



# **ANÁLISIS E IMPLEMENTACIÓN DE ALPHAZERO**

**Alejandro Manuel Trujillo Sánchez**





# ANÁLISIS E IMPLEMENTACIÓN DE ALPHAZERO

Alejandro Manuel Trujillo Sánchez

Memoria presentada como parte de los requisitos para la obtención de los títulos de Grado en Matemáticas y Grado en Ingeniería informática. Tecnologías informáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Tutor Fernando Sancho Caparrini  
**Dpt. Ciencias de la Computación  
e Inteligencia Artificial**



# Índice general

<b>Abstract</b>	<b>1</b>
<b>Resumen</b>	<b>3</b>
<b>Estructura de la Memoria</b>	<b>5</b>
<b>1. Contexto histórico</b>	<b>7</b>
1.1. Introducción . . . . .	7
1.2. Proceso histórico . . . . .	7
1.3. Precursores . . . . .	8
1.3.1. Deep Blue . . . . .	8
1.3.2. Stockfish . . . . .	8
1.3.3. AlphaGo . . . . .	9
1.4. Ajedrez por computador: algo de historia . . . . .	9
1.5. Consecuencias . . . . .	10
<b>2. Conceptos generales</b>	<b>11</b>
2.1. Algoritmos de Búsqueda . . . . .	11
2.1.1. Minimax . . . . .	11
2.1.1.1. Poda $\alpha - \beta$ . . . . .	19
2.1.2. Búsqueda de Montecarlo . . . . .	22

## **II ANÁLISIS E IMPLEMENTACIÓN DE ALPHAZERO**

2.1.2.1.	Problema de las Tragaperras Múltiples . . . . .	22
2.1.2.2.	Algoritmo MCTS . . . . .	23
2.2.	Redes Neuronales . . . . .	28
2.2.1.	Nodos y capas . . . . .	28
2.2.1.1.	Función de activación . . . . .	28
2.2.1.2.	Propagación hacia atrás . . . . .	32
2.2.2.	Red Neuronal Convolutinal . . . . .	36
2.2.3.	Red Neuronal Residual . . . . .	38
2.2.4.	Teorema de Aproximación Universal . . . . .	39
2.2.5.	Teorema No Free Lunch . . . . .	45
2.3.	Aprendizaje por Refuerzo . . . . .	49
2.3.1.	Modelado con Procesos de Decisión de Markov . . . . .	55
<b>3. Análisis General de AlphaZero</b>		<b>57</b>
3.1.	Introducción . . . . .	57
3.1.1.	Estructura General . . . . .	58
3.2.	Representación del Juego . . . . .	63
3.3.	Redes de Tácticas y Valores . . . . .	64
3.4.	MCTS en AlphaZero . . . . .	65
3.5.	Rendición Temprana . . . . .	66
3.6.	Configuración . . . . .	66
3.7.	Evaluación . . . . .	66
<b>4. Implementación</b>		<b>69</b>
4.1.	Reglas del Juego . . . . .	69
4.1.1.	Implementación . . . . .	69
4.2.	Modelos . . . . .	73

4.2.1. Implementación . . . . .	73
4.3. MCTS . . . . .	81
4.4. Algoritmo Original: Implementación y Ajustes . . . . .	89
4.5. Consola . . . . .	92
<b>5. Resultados</b>	<b>95</b>
5.1. Enfrentamiento entre Algoritmos . . . . .	95
5.2. Enfrentamiento contra Humanos . . . . .	100
<b>Conclusión (español)</b>	<b>103</b>
<b>Conclusion (english)</b>	<b>105</b>
<b>Anexos</b>	<b>111</b>
Ruido de Dirichlet . . . . .	113
Juego perfecto . . . . .	114
Estimadores de arquitecturas óptimas . . . . .	115
Problemas de gradiente e inicialización Xavier . . . . .	121



# Abstract

In recent years AI has made significant steps, one of those being AlphaZero's development, a general purpose reinforcement learning algorithm that has demonstrated unprecedented capabilities to surpass humans in complex games, such as chess.

AlphaZero's innovation lies in its unique approach to learning. Unlike previous AIs that relied in human expertise and already programmed knowledge, AlphaZero uses a neural network combined with Montecarlo tree search to learn from scratch. Without prior knowledge, AlphaZero is able to play millions of game after only receiving the game rules, improving in each iteration.

