pertenecen los casos donde no aparece esta estructura en la posición. Los casos donde se da la pistola de Alekhine los clasifica como que no se encuentra esta estructura en la posición, porque no ha sido capaz de distinguir estos casos al aparecer en menos del 1 % del total de posiciones de la base de datos. Los resultados logrados con el modelo de la segunda representación se muestran en la tabla 15.8, en este caso aparecen resultados más variados y el error es mayor, se observa que hay mayor diferencia entre la tercera y la cuarta columna. Por último, los resultados del tercer modelo, el que utiliza la tercera representación, se recogen en la tabla 15.9 y, en líneas generales, se pueden apreciar algunos muy buenos y otros muy pobres, sobre todo, los casos de etiquetas que tienen clases con pocas instancias.

Antes de pasar a analizar los resultados y a compararlos vamos a presentar la tabla 15.10 donde se recogen los resultados de los modelos habiendo empleado la métrica F1-score con ponderación macro para facilitar el análisis. La comparativa la hemos realizado con esta métrica y no con Accuracy, puesto que tiene en cuenta el desbalanceo de clases y en la base de datos hay algunas etiquetas con clases muy desbalanceadas.

En primer lugar, se van a analizar las primeras cuatro etiquetas. Para el caso de la etiqueta ventaja de espacio se puede ver en las gráficas 15.1, 15.3 y 15.5 las diferentes curvas de aprendizaje donde se observa que se evita el sobreajuste gracias al uso de *early stopping*. En estas gráficas también se observa que el número de épocas que ha empleado cada uno de los modelos difiere, debido a que cada modelo requiere de un determinado número de épocas para converger. Gracias a las matrices de confusión de las tablas 15.1, 15.2 y 15.3 se puede apreciar que hay muchos casos donde no se realizan predicciones correctamente, sobre todo, para los modelos de la segunda y tercera representación en el caso de la clase igualdad de espacio. En particular, en la tabla 15.2 se muestra que no se ha predicho de forma correcta ninguna posición donde se diera la igualdad. Para terminar el análisis de esta etiqueta se utilizará la gráfica 15.13 donde se comparan los tres modelos empleando las

15. Resultados Experimentales y Análisis

diferentes métricas, en esta se puede observar que el primer modelo es mucho mejor que cualquiera de los otros dos y donde más se acentúa esta diferencia es en la métrica F1-score con predicción macro. En esta etiqueta las clases se encuentran desbalanceadas aunque no lo están tanto como en otros casos. Al haber solamente dos clases para las etiquetas donde se predice si en la posición hay alfiles del mismo color o de distinto color se obtienen con los modelos de las dos primeras representaciones muy buenos resultados, sin embargo, con el modelo bidimensional los resultados que se alcanzan no son los esperados y el modelo no es bueno. Esto se puede deber a que más del 80 % de los datos pertenecen a una clase y al modelo de la tercera representación le cuesta discriminar esto. Los resultados conseguidos al predecir el número de columnas abiertas son excelentes rozando el 100 % de precisión con cualquiera de los modelos y las métricas usadas.

Ahora se examinarán de manera conjunta los resultados obtenidos para las diferentes etiquetas por bando. Las primeras etiquetas analizadas serán las del número de columnas semiabiertas, donde se puede apreciar para el caso del bando blanco que las predicciones son mucho mejores que para las del bando negro, debido a que en el bando negro hay un mayor desbalanceo en las clases, en particular, en la última clase donde aparecen seis columnas semiabiertas. Una recomendación para obtener mejores resultados sería añadir más posiciones y agrupar en una misma clase si en la posición aparecen seis, siete u ocho columnas semiabiertas, aunque son casos muy poco probables. Para esta etiqueta los mejores resultados se han alcanzado con el modelo de la tercera representación. Para las etiquetas donde se busca conocer el número de piezas se va a analizar por separado el caso del número de peones y el del número de damas, ya que se han presentado gráficas y matrices de confusión para estos casos, en concreto, para los del bando blanco. Todos los modelos obtienen buenos resultados a la hora de predecir el número de peones blancos y peones negros que aparecen sobre el tablero. En las matrices de confusión [15.4](#), [15.5](#) y [15.6](#) se puede comprobar que el modelo tridimensional (primera representación) predice de forma correcta todos los casos, el modelo bidimensional (tercera representación) clasifica correctamente casi todos los casos a excepción de unos pocos para cinco, seis y siete peones, y en la matriz de confusión del modelo de la segunda representación se puede comprobar que hay un cierto número de casos erróneos, acentuándose en el caso de cero peones, debido a que hay pocas instancias. Por tanto, los modelos de la primera y tercera representación son mucho mejores que el de la segunda, esto se puede apreciar mejor en la gráfica [15.16](#). Para el caso del número de damas blancas, se puede apreciar que los mejores modelos son los de la primera y segunda representación donde aparece una diferencia notable con respecto al modelo de la tercera representación [15.15](#). Cabe destacar que el primer modelo no predice nada bien el número de damas negras en comparación con el caso del número de damas blancas y esto se debe a que el desbalanceo en las clases de la etiqueta damas negras es mayor que en el caso del número de damas blancas. Para el resto de etiquetas donde se clasifican el número de piezas se puede apreciar que el modelo de la tercera representación no alcanza buenos resultados y, en general, exceptuando el caso del número de damas negras el que mejor resultados obtiene es el modelo que utiliza la primera representación.

Conviene destacar el caso de la Pistola de Alekhine, los resultados obtenidos sin importar el bando son muy pobres y esto es debido al desbalanceo de clases. Al no haber prácticamente casos donde se produce esta estructura los modelos no son capaces de clasificarla y por eso el resultado es que en todas las posiciones no aparece la estructura. Por tanto, se deben extraer muchas más posiciones donde se produzca esta composición para poder realizar una clasificación adecuada.

Para los casos de torre en séptima, torres dobladas y torres ligadas se puede observar en la tabla 15.10 que para los dos primeros no se obtienen buenos resultados al utilizar el modelo de la tercera representación pero para el resto de casos, en concreto, para el modelo que emplea la primera representación los resultados son magníficos. Conviene destacar que a pesar de la falta de localidad espacial para el modelo de la segunda representación, este predice mejor los resultados que el de la tercera representación.

Por último, solo queda analizar los casos relacionados con las estructuras de peones donde se va a apreciar la importancia de tener una representación que permita mantener la localidad espacial. Se debe destacar el caso del número de peones pasados blancos, el cual ha sido representado gráficamente en 15.16 donde se muestra que el modelo de la primera representación es el mejor y el peor es el modelo de la segunda representación. También se aprecia que la diferencia entre el modelo de la primera representación y los otros dos, aumenta al emplear la métrica F1-score con ponderación macro. La etiqueta peones doblados es la que peor se predice con bastante diferencia y a pesar de contar con menos clases que otras etiquetas al contar con una clase donde aparecen muy pocas instancias, hace que los resultados no sean tan buenos. Para el resto de etiquetas, el modelo de la primera representación consigue muy buenos resultados, el modelo de la segunda representación no obtiene grandes resultados, puesto que en esta representación no se tiene en cuenta la localidad espacial, y esto se ve claramente en el caso de peones pasados. El modelo que utiliza la tercera representación alcanza resultados relativamente buenos menos para el caso de peones pasados.

Una vez analizados todos los casos y viendo los gráficos, matrices y tablas presentados, destacando la tabla 15.10, se puede afirmar que de las representaciones analizadas la más adecuada para representar una posición de ajedrez es la formada por siete tableros, uno por pieza y uno adicional para el turno, aunque para este caso, este último no es muy importante. En general, los resultados obtenidos con el modelo de la primera representación (3D) son los mejores, en segundo lugar aparece el modelo de la tercera representación (2D) y el segundo modelo (1D) es el que presenta peores resultados, por lo que para este experimento la peor representación es la formada por el vector unidimensional. También se ha comprobado que la métrica Accuracy no es una buena métrica a tener en cuenta para esta base de datos, sin embargo, la métrica F1-score sí aporta unos resultados coherentes y en los casos donde se producen resultados pobres, especialmente los logrados por los modelos de la primera representación, es necesario extraer más posiciones que permitan aumentar el número de instancias para aquellas clases donde no hay suficientes datos. También se podrían agrupar clases de ciertas etiquetas, ya que algunas de estas son poco frecuentes en posiciones de ajedrez.

Por último, se realizará el análisis del ejemplo presentado anteriormente, teniendo en cuenta que las posiciones se han dado en partidas entre la jugada veinte y la cincuenta. Las tablas 15.11, 15.31 y todas las que hay entre ellas muestran, para cada una de las etiquetas escogidas, la frecuencia con la que aparece cada clase en una serie de posiciones. Estos datos se han extraído a partir de 3000 posiciones de ajedrez donde a un jugador que comandaba las piezas blancas le tocaba mover y realizó una mala jugada. En las tablas de este caso, se puede observar que no hay un bando que suela tener ventaja de espacio, en la mayoría de partidas la posición es cerrada, al haber pocas columnas abiertas o semiabiertas. Suele haber un alto número de peones y juega tanto el jugador como su rival sin pareja de alfiles y sin pareja de caballos, en concreto, con una pieza de cada. En la mayoría de las ocasiones todavía se mantienen las cuatro torres sobre el tablero donde las torres no aparecen ni

15. Resultados Experimentales y Análisis

ligadas, ni dobladas ni en séptima fila. No se puede sacar información respecto al número de damas, ya que a veces aparecen y otras no. Con respecto a las estructuras de peones, normalmente no aparecen peones doblados y hay muy pocos o ningún peón aislado. El número de peones retrasados suele ser uno y normalmente ningún jugador tiene peones pasados. Suele haber dos o tres islas de peones, ninguna o una falange, y el número de grupos de peones conectados es de uno o ninguno. Por tanto, algunos peones han sido avanzados y se ha producido algún intercambio de peones. Gracias a este estudio y utilizando los modelos desarrollados, se le puede aconsejar a este jugador que se centre y estudie partidas donde se llegan a posiciones con las características o patrones que acaban de ser descritos y así, el jugador será capaz de mejorar y de no cometer los mismos errores.

Parte V.

Conclusiones y Trabajo Futuro

16. Conclusiones y Trabajo Futuro

16.1. Conclusiones

A lo largo de este trabajo se han ido cumpliendo los objetivos iniciales que se marcaron. Sin embargo, para el segundo objetivo de la parte de informática conseguimos desarrollar la página web, pero no logramos alcanzar el número suficiente de respuestas que necesitamos para desarrollar los modelos con los que clasificar las posiciones de ajedrez, ya que obtuvimos tan solo unas seiscientas respuestas. Por ello, consideramos necesaria la implementación de una serie de algoritmos, que se pueden encontrar en el apéndice B, los cuales nos permiten extraer las características de una posición y con estas poder etiquetar las posiciones. De esta forma, conseguimos alcanzar nuestro objetivo final, poder clasificar posiciones de ajedrez y aplicar esta clasificación a un ejemplo real.

A partir de los objetivos desarrollados hemos podido obtener las siguientes conclusiones.

Por una parte, hemos realizado un breve repaso de la historia y de las reglas básicas del ajedrez y hemos estudiado algunas de las notaciones más importantes. También nos hemos adentrado en el ajedrez computacional donde hemos visto sus inicios y las diferentes épocas por las que ha ido pasando. Actualmente se encuentra en la era de la inteligencia artificial, la cual estamos seguros que no será la última. Hemos explicado el concepto de motor de ajedrez y sus principales componentes. En particular, hemos estudiado algunos de los motores más famosos, donde hemos comprobado la importancia de las redes neuronales y el impulso que han supuesto en estos. Las redes neuronales han permitido que los motores de ajedrez evalúen las posiciones de ajedrez de forma más compleja y eficiente dando como resultado una mejor evaluación. Además, han provocado que el estilo de juego de los motores varíe.

Por otra parte, hemos entendido el concepto de algoritmo de aprendizaje y, gracias al teorema de No Free Lunch, sabemos que no existe un algoritmo de aprendizaje universal. Esto es debido a que para cada algoritmo de aprendizaje siempre existirá una distribución de probabilidad en la que falle. Por tanto, necesitaremos incorporar en el algoritmo cierto conocimiento específico del problema con el que trabajemos. Asimismo, el teorema de aproximación universal nos ha permitido conocer el potencial que presentan las redes neuronales, ya que son capaces de aproximar cualquier función continua en un espacio compacto. También hemos comprendido y analizado el funcionamiento de las redes neuronales prealimentadas y de las redes neuronales convolucionales.

Además, hemos desarrollado un proyecto para la clasificación de posiciones de ajedrez. Para ello, comenzamos diseñando una base de datos con posiciones que reunían las características deseadas. Posteriormente, desarrollamos una aplicación web que no tuvo el éxito que hubiéramos querido, debido a esto, tuvimos que implementar una serie de algoritmos para codificar y etiquetar las posiciones de ajedrez. Por último, hemos desarrollado un software basado en aprendizaje profundo mediante redes neuronales que aprendió a clasificar po-

16. Conclusiones y Trabajo Futuro

siciones de ajedrez de acuerdo a una serie de etiquetas. También hemos podido comparar por cada etiqueta los resultados obtenidos de los diferentes modelos, dependiendo de la representación del tablero que han utilizado. En general, los mejores resultados han sido obtenidos por el modelo que emplea la representación formada por siete tableros de tamaño 8x8 (primera representación (3D)). No obstante, para algunas etiquetas hemos podido comprobar que ninguno de los modelos aportaba buenos resultados debido al desbalanceo de las clases de ciertas etiquetas. En conclusión, hemos conseguido clasificar posiciones de ajedrez en base a una serie de etiquetas y estas clasificaciones nos han permitido estudiar para un ejemplo de tres mil posiciones, jugadas todas las partidas con blancas, aquellos patrones de las posiciones que aparecen cuando el jugador comete una mala jugada.

16.2. Trabajo Futuro

En vista de los resultados que hemos logrado se abren multitud de vías para realizar trabajos futuros donde poder mejorar la calidad de los resultados, profundizar en ellos y entender mejor aquellas situaciones complicadas y delicadas que tiene una posición de ajedrez. Así, podremos tener herramientas que nos permitan conocer desde un punto de vista más humano los secretos que esconde. Nos centraremos sobre todo en intentar dar indicaciones para solventar los problemas que nos han surgido, pero como apenas hay documentación relacionada con este trabajo finalizaremos lanzando algunas propuestas sobre posibles trabajos que no están enfocados en los problemas que nos hemos encontrado.