This grabbed the attention of the scientific community not only because of its results, which were outstanding, but also because of the algorithms it uses which could be replicated in various fields such as optimization, decision-making, autonomous systems...

The purpose of this project is to inquire into AlphaZero's architecture, exploring how it represents a huge leap forward in AI and what it means for the future of intelligent systems. By examining it, taking special focus on chess, the objective is to gain a deeper understanding on how reinforcement learning and neural networks can achieve superhuman performance, and how it can be applied to solve real world problems.

# Keywords

- AlphaZero
- DeepMind
- Reinforcement learning
- Montecarlo Tree Search (MCTS)
- Dots and Boxes
- Computer chess
- Convolutional Neural Network (CNN)



# Resumen

En los últimos años, la inteligencia artificial ha avanzado significativamente, entre otros motivos, por el desarrollo de AlphaZero, un algoritmo de aprendizaje por refuerzo de propósito general que ha demostrado capacidades sin precedentes al superar el rendimiento humano en juegos como el ajedrez.

La innovación de AlphaZero radica en su enfoque único para aprender. A diferencia de las IA anteriores que dependían de la experiencia humana y del conocimiento preprogramado, AlphaZero utiliza una red neuronal combinada con una búsqueda de árbol de Montecarlo para aprender desde cero. Sin conocimiento previo, AlphaZero es capaz de jugar millones de partidas después de solo recibir las reglas del juego, mejorando en cada iteración.

Esto ha captado la atención de la comunidad científica no solo por sus resultados sobresalientes, sino también porque los elementos que utiliza podrían replicarse en diversos campos como la optimización, la toma de decisiones y los sistemas autónomos.

El propósito de este proyecto es indagar en la arquitectura de AlphaZero, explorando cómo representa un avance significativo en la IA y lo que significa para el futuro de los sistemas inteligentes. Al examinarlo, con un especial enfoque en el ajedrez, el objetivo es obtener una comprensión más profunda de cómo el aprendizaje por refuerzo y las redes neuronales pueden lograr un rendimiento sobrehumano y cómo estos métodos pueden aplicarse para resolver problemas del mundo real.

## Palabras clave

- AlphaZero
- DeepMind
- Aprendizaje por refuerzo
- Búsqueda de Montecarlo (MCTS)
- Dots and Boxes
- Ajedrez en ordenador
- Red neuronal convolucional (CNN)



# Estructura de la Memoria

El objetivo de la memoria es conocer la estructura de AlphaZero y razonar sobre su importancia en el avance de la inteligencia artificial. Se usarán diversas fuentes, sirviendo [30] como base. La memoria se dividirá en distintos capítulos con diversas secciones que tratarán de explicar el cómo y el por qué de todo lo que envuelve al algoritmo.

En el primer capítulo explicaremos el contexto histórico en el que se desarrolló y qué supuso, haremos un repaso con los primeros avances en la inteligencia artificial específicamente en el ámbito del ajedrez con computadoras y veremos qué consecuencias tuvieron cada uno.

En el segundo capítulo revisaremos conceptos generales sobre elementos usados en AlphaZero. Hablaremos por un lado sobre algoritmos de búsqueda y seguiremos el camino trazado en el desarrollo de gran parte de los algoritmos usados en el ajedrez por computadores tratando de explicar cómo se originaron y por qué funcionan. También hablaremos de conceptos básicos de las redes neuronales, tratando de explicar su funcionamiento y definir los elementos que la conforman, para posteriormente analizar las usadas en AlphaZero y explicar por qué funcionan. Y finalmente veremos conceptos generales de aprendizaje por refuerzo y su formalización como procesos de decisión de Markov.

En el tercer capítulo analizaremos la arquitectura de AlphaZero, desde la representación del estado de juego hasta el proceso de creación de las redes y la evaluación de los movimientos.

En el cuarto capítulo, tras haber comprendido cómo funciona AlphaZero, trataremos de explicar una implementación del algoritmo pero esta vez para el juego Dots and Boxes, que nos permitirá visualizar cómo el algoritmo es capaz de jugar a cualquier juego con tan solo conocer las reglas que lo componen. Posteriormente definiremos los añadidos implementados al juego y las instrucciones para su ejecución.

En el quinto capítulo, veremos los resultados obtenidos, trataremos de analizarlos y ver en qué destaca AlphaZero, además de realizar un análisis crítico de las observaciones obtenidas.