En primer lugar, debemos destacar el principal problema que tenemos, partimos de una base de datos propia, la cual requiere un crecimiento en el número de instancias para seguir avanzando en la búsqueda de mejores resultados. Esta base de datos es algo escasa al contar con cincuenta mil posiciones cuando vimos que el número de partidas diferentes que pueden desarrollarse en el juego del ajedrez es mayor que todos los átomos del universo y según Shirish Chinchalkar [17] el límite superior para el número de posiciones alcanzables es 10^{46} . Por tanto, el número de posiciones posibles asciende muy por encima de la cifra que hemos manejado.

Si bien es cierto que existen numerosas bases de datos de partidas de ajedrez de las cuales podemos extraer las posiciones que nos interesan, en nuestro caso aquellas que son previas a una jugada mala, no hemos encontrado una base de datos para las características que buscamos. Por todo esto, hace falta un gran trabajo de estudio para seguir investigando, ya que es necesario filtrar las posiciones de las partidas que aparecen en las diferentes bases de datos para obtener una base de datos de posiciones que reúna los requisitos. Una vez obtenida esta base de datos se podrían filtrar las posiciones por la fase de la partida en la que se encuentran, permitiendo así realizar una mejor clasificación.

Por otra parte, al contar en este caso con 38 problemas de clasificación hemos comprobado que algunos de ellos no tienen clases bien balanceadas, por ejemplo para el problema de clasificación del número de damas blancas o negras tenemos muy pocas instancias en la base de datos con posiciones en las que aparecen dos damas blancas o dos damas negras sobre el tablero. Este es otro problema de la base de datos que tenemos, ya que a pesar de que las etiquetas consideradas han sido muy generales y que algunas clases de cada una hayan sido descartadas sigue siendo necesario aumentar el volumen de la base de datos. Ante esto, planteamos una serie de propuestas como por ejemplo, agrupar algunas clases de

ciertas etiquetas como el número de columnas abiertas, agrupando estas cuando hay más de seis, ya que esta posición es una posición abierta y tal vez no sería necesario realizar una mayor división, etc. Otra propuesta sería que al filtrar la base de datos por fases de la partida podamos escoger aquellas clases de las diferentes etiquetas que se correspondan con la fase del juego, ya que por ejemplo en la apertura no van a aparecer dos damas de un mismo bando. Como hemos podido comprobar, nuestros modelos son buenos en unos casos y bastante pobres en otros, por lo que una vez hayamos aumentado el volumen de datos a estudiar proponemos aplicar un estudio experimental mucho más amplio o discernir cada tipo de característica a considerar en la posición entrenando cada una con un modelo más específico permitiendo obtener mejores resultados.

Si nos vamos a centrar en analizar posiciones concretas de un jugador necesitamos saber qué posiciones suele jugar y aquellas en las que se suele equivocar para poder extraer esta serie de etiquetas con las que poder ayudarlo. También podemos incrementar el número de características a analizar en una posición de ajedrez haciendo que aumente el número de etiquetas a considerar para especificar mejor la situación de dicha posición. Algunas etiquetas que podemos tener en cuenta son la conectividad de las piezas como la batería alfil y dama, si el rey tiene peones protegiéndolo, etc. Asimismo, podemos tener en cuenta motivos tácticos además de posicionales como los rayos x, las sobrecargas, las enfiladas o si una pieza se encuentra atrapada. Para este tipo de etiquetas más complejas recomendamos codificar las posiciones mediante la primera representación del tablero que hemos considerado o por medio de otras similares a esta, como una formada por trece tableros, uno por cada pieza y uno adicional para saber el turno o probar a añadirle tableros de bits para saber los movimientos que pueden realizar, al igual que manejan los motores de ajedrez.

Al no conseguir tener la respuesta que esperábamos de la comunidad de ajedrez también me gustaría proponer como posible trabajo futuro conseguir que esta aporte información a través de la aplicación web desarrollada o a partir de otra semejante. Así, podríamos realizar muchos estudios gracias a la información que puedan aportar, ya sea etiquetando la posición de acuerdo a patrones como los que hemos utilizado en este trabajo y que ellos marquen aquellos que encuentren en la posición permitiendo realizar un estudio semejante al descrito o poder realizar otra clase de estudios como por ejemplo poder profundizar en el estudio que realizaron en 2007 Linhares y Brum [57] clasificando las respuestas por nivel, siendo los jugadores los que clasifiquen las posiciones mediante sus propios criterios y analizar mediante aprendizaje profundo todas las respuestas en busca de patrones en la posición y similitudes entre los diferentes jugadores dependiendo de su nivel, forma de jugar, etc.

Hasta ahora, hemos expuesto posibles vías a seguir en los problemas que nos han surgido y también hemos lanzado algunas propuestas que no tienen que ver con estos. Sabemos que nuestra misión consiste en analizar posiciones de ajedrez para poder entenderlas mejor y saber cuáles son las que provocan que se produzca un error y a partir de estas intentar sacar patrones que ayuden a los jugadores. Gracias a esto y con la ayuda de nuevas etiquetas, un jugador en un futuro no tendrá que seguir el procedimiento que se lleva a cabo hoy en día para mejorar, sino que además de entrenar con módulo para conocer y familiarizarse con nuevas posiciones que no se han explorado anteriormente, podrá analizar el conjunto de posiciones donde más se equivoca, estando estas posiciones agrupadas según unos mismos patrones, facilitando así esta tarea. Incluso, podrá estudiar aquellas posiciones donde los rivales no se encuentran cómodos y cometan imprecisiones, pudiendo preparar aperturas y sistemas con los que jugar para llegar a ese tipo de posiciones donde sabe que su rival suele

16. Conclusiones y Trabajo Futuro

cometer una mala jugada y de esta manera, conseguir su objetivo, la victoria.

A. Apéndice A

A.1. Aplicación Web

Esta aplicación web está desarrollada con el propósito de que jugadores de todo el mundo aporten información sobre diferentes posiciones de ajedrez a través de una serie de etiquetas de motivos posicionales que fueron obtenidas gracias a Chesstempo [83], permitiendo así, poder clasificar las posiciones de ajedrez y con estos datos ser capaces de entrenar redes neuronales de diversos tipos. A pesar de difundir la aplicación por una serie de foros de ajedrez y de que esperamos unos tres o cuatro meses a que la gente pudiera acceder para recabar información, tan solo se conseguieron unos seiscientos datos, algo insignificante para poder entrenar con garantías una red neuronal. Debido a esto, se pensó en otra alternativa para solucionar este problema y se optó por diseñar una serie de algoritmos deterministas B.1 que simularan la información que deberían haber aportado los jugadores. Además, se aumentó el tamaño la base de datos, todo esto lo hemos detallado en el capítulo 12.

Francisco Javier Melero Rus compró un nombre de dominio para poder desplegar la aplicación, el código de la aplicación se puede encontrar en GitHub B.2 y se puede acceder a ella a través del siguiente enlace:

<https://lsi2.ugr.es/fjmelero/chessai/>

A.2. Análisis de Requisitos

En esta sección se exponen de forma individual y detallada el conjunto total de requisitos que componen el sistema. Estos requisitos se dividen en dos, por una parte se encuentran los requisitos funcionales, que son aquellos que completan la funcionalidad pedida por el cliente y están relacionados con el flujo de la aplicación, y por otro lado están los requisitos no funcionales, que son aquellos que están relacionados con todo aquello que no está estrechamente relacionado con la lógica de la aplicación, como la mantenibilidad, el rendimiento, la usabilidad, etc, es decir, determinan el comportamiento de la aplicación.