# 1 | Contexto histórico

## 1.1 Introducción

El estudio del ajedrez en computadores ha sido uno de los retos a los que más atención se le ha prestado desde sus inicios, convirtiéndose en un desafío muy interesante desde el punto de vista de la inteligencia artificial que ha captado el interés de científicos como John Von Neumann o Alan Turing.

No es de extrañar que el ajedrez se haya considerado la mosca de la fruta del mundo de la inteligencia artificial, marcando el proceso de avance dentro del conocimiento en la ingeniería. Según [18], esto es debido a que, como la mosca de la fruta, es una forma accesible, familiar y relativamente simple en la que tecnología experimental puede ser usada para producir conocimiento útil en otros sistemas más complejos.

La idea se remonta alrededor del 1769, donde se construye la máquina "El Turco", que se convierte en uno de los mayores engaños de ese periodo, y no es hasta el 1912 en el que se hacen los primeros pasos reales por Leonardo Torres Quevedo, que construye una máquina capaz de jugar finales de rey y torre contra rey.

Esto poco a poco llevó a programas que lograron superar el nivel de cualquier ser humano; sin embargo, estos estaban desarrollados específicamente para el ámbito para el que se crearon y no pudieron ser generalizados sin un esfuerzo humano significativo hasta recientemente.

## 1.2 Proceso histórico

El proceso que se ha seguido desde el inicio del ajedrez de ordenador es la creación de árboles que guíen el curso del juego, puntuando de una manera u otra la posición del tablero, el movimiento a realizar, la situación del oponente... Para ello se han usado algoritmos de búsqueda de árboles que han ido progresando poco a poco hasta los actuales. Los más básicos son los algoritmos de búsqueda de anchura y profundidad que exploraban todos los movimientos, ambos con complejidad lineal que hacían que el problema fuese inabordable.

El primer y más común algoritmo que era aplicable al problema era el algoritmo Minimax, en el que el jugador busca el movimiento con el mayor valor dentro de los posibles reacciones de los rivales. Posteriormente, para agilizar la búsqueda, se emplea la poda alfa-beta, una técnica de búsqueda que reduce el número de nodos evaluados en un árbol por el algoritmo Minimax, reduciendo la complejidad de  $b^d$  a  $b^{d/2}$ , además todos estos algoritmos podían ser ayudados de heurísticas que permitiesen una búsqueda más eficaz en algunas situaciones.

Otro enfoque al Minimax es el dado por el algoritmo MCTS, que analiza los movimientos más prometedores, ampliando el árbol de búsqueda basado en un muestreo aleatorio del espacio de búsqueda que, mediante la repetición de muchas partidas, pondera los nodos en el árbol de juego para evaluar cómo de prometedor es un movimiento. Finalmente, AlphaZero incorporó el uso de MCTS para la creación de redes neuronales que sean capaces de interpretar el mismo procedimiento que haría el MCTS pero funcionando como memoria, siendo mucho más rápido a la hora de consultarla, y permitiendo el uso de redes neuronales como guía para elegir los movimientos más interesantes de cara a ganar la partida.

## 1.3 Precursores

### 1.3.1 Deep Blue

Un punto destacado en los hitos de la inteligencia artificial se sitúa en 1997, cuando la computadora conocida como Deep Blue pudo vencer a Garry Kasparov. Deep Blue es un supercomputador creado específicamente con la finalidad de ejecutar operaciones relacionadas con el ajedrez, permitiéndole valorar 100 millones de jugadas por segundo en su primera versión.

Su software fue escrito en lenguaje C. El sistema saca su fuerza de juego principalmente a través de la fuerza bruta de cálculo del sistema central, destacando por ser una computadora de procesamiento masivamente paralelo con 30 nodos, cada uno con 30 microprocesadores de 120 MHz, ampliados con 480 procesadores especializados para el ajedrez.

### 1.3.2 Stockfish

Otro hito de la inteligencia artificial sucede en 2008 con la aparición de Stockfish, el cual utiliza un algoritmo de búsqueda llamado poda alfa-beta, que mejora el algoritmo Minimax evitando variaciones que nunca serán alcanzadas en partidas profesionales porque cualquiera de los jugadores redireccionará la partida.

Ya que computacionalmente es inabordable la búsqueda hasta el final de la partida, se limita la profundidad de la búsqueda a un cierto nivel, esto es, se limita la distancia entre los nodos raíz y los nodos hoja que se exploran. Este límite se calcula en cada turno del jugador y el algoritmo lo puede

incrementar mediante un proceso llamado profundización iterativa. Esto, y el uso de heurísticas, permiten que el sistema pueda aumentar o disminuir este límite en función de lo prometedor que sea el movimiento.