A.2.1. Requisitos Funcionales

- *RF1. Iniciar sesión:* El usuario debe ser capaz de entrar al sistema al introducir el nivel de Elo y la plataforma.
- *RF2. Modificar datos:* El sistema debe permitir el cambio de datos introducidos en el inicio de sesión.
- *RF3. Seleccionar idioma:* El usuario debe poder seleccionar uno de los tres idiomas disponibles.

A. Apéndice A

- *RF4. Visualización de posiciones:* El usuario debe poder visualizar una posición de ajedrez.
- *RF5 Visualización de etiquetas:* El usuario debe localizar las diez etiquetas que se muestran.
- *RF6. Seleccionar etiquetas:* Esta funcionalidad es de las más importantes, permite al usuario poder marcar aquellas etiquetas que están representadas en la posición de ajedrez.
- *RF7. Cambiar etiquetas no seleccionadas:* El usuario podrá cambiar las etiquetas que no haya seleccionado por otras de la base de datos.
- *RF8. Enviar respuestas:* permite registrar las etiquetas que el jugador ha asociado a una determinada posición.
- *RF8. Continuar etiquetando:* funcionalidad que permite seguir etiquetando posiciones.
- *RF8. Salir:* esta funcionalidad permite al usuario poder salir de la aplicación.

A.2.2. Requisitos no funcionales

- *RNF1. Usabilidad:* la aplicación debe ser de uso sencillo, para que pueda utilizarla cualquier persona, para ello dispondrá de una sencilla interfaz.
- *RNF2. Portabilidad:* la aplicación puede ser utilizada en distintos navegadores y dispositivos sin necesidad de instalar nada.
- *RNF3. Rendimiento:* el tiempo de respuesta debe ser aceptable, tanto para la comunicación con el servidor como para las interacciones internas de la aplicación.
- *RNF4. Integridad:* la información proporcionada y usada ha de ser cuidada para protegerse contra la corrupción y estados inconsistentes.
- *RNF5. Confidencialidad:* el acceso a la aplicación está garantizado para cualquier usuario y no requiere de datos sensibles.
- *RNF6. Disponibilidad:* se garantiza el acceso a los usuarios.

A.3. Diseño

En esta sección se van a desarrollar las técnicas que se han empleado para el diseño de la aplicación. Una vez se ha obtenido la información del sistema se ha elaborado el diseño de la base de datos y el diseño de las vistas de la aplicación mediante bocetos.

A.3.1. Diseño de la Base de Datos

La base de datos desarrollada para este proyecto es una base de datos relacional. Para llevarla a cabo se ha utilizado MySQL, un sistema de gestión de bases de datos realacionales de código abierto respaldado por Oracle y basado en el lenguaje de consulta estructurado SQL. La particularidad de este tipo de bases de datos es que siguen el modelo relacional.

Este modelo está basado en la organización y gestión de los datos en tablas. Estas tablas están formadas por tuplas, que contienen la información relativa a una única entidad.

En este caso, la base de datos se denomina ChessDB, la cual contiene tres tablas como se puede ver en la figura A.1. La tabla etiquetas contiene las 59 etiquetas que han sido extraídas de los motivos posicionales de Chesstempo [83]. La tabla posiciones contiene diez mil posiciones extraídas de la base de datos de [15] las cuales se dan antes de realizar una mala jugada. Por último, la tabla respuestas es la que almacena para cada posición las etiquetas que ha seleccionado el usuario. Las tres tablas tienen como clave primaria su propio identificador para saber la etiqueta, la posición y la respuesta, y estos identificadores son autoincrementables.



Figura A.1.: Diagrama de la base de datos

A.3.2. Bocetos

Se va a presentar mediante unos bocetos básicos, también llamados *wireframes*, la apariencia de nuestra página web. Esta se compone de tres vistas aunque son bastante similares. Al acceder a la aplicación aparece la pantalla de inicio A.2. En ella, se puede observar una cabecera donde aparece el escudo de la Universidad de Granada, ciertos datos relevantes sobre la creación de esta y la explicación del propósito de la aplicación. A la derecha aparecen dos celdas, una para ingresar en la aplicación y poder etiquetar posiciones y otra para cambiar el idioma. Hay tres idiomas disponibles, castellano, inglés y francés. Tras esto, se muestra una extensión de la cabecera donde se explican las instrucciones. Para entrar en la aplicación se deben introducir unas credenciales, las cuales son un número entero que será el nivel de Elo y la plataforma donde se tiene ese nivel.

Una vez se han introducido estos datos la aplicación redirige al usuario a la página principal A.3, en ella aparece una cabecera muy similar a la anterior, ahora ya no aparecen las instrucciones y en su lugar se muestran los datos que ha introducido el usuario en la página de inicio y un botón para modificarlos. En la parte central se encuentra el tablero con una posición de la base de datos y a su lado una serie de etiquetas para seleccionar. Si el usuario detecta que alguna etiqueta está representada en la posición la pulsa y esta quedará seleccionada, se pueden marcar todas las etiquetas que aparecen o la opción ninguna opción. Además, existe un botón que permite cambiar las etiquetas mostradas que no hayan sido seleccionadas por otras de la base de datos, permitiendo así, seleccionar un mayor número

A. Apéndice A

de etiquetas para cada posición.

Tras seleccionar las etiquetas para almacenar las respuestas el usuario pulsará en enviar, y ahora tendrá la posibilidad de seguir etiquetando una nueva posición o de salir de la aplicación. Si decide continuar, volverá a la misma página pero con una nueva posición y nuevas etiquetas, por el contrario, si no quiere seguir etiquetando posiciones de ajedrez, será redirigido a la página de despedida [A.4](#).

Los bocetos de las vistas de móvil son muy similares, las diferencias se encuentran en la cabecera y en la parte central de la página principal. Al contar con dispositivos con menor tamaño de pantalla se debe reducir el ancho de la cabecera y para ello se han agrupado en una columna los componentes de la cabecera y en la parte central aparecerá en primer lugar la posición y posteriormente, las etiquetas con los botones.



Figura A.2.: Wireframe de la página de inicio.

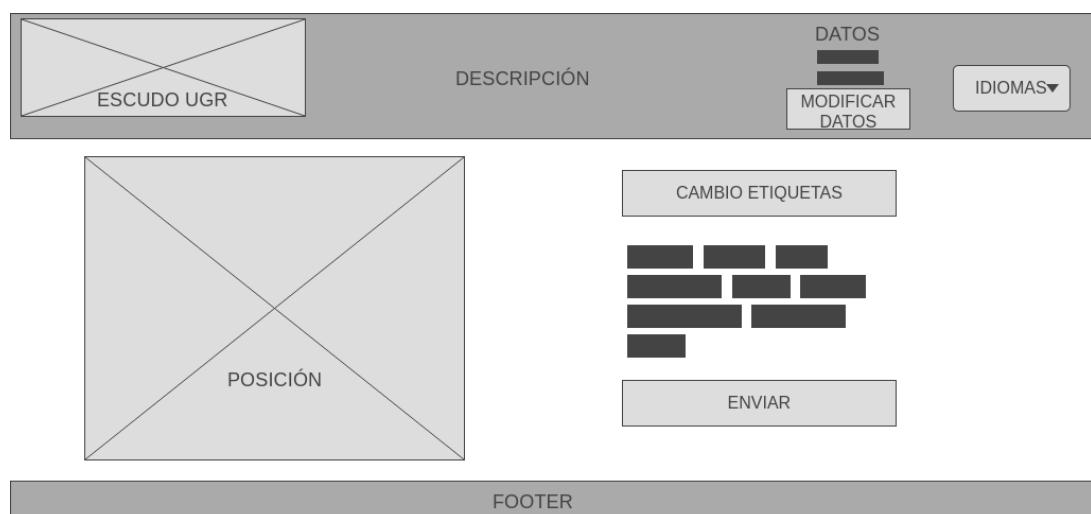


Figura A.3.: Wireframe de la página principal.

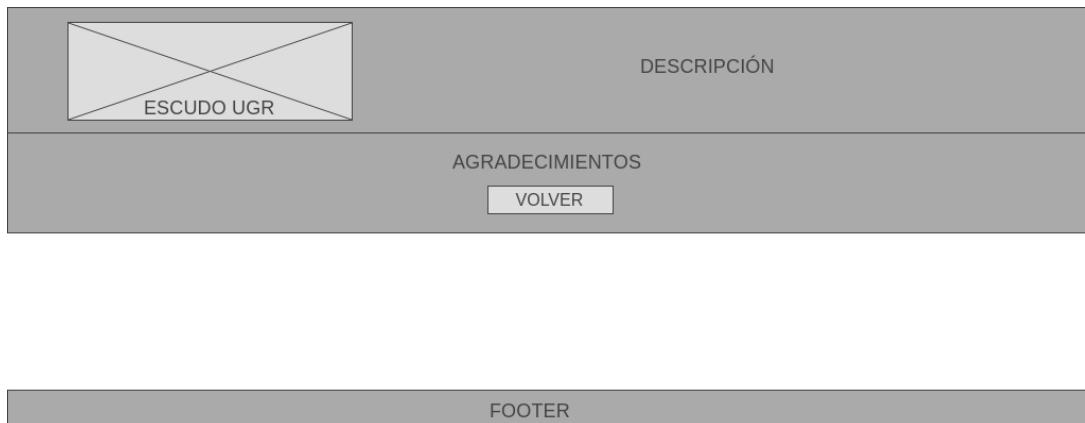


Figura A.4.: Wireframe de la página de despedida.

A.4. Implementación

Ahora que ya se han descrito los requisitos que debe cumplir la aplicación y se ha realizado el diseño de esta, es el turno de realizar la implementación. En primer lugar, se escogieron los lenguajes de programación.

A.4.1. Lenguajes de Programación

Los lenguajes de programación que se han utilizado en la página web han sido:

- *HTML*: es el lenguaje de marcado para la elaboración de páginas web, el componente más básico de estas. Define el significado y la estructura del contenido web, pero no su funcionalidad.
- *PHP*: es un lenguaje de código abierto muy popular especialmente destinado a desarrollar aplicaciones para la web y a crear páginas web y que puede ser incrustado en HTML. Favorece la conexión entre los servidores y la interfaz de usuario.
- *JavaScript*: es un lenguaje de programación ligero, interpretado, o compilado en tiempo de ejecución con funciones de primera clase. Es más conocido como un lenguaje de scripting (secuencia de comandos) para páginas web. Javascript es un lenguaje de programación basada en prototipos, multiparadigma, de un solo hilo, con soporte para programación orientada a objetos y dinámico. Hemos utilizado la librería jQuery de JavaScript.
- *CSS*: es el lenguaje de estilos utilizado para describir la presentación de documentos HTML o XML. CSS describe cómo debe ser renderizado el elemento estructurado en la pantalla.
- *SQL*: es un lenguaje de programación estructurado que da acceso a un sistema de gestión de bases de datos que permite especificar diversos tipos de operaciones en ellos. Este lenguaje es el que hemos utilizado para la implementación de la base de datos.

A. Apéndice A

A.4.2. Software

Para la implementación de la aplicación web se ha utilizado XAMPP, una distribución de Apache que incluye diferentes softwares libres. El nombre es un acrónimo compuesto por las iniciales de los programas que lo constituyen:

- *Linux*: sistema operativo donde estará instalada nuestra aplicación.
- *Apache*: servidor web de código abierto, es la aplicación usada globalmente para la entrega de contenidos web.
- *MySQL/MariaDB*: XAMPP cuenta con uno de los sistemas relacionales de gestión de bases de datos más populares del mundo.
- *PHP*: lenguaje de programación de código de lado del servidor que permite crear páginas web o aplicaciones dinámicas. Es independiente de la plataforma y soporta varios sistemas de bases de datos.
- *Perl*: lenguaje de programación que se usa en la administración del sistema, en el desarrollo web y en la programación de red. También permite programar aplicaciones web dinámicas.

También se ha hecho uso de Bootstrap para el *front-end* (pantalla de interfaz con el usuario), un framework CSS desarrollado por Twitter en 2010. Además, se ha utilizado el editor de código fuente Visual Studio Code.

A.4.3. Estructura de la Aplicación Web

En esta sección se va a explicar la estructura de carpetas que tiene la aplicación web donde se muestran cada uno de los archivos que la conforman y la utilidad de estos. La estructura de carpetas es la siguiente:

- *assets*: esta carpeta contiene otra carpeta con las imágenes de las piezas de ajedrez (una para cada una de las doce piezas). En esta carpeta también se encuentran el resto de imágenes que se han empleado para desarrollar la página web.
- *CSS*: esta carpeta contiene los archivos de las hojas de estilo.
 - *estilo.css*: es la hoja de estilos para la cabecera, el footer, las etiquetas y la ubicación de la posición del tablero.
 - *fen.css*: este archivo contiene la hoja de estilos donde se representa una posición en el tablero, aquí es donde se especifica cómo se verá la posición.
- *lang*: en esta carpeta se encuentran tres archivos, cada uno con los textos en diferentes idiomas.
 - *eng.php*: contiene los textos en inglés.
 - *esp.php*: archivo donde se encuentran los textos en castellano.
 - *fre.php*: contiene los textos en francés.
- *app.js*: este archivo es el que permite visualizar a partir de un código FEN una posición por pantalla. Se van a destacar las funciones que permiten que se lleve a cabo este

propósito. Una de las funciones más importantes es la que permite validar el código FEN. También contiene una función que sirve para crear el tablero y otra para crear las piezas, la cual hace uso de una función que es la que coloca las piezas sobre el tablero. Asimismo, se ha implementado una función que limpia el tablero y lo deja sin piezas. Para saber a quién le toca mover se utiliza una función, y otra permite conocer y mostrar las coordenadas sobre el tablero. Un método que permite marcar las casillas que se corresponden con el movimiento que se acaba de producir. Todas estas funciones son usadas en la función que resetea el tablero para mostrar una nueva posición. Además, este archivo tiene otras funcionalidades, valida que los datos de entrada sean correctos, se encarga de cambiar las etiquetas y muestra u oculta ciertas componentes.

- *database.php*: este archivo de código fuente PHP permite la conexión con la base de datos.
- *index.php*: este archivo se corresponde con la página de inicio donde aparece la cabecera, el footer y las instrucciones.
- *formulario.php*: en este archivo se realizan varias acciones. En primer lugar, una vez nos hemos conectado con la base de datos extraemos una posición y las etiquetas que se visualizarán por pantalla. Una vez realizado esto, se presenta la posición con sus etiquetas y es aquí donde seleccionamos aquellas que se corresponden con la posición. Este archivo contiene la visualización de la parte central de la página principal.
- *insertar.php*: este archivo es el que comprueba que todos los datos son correctos para insertar nuevos registros en la base de datos. Si todos los datos son correctos realiza la inserción.
- *despedida.php*: este archivo de código fuente PHP es el que muestra la página de despedida y contiene un botón que permite al usuario regresar a la página principal.