Aunque a día de hoy Stockfish ha incorporado el uso de redes neuronales a la hora de evaluar el tablero, su arquitectura consiste esencialmente en tres partes; la representación del tablero, que busca una codificación del tablero de manera eficiente y que permita una búsqueda rápida; la búsqueda de árbol mediante heurísticas, y la evaluación del tablero, con el objetivo de describir cómo de "bueno" es el tablero al hacer un movimiento.

### 1.3.3 AlphaGo

Es importante mencionar AlphaGo, un algoritmo surgido en 2015 que, aunque jugase a Go, sirvió de inspiración para AlphaZero. Su principal innovación consiste en la representación del conocimiento en Go mediante el uso de redes neuronales convolucionales y el uso del algoritmo de búsqueda MCTS. Estas son entrenadas exclusivamente mediante aprendizaje por refuerzo a través de partidas jugadas por sí mismo.

El uso de redes neuronales y el algoritmo de búsqueda MCTS permitieron contrarrestar las llamadas "tácticas anticomputadoras", que abusaban de la poda alfa-beta y otros algoritmos minimax. Estas tácticas buscan un juego conservador con el objetivo de sacar ventaja a la larga para ganar mediante peones pasados, que no tienen oposición de peones contrarios que lo paren del avance hacia la octava fila.

Su principal diferencia con el algoritmo AlphaZero es su uso de las simetrías e invarianzas en la rotación que le permite hacer el juego de Go, la inexistencia de empates y su elección de partidas usadas en el aprendizaje, siendo AlphaZero una versión más general que tan solo necesita conocer las reglas básicas del juego en cuestión.

## 1.4 Ajedrez por computador: algo de historia

Los primeros avances en el ajedrez con computadores mediante el uso de redes neuronales se dieron en 1995, cuando *NeuroChess*, un programa que aprendía a jugar al ajedrez mediante los resultados finales de partidas, evaluó posiciones mediante una red neuronal que usaba 175 atributos de entrada elegidos manualmente. Fue entrenado mediante aprendizaje por diferencias temporales para predecir el resultado final de una partida.

Otro intento para evaluar las distintas piezas fue llevado a cabo mediante *Beal and Smith*, que también usó aprendizaje por diferencias temporales empezando por valores totalmente aleatorios y aprendiendo exclusivamente mediante partidas que jugaba solo.

*Knightcap* tomó un camino distinto en el intento de evaluar las distintas posiciones mediante una red neuronal que usaba un tablero de ataque basado en el conocimiento de qué casilla estaba siendo atacada o defendida y por qué piezas. Fue entrenada por una variante del aprendizaje por diferencias temporales llamado *TD(leaf)*, que actualizaba el valor de las hojas de la búsqueda alfabética.

Por último, *DeepChess* entrenó una red neuronal para evaluar por pares las distintas posiciones. Optó por tomar una base de datos de partidas de expertos (humanos) que fue previamente filtrada para evitar partidas con empates y movimientos de captura.

Sin embargo, todos estos programas combinaron las funciones de evaluación aprendidas junto con una *búsqueda alfa-beta* mejorada mediante una variedad de extensiones, de aquí la importancia de *AlphaZero* y *AlphaGo*.

## 1.5 Consecuencias

*AlphaZero* supuso un gran avance en el mundo del ajedrez, en el que cambió la forma de jugar demostrando que los avances con torre y peón eran muy poderosos, siendo elogiado por leyendas en el mundo del ajedrez como Garry Kasparov. Pero su influencia no solo se limita al mundo del ajedrez, **si no** que motivó grandes hitos en otros campos de la ciencia. La empresa creadora de *AlphaZero*, Deep Mind, logró en 2020 uno de los mayores adelantos en la historia de la biología descifrando el comportamiento de las proteínas y actualmente investiga el uso de sistemas inspirados en *AlphaZero* para la creación de sistemas inteligentes capaz de resolver problemas científicos complejos [15].

A día de hoy también ha servido de inspiración para el desarrollo de Leela Chess Zero y MuZero, este último, publicado en 2019 por Deep Mind, es un sistema único que juega a niveles excelentes en ajedrez, shogi, go y juegos de Atari, sin ser programado anteriormente con sus reglas.