A.5. **Vistas**

En esta sección se presentan las vistas que aparecen en nuestra página web. En primer lugar, se muestran las vistas desde la web y posteriormente las vistas desde un móvil. La primera figura representa la vista de la página de inicio ([A.5](#), [A.9](#)). Posteriormente aparecen dos figuras, en una aparece la cabecera una vez se han ingresado los datos ([A.6](#), [A.10](#)) y en otra la parte central con la posición y las etiquetas ([A.7](#), [A.11](#)). Finalmente, se muestra la vista de despedida ([A.8](#), [A.12](#)).

A. Apéndice A

A.5.1. Vistas Web



Figura A.5.: Vista web de la página de inicio.



Figura A.6.: Vista web de la cabecera una vez hemos introducido los datos



Figura A.7.: Vista web de la parte central de la página principal.



Figura A.8.: Vista web de la página de despedida.

A. Apéndice A

A.5.2. Vistas Móvil

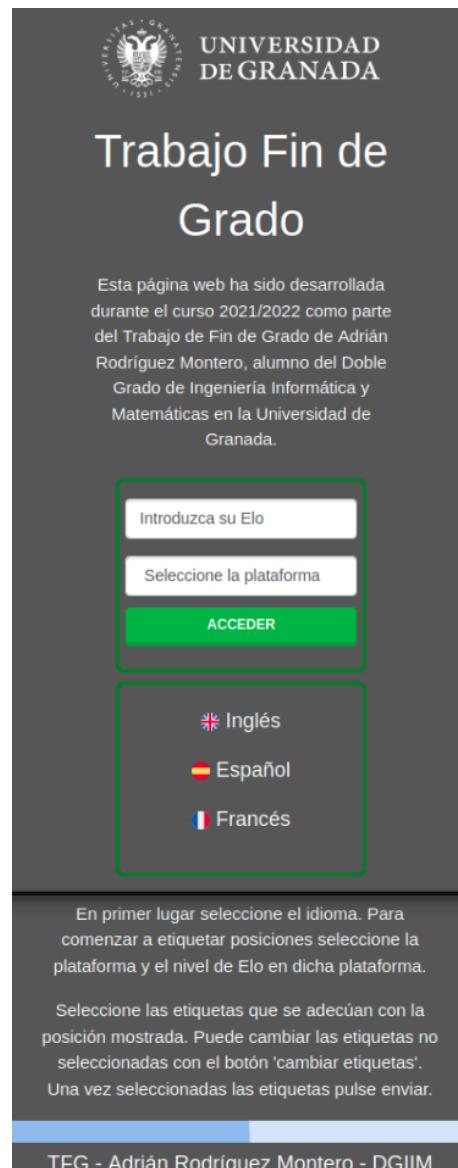


Figura A.9.: Vista móvil de la página de inicio.



Figura A.10.: Vista móvil de la cabecera una vez hemos introducido los datos

A. Apéndice A



Figura A.11.: Vista móvil de la parte central de la página principal.

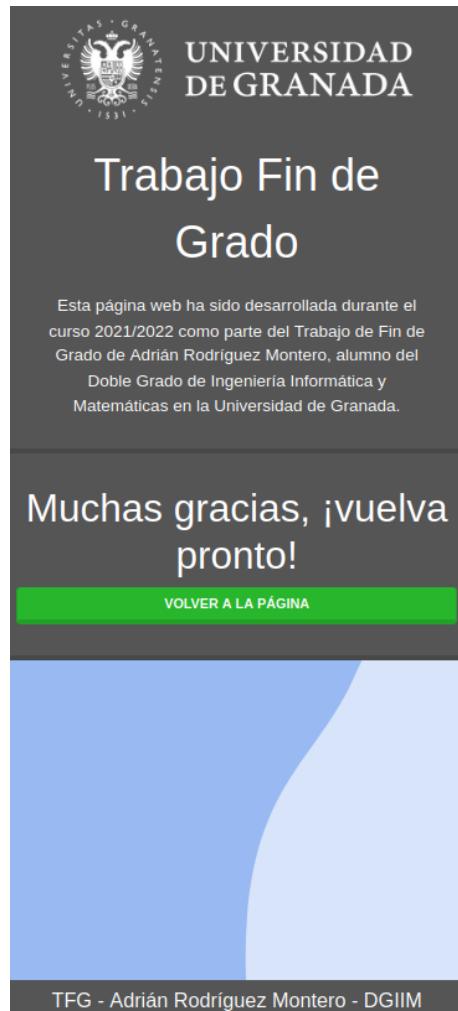


Figura A.12.: Vista móvil de la página de despedida.

A. Apéndice A

A.6. Foros

En esta sección se adjunta una imagen de una de las publicaciones que se realizaron en el foro Chess.com [16].

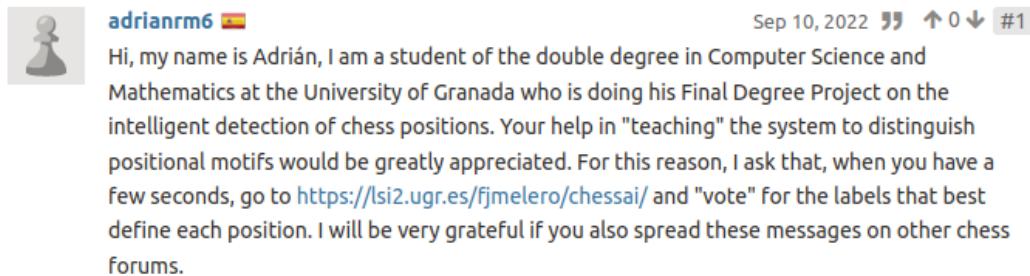


Figura A.13.: Publicación realizada en el foro Chess.com.

B. Apéndice B

B.1. Algoritmos

En esta sección se presentan los algoritmos que han sido implementados para obtener las representaciones del tablero y las etiquetas necesarias para entrenar los modelos. Estos algoritmos se encuentran en un único fichero con extensión *.cpp*, ya que se ha hecho uso del lenguaje de programación C++. Este es un lenguaje de programación compilado, multiparadigma, principalmente de tipo imperativo y orientado a objetos.

El fichero que se va describir se puede encontrar en GitHub (ver [B.2](#)) en la carpeta *Algoritmos*. En este archivo se han trabajado con varias estructuras de datos, tanto *arrays* como *vectores*. En primer lugar, al abrir el documento aparecen una serie de variables globales, como son las filas y las columnas y también el número de piezas distintas que puede haber en una posición, en total doce, ya que hay seis piezas distintas por bando. Posteriormente se aprecian una serie de constantes, cada una de ellas se corresponde con un tipo de pieza y estas tienen asignado un valor del cero al once, ambos inclusive. Esto es así, ya que se hace uso de un vector de vectores donde cada vector contendrá la posición de un mismo tipo de pieza. Por ejemplo, el primer vector será el que contenga la posición del rey blanco, el segundo, la posición de la dama o las damas blancas si es que hay, en caso de no haber damas blancas no se almacena ninguna casilla en el vector de damas blancas y así con el resto de piezas. Para almacenar las casillas se ha realizado una codificación especial, las casillas se ordenarán con números del 11 al 88 ambos inclusive. Para que los algoritmos sean más sencillos de implementar se han numerado las casillas teniendo en cuenta que las casillas de la primera columna terminan en 1, las de la segunda en 2 y así hasta las de la octava columna que terminan en 8. La primera fila queda numerada partiendo de la casilla 11 y terminando en la casilla 18, la segunda va de la casilla 21 a la 28, así, hasta la octava fila, que queda numerada de la casilla 81 a la 88. En la siguiente figura se puede ver un ejemplo de lo que se acaba de describir:

81	82	83	84	85	86	87	88
71	72	73	74	75	76	77	78
61	62	63	64	65	66	67	68
51	52	53	54	55	56	57	58
41	42	43	44	45	46	47	48
31	32	33	34	35	36	37	38
21	22	23	24	25	26	27	28
11	12	13	14	15	16	17	18

Figura B.1.: Codificación empleada para saber las casillas en las que se encuentran las piezas.