También ha impulsado el uso de redes neuronales como apoyo a algoritmos de búsqueda, es el caso de Stockfish que tras su derrota contra *AlphaZero* en 2017 [12], empezó a usar redes neuronales para evaluar el tablero.

## 2 | Conceptos generales

En este capítulo vamos a definir conceptos básicos para el entendimiento de AlphaZero. Se dividirá en 3 secciones; *Algoritmos de búsqueda*, que tratará de definir los algoritmos usados para la búsqueda de elementos con propiedades deseadas por orden histórico, *redes neuronales*, que se encargará de explicar su funcionamiento y los distintos tipos de redes usadas en AlphaZero y, por último, *Aprendizaje por refuerzo*, que tratará de explicar cómo podemos usar los procesos de decisión de Markov a la hora de formalizar problemas como el que propone AlphaZero.

### 2.1 Algoritmos de Búsqueda

Un algoritmo de búsqueda es un conjunto de instrucciones diseñadas para encontrar un elemento con las propiedades deseadas en una estructura de datos; en nuestro caso, el mejor movimiento en una partida de ajedrez dentro de todos los movimientos posibles.

Existen multitud de algoritmos de búsqueda con diferentes estrategias que permiten una mayor eficiencia según la propiedad de los datos y restricciones de la aplicación. En el caso de la computación de ajedrez, un juego en el que podemos acceder constantemente a la información del resto de jugadores y no hay elementos aleatorios, es posible la construcción de un árbol con todos los resultados posibles.

Aunque es posible la construcción de tal árbol, es computacionalmente inabordable, por lo que se opta por métodos capaces de explorar las distintas opciones con distintos procesos.

#### 2.1.1 Minimax

**Minimax** [9] es un método de decisiones que busca minimizar la posible pérdida en el peor caso posible. Inicialmente fue formulada para el contexto de los denominados *Juegos de Suma Cero*, juegos para dos jugadores (se puede generalizar a más jugadores de diversas formas) en los que la recompensa de un jugador ganador es equivalente (en valor absoluto) a la pérdida por derrota del otro. Por ejemplo, si asociamos a una victoria una ganancia de +1, la derrota debe conllevar una pérdida de -1, dando como resultado total del juego siempre 0.

El ajedrez, al ser un juego de suma cero de 2 jugadores y con acceso completo a la información, es válido para su uso, pero, ¿cómo sabemos si esto conduce a una estrategia efectiva a la hora de jugar en estas condiciones? Esta pregunta no fue respondida hasta el 1928 con la llegada del **Teorema Minimax** por parte de *John Von Neumann*.

Según Von Neumann, un juego *es una situación conflictiva con la que uno debe tomar una decisión sabiendo que los demás también toman decisiones, y que el resultado del conflicto se determina, de algún modo, a partir de las decisiones realizadas*, y afirmó que *siempre existe una forma racional de actuar en juegos de 2 participantes, si los intereses que gobiernan son completamente opuestos*.

En general, el Teorema Minimax asegura que en las condiciones descritas, siempre existe una estrategia para cada jugador tal que la recompensa esperada por cada jugador es la misma, y, adicionalmente, esta recompensa es la mejor que cada jugador puede esperar conseguir en el caso de que su oponente juegue racionalmente. Formalmente, Von Neuman probó que:

$$v_1 = \max_i \min_j a_{ij} = \min_j \max_i a_{ij} = v_2$$

**Demostración.** Para probar el Teorema Minimax, usando [7] y [25] de guías, es necesario revisar un par de definiciones y teoremas antes de empezar la prueba:

Un problema de programación lineal enunciado en la siguiente forma se dice estar en forma primitiva

$$\text{Maximiza } z = c_1x_1 + \dots + c_nx_n$$

Sujeto a

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2$$

...