B. Apéndice B

Al emplear esta codificación y al haber almacenado las casillas de cada pieza en el vector de vectores, para saber si se encuentran dos peones en la misma columna tan solo se realiza una operación módulo 10 sobre la numeración de ambas casillas, si en módulo 10 coinciden significa que están en la misma columna, en caso contrario, se encuentran en diferentes columnas. Por tanto, ya se ha explicado la manera en la que se almacenan las piezas y la codificación que se utiliza para saber en qué escaques del tablero se encuentran.

Antes de llevar a cabo todo lo que se acaba de explicar se necesita disponer de las diferentes posiciones de la base de datos. Estas han sido obtenidas leyendo el fichero con extensión *.txt* el cual se denomina *FEN_DB.txt* donde se alojan las cincuenta mil posiciones de la base de datos. Para obtener estas posiciones se leerán los datos gracias a la clase *fstream*. Una vez leídos se obtendrán el primer campo del código fen donde se representa la posición y el tercer campo donde se indica si hay o no enroques y por cada posición de ajedrez se realizará lo descrito anteriormente, obtener el vector de vectores. Este vector de vectores se ordenará de manera que las posiciones de cada vector aparezcan en orden ascendente, de la menor casilla a la mayor para cada una de las piezas, gracias al método *sort*. Una vez hecho esto, se obtendrán las etiquetas y las codificaciones para representar los tableros gracias a los diferentes métodos que han sido implementados y estas se almacenarán en el archivo *Etiquetas.csv*. Este archivo contendrá más información además de las etiquetas que se han descrito en la sección 12.3, por ejemplo, además del número de columnas abiertas aparecerán marcadas aquellas que están abiertas, o también se guardará una representación del tablero donde se marcará la posición de los peones que están conectados, etc. Para la parte de la obtención de las codificaciones de los tableros que servirán como datos de entrada para los modelos se abrirán dos ficheros, gracias de nuevo a la clase *fstream*, uno por cada forma de representar el tablero y en cada uno de ellos se almacenarán las codificaciones de cada posición gracias al método *ImprimirTablero*, generando los archivos *PrimeraCodificacionTablero.csv* y *SegundaCodificacionTablero.csv*. Por tanto, gracias a estos algoritmos se conseguirá obtener para cada posición sus etiquetas y sus representaciones, y se almacenarán en los archivos con extensión *csv*.

B.2. Código

Todo el código implementado para la realización de este trabajo se puede encontrar en GitHub, en concreto:

<https://github.com/adrianrm6/TFG>

B.3. Redacción del documento

El documento que está leyendo ha sido redactado en L^AT_EX y la plantilla escogida se encuentra ubicada en el siguiente repositorio de GitHub:

<https://github.com/latex-mat-ugr/Plantilla-TFG>.

Bibliografía

- [1] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data*, volume 4. AMLBook New York, 2012.
- [2] Ajedrezutea. Problemas de ajedrez. <https://ajedrezutea.com/>. Recurso online.
- [3] Elizabeth Sianquez Bautista. Elementos del juego. <http://ajedrezesb.blogspot.com/p/elementos-del-juego.html>. Recurso online.
- [4] Fernando Berzal. *Redes neuronales & deep learning*. Independently published, 2018.
- [5] Merim Bilalić and Fernand Gobet. They do what they are told to do: The influence of instruction on (chess) expert perception—commentary on linhares and brum (2007). *Cognitive Science*, 33(5):743–747, 2009.
- [6] Haim Brezis and Haim Brézis. *Functional analysis, Sobolev spaces and partial differential equations*, volume 2. Springer, 2011.
- [7] Raúl E. López Briega. Libro online de la comunidad de inteligencia artificial argentina. <https://iaarbook.github.io/>. Recurso online.
- [8] Britannica. Encyclopedia britannica. <https://www.britannica.com/>. Recurso online.
- [9] Keith T. Butler. Example of overfitting and underfitting in machine learning. https://keeeto.git hub.io/blog/bias_variance/. Recurso online.
- [10] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.
- [11] Pedro Castro. Motores de ajedrez. <https://sites.google.com/site/motoresdeajedrez/ajedrez-con-ordenador/intefaces>. Recurso online.
- [12] Augustin Cauchy et al. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.
- [13] Antoine Champion. Diseccionando Stockfish Parte 1: Una mirada en profundidad a un motor de ajedrez — ichi.pro. <https://ichi.pro/es/diseccionando-stockfish-parte-1-una-mirada-en-profundidad-a-un-motor-de-ajedrez-140590685107577>. Recurso online.
- [14] William G Chase and Herbert A Simon. Perception in chess. *Cognitive psychology*, 4(1):55–81, 1973.
- [15] ChessBase. Chessbase. <https://es.chessbase.com/>. Recurso online.
- [16] Chess.com. Chess.com. <https://www.chess.com/>. Recurso online.
- [17] Shirish Chinchalkar. An upper bound for the number of reachable positions. *J. Int. Comput. Games Assoc.*, 19:181–183, 1996.
- [18] Pratik Choudhari. Understanding convolution operations in cnn. <https://medium.com/analytics-vidhya/understanding-convolution-operations-in-cnn-1914045816d4>. Recurso online.
- [19] Joe H Condon and Ken Thompson. Belle chess hardware. In *Advances in computer chess*, pages 45–54. Elsevier, 1982.
- [20] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

Bibliografía

- [21] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [22] Maciej A Czyzowski, Artur Laskowski, and Szymon Wasik. Chessboard and chess piece recognition with the support of neural networks. *arXiv preprint arXiv:1708.03898*, 2017.
- [23] Paul Dailly, Dominik Gotojuch, Neil Henning, Keir Lawson, Alec Macdonald, and Tamerlan Tadjdinov. A chess engine. 11 2022.
- [24] Jon Dart. Chess programming wiki. <https://www.chessprogramming.org/>, 2018. Recurso online.
- [25] Leonid Datta. A survey on activation functions and their relation with xavier and he normal initialization. *arXiv preprint arXiv:2004.06632*, 2020.
- [26] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, 27, 2014.
- [27] Omid E David, Nathan S Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks*, pages 88–96. Springer, 2016.
- [28] Omid E David, H Jaap van den Herik, Moshe Koppel, and Nathan S Netanyahu. Genetic algorithms for evolving computer chess programs. *IEEE transactions on evolutionary computation*, 18(5):779–789, 2013.
- [29] Adriaan D de Groot. Thought and choice in chess: An overview of a study based on selzean theory. *His Contribution to Psychology*, page 192, 1965.
- [30] Adriaan D De Groot. Thought and choice in chess. In *Thought and Choice in Chess*. De Gruyter Mouton, 2014.
- [31] Adriaan D De Groot, Fernand Gobet, and Riekent W Jongman. *Perception and memory in chess: Studies in the heuristics of the professional eye*. Van Gorcum & Co, 1996.
- [32] Universidad de Sevilla. Dpto. de ciencias de la computación e inteligencia artificial. <https://www.cs.us.es/>. Recurso online.
- [33] Historia del Ajedrez. Historia del ajedrez. <https://historiadelaajedrez.com/>. Recurso online.
- [34] dpthegrey. Best first search heuristic search technique. <https://medium.com/@dpthegrey>. Recurso online.
- [35] estructurasite. Árboles. <https://estructurasite.wordpress.com/arbol/>. Recurso online.
- [36] Fei-Fei Li, Justin Johnson, Serena Young , Stanford University. cs231n. <http://cs231n.stanford.edu/2017/syllabus.html>, 2017.
- [37] Peter W Frey and Peter Adesman. Recall memory for visually presented chess positions. *Memory & Cognition*, 4(5):541–547, 1976.
- [38] Julio Ganzo. *Historia General del Ajedrez*, volume 3. Ricardo Aguilera, 1973.
- [39] GeeksforGeeks. Cnn introduction to pooling layer. <https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/>. Recurso online.
- [40] Fernand Gobet and Herbert A Simon. Recall of random and distorted chess positions: Implications for the theory of expertise. *Memory & cognition*, 24(4):493–503, 1996.
- [41] Fernand Gobet and Herbert A Simon. Five seconds or sixty? presentation time in expert memory. *Cognitive science*, 24(4):651–682, 2000.
- [42] Sonja Musser Golladay. *Los libros de ajedrez dados e tablas: Historical, artistic and metaphysical dimensions of Alfonso X's "Book of Games"*. The University of Arizona, 2007.

- [43] Camilo José Carrillo González and Escola Técnica Superior de Enxeñeiros Industriais. *Fundamentos del análisis de Fourier*. GAMESAL, 2003.
- [44] Francisco Javier Pérez González. Cálculo vectorial series de fourier variable compleja. *Granada: Universidad de Granada*, 2007.
- [45] Gonzalo. La Historia de los Motores de Ajedrez — jugadoresdeajedrez.com. <https://jugadoresdeajedrez.com/informacion-general/historia-de-los-motores-de-ajedrez/>. Recurso online.
- [46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [47] Leonardo Ferreira Guilhoto. An overview of artificial neural networks for mathematicians. 2018.
- [48] F-h Hsu, Thomas S Anantharaman, Murray S Campbell, and Andreas Nowatzyk. Deep thought. In *Computers, Chess, and Cognition*, pages 55–78. Springer, 1990.
- [49] iChess. Aprende a jugar ajedrez. <https://www.ichess.es/>. Recurso online.
- [50] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [51] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.
- [52] Achim Klenke. *Probability theory: a comprehensive course*. Springer Science & Business Media, 2013.
- [53] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.
- [54] Pablo León-Villagrá and Frank Jäkel. Categorization and abstract similarity in chess. In Markus Knauff, Michael Pauen, Natalie Sebanz, and Ipke Wachsmuth, editors, *Proceedings of the 35th Annual Meeting of the Cognitive Science Society, CogSci 2013, Berlin, Germany, July 31 - August 3, 2013*. cognitivesciencesociety.org, 2013.
- [55] Lichess. Base de datos de lichess.org. <https://database.lichess.org/>, 2013. Recurso online.
- [56] Andrés Cerón Lillo. El desafío de arimaa, 2010.
- [57] Alexandre Linhares and Paulo Brum. Understanding our understanding of strategic scenarios: What role do chunks play? *Cognitive Science*, 31(6):989–1007, 2007.
- [58] Alexandre Linhares and Paulo Brum. How can experts see the invisible? reply to bilalić and gobet. *Cognitive Science*, 33(5):748–751, 2009.
- [59] Guy Lories. Recall of random and non random chess positions in strong and weak chess players. *Psychologica Belgica*, 1987.
- [60] Tony A Marsland. Computer chess methods. *Encyclopedia of Artificial Intelligence*, 1:159–171, 1987.
- [61] Fenil Mehta, Hrishikesh Raipure, Shubham Shirsat, Shashank Bhatnagar, and Bailappa Bhovi. Predicting chess moves with multilayer perceptron and limited lookahead. *Journal of Engineering Research and Applications*, 10(4):05–08, 2020.
- [62] Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.
- [63] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.
- [64] H.J.R. Murray. *A History of Chess*. Clarendon Press, 1913.
- [65] Yu Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 2018.

Bibliografía

- [66] Afonso de Sá Delgado Neto and Rafael Mendes Campello. Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 152–160. IEEE, 2019.
- [67] Monty Newborn. *Kasparov versus Deep Blue: Computer chess comes of age*. Springer Science & Business Media, 2012.
- [68] Barak Oshri and Nishith Khandwala. Predicting moves in chess using convolutional neural networks, 2016.
- [69] Maria Pinargote. Unidad 4 redes neuronales. <https://sites.google.com/site/mayinteligenciaartificial/unidad-4-redes-neuronales>. Recurso online.
- [70] Armando Reyes Villena. *Análisis de Fourier, Tema 7*. Departamento de Análisis Matemático, Universidad de Granada, 2020.
- [71] Rom77. Stockfish nn release (nnue). <http://talkchess.com/forum3/viewtopic.php?f=2&t=74059&start=139>. Recurso online.
- [72] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [73] Pertti Saariluoma. Location coding in chess. *The Quarterly Journal of Experimental Psychology Section A*, 47(3):607–630, 1994.
- [74] Fernando Sancho Caparrini. Redes neuronales: una visión superficial. <http://www.cs.us.es/~fsancho/?e=72>, 2022. Recurso online.
- [75] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [76] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.
- [77] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12):310–316, 2017.
- [78] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- [79] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [80] Herbert A Simon and Kevin Gilmartin. A simulation of memory for chess positions. *Cognitive psychology*, 5(1):29–46, 1973.
- [81] Miguel Sotaquirá. La convolución en las redes convolucionales. <https://www.codificandobits.com/blog/convolucion-redes-convolucionales/>. Recurso online.
- [82] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [83] Chess Tempo. Chess tempo. <https://chesstempo.com/>. Recurso online.
- [84] Sebastian Thrun. Learning to play the game of chess. *Advances in neural information processing systems*, 7, 1994.
- [85] Alan M Turing. Chess. In *Computer Chess Compendium*, pages 14–17. Springer, 1988.

Bibliografía

- [86] Joel Vikström. Training a convolutional neural network to evaluate chess positions, 2019.
- [87] Wikipedia. Wikipedia. <http://es.wikipedia.org/w/index.php?title=Wikipedia%20en%20espa%C3%B1ol&oldid=146829787>, 2022. Recurso online.
- [88] WikiWand. Wikiwand. <https://www.wikiwand.com/>. Recurso online.
- [89] Georg Wölflein and Ognjen Arandjelović. Determining chess game state from an image. *Journal of Imaging*, 7(6):94, 2021.
- [90] JLPM y Rey Blanco. Notación fen. <https://www.ajedrez365.com/2020/12/notacion-fen-o-codigo-fen.html>. Recurso online.