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m$$

$$x_1, x_2, \dots, x_n \geq 0$$

El problema dual del problema en forma primitiva es el siguiente problema de programación lineal:

$$\text{Minimiza } v = b_1y_1 + \dots + b_my_m$$

Sujeto a

$$a_{11}y_1 + a_{12}y_2 + \dots + a_{1m}y_m \geq c_1$$

$$a_{12}y_1 + a_{22}y_2 + \dots + a_{m2}y_m \geq c_2$$

...

$$a_{1n}y_1 + a_{2n}y_2 + \dots + a_{mn}y_m \geq c_n$$

$$y_1, y_2, \dots, y_m \geq 0$$

La notación por matrices puede ayudarnos a expresar estas definiciones. Sean  $A$  matriz,  $b, c$ ,  $x$  vectores columnas como los siguientes:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Por lo tanto, el problema primitivo es equivalente a maximizar  $z = cx$  sujeto a  $Ax \leq b, x \geq 0$ . Ahora, sea  $y$  el vector columna de dimensión  $m$ ,  $(y_1, y_2, \dots, y_m)$ , entonces el problema dual es minimizar  $v = by$  sujeto a  $A^T y \geq c, y \geq 0$ . En resumen, buscamos:

$$\text{Maximizar } z = cx \text{ sujeto a } Ax \leq b, x \geq 0$$

$$\text{Minimizar } v = by \text{ sujeto a } A^T y \geq c, y \geq 0$$

El teorema de la dualidad afirma que si el problema primitivo o el dual tiene una solución óptima finita, entonces el otro también la tiene, y, además, los valores óptimos de las funciones objetivo son iguales, esto es:

$$\max z = \min v$$

Procedamos ahora con la demostración particular de  $v_1 = v_2$ , con  $v_1 = \max_i \min_j a_{ij}$  y  $v_2 = \min_j \max_i a_{ij}$

Supongamos que el juego tiene una matriz  $m \times n$  de recompensas  $A = (a_{ij})$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ . Para poder usar técnicas de programación lineal en esta prueba necesitaremos asumir que  $v_1, v_2$  son positivas. Esto sucede claramente si todos los coeficientes de la matriz  $A$ ,  $a_{ij}$ , son positivos. Por lo tanto, consideraremos en esta prueba ambos casos, el primero en el que todo  $a_{ij} > 0$ , y el segundo, el caso general.

Caso 1. Todo  $a_{ij} > 0$ .

Consideremos el juego desde la perspectiva del jugador 1. Sabemos que  $v_1 = \max_{x \in X} \min_{1 \leq j \leq n} xA^j$ . Para determinar  $v_1$  para cada estrategia  $x \in X$  tenemos que encontrar el mínimo de  $xA^j$ . Es decir, debemos encontrar el mínimo en las siguientes  $n$  expresiones:

$$a_{11}x_1 + a_{21}x_2 + \dots + a_{m1}x_m$$

$$a_{12}x_1 + a_{22}x_2 + \dots + a_{m2}x_m$$

...

$$a_{1n}x_1 + a_{2n}x_2 + \dots + a_{mn}x_m$$

El mínimo de estas  $n$  expresiones es menor o igual que cualquiera de las  $n$  expresiones y es igual a al menos uno de ellas. Sea  $w$  el mínimo,  $w$  es el mayor número real satisfaciendo las siguientes  $n$  expresiones:

$$a_{11}x_1 + a_{21}x_2 + \dots + a_{m1}x_m \geq w$$

$$a_{12}x_1 + a_{22}x_2 + \dots + a_{m2}x_m \geq w$$

...

$$a_{1n}x_1 + a_{2n}x_2 + \dots + a_{mn}x_m \geq w$$

Por lo tanto, para cada  $x \in X$  el máximo  $w$  que cumple las desigualdades anteriores ha de ser calculado. Ahora, consideremos  $v_1$ , que es el máximo dentro de todos los  $x \in X$  de todas las  $w$ . Ya que

$$X = \{(x_1, x_2, \dots, x_m) | x_1 + x_2 + \dots + x_m = 1, x_i \geq 0, 1 \leq i \leq m\}$$

$v_1$  es el máximo  $w$  que satisface:

$$a_{11}x_1 + a_{21}x_2 + \dots + a_{m1}x_m \geq w$$

...

$$a_{1n}x_1 + a_{2n}x_2 + \dots + a_{mn}x_m \geq w$$

$$x_1 + \dots + x_m = 1$$

$$x_i \geq 0, 1 \leq i \leq m$$

Y por lo tanto, la estrategia  $x = (x_1, x_2, \dots, x_m)$  en la que el máximo es alcanzado es la mejor estrategia posible para el jugador 1. Ya que todo  $a_{ij} > 0$ , sabemos que  $v_1 > 0$ , por lo que podemos considerar que todo  $w > 0$ . Ahora, dividiendo las expresiones anteriores por  $w$  tenemos el problema siguiente:

Maximiza  $w$  sujeto a

$$a_{11} \frac{x_1}{w} + \dots + a_{m1} \frac{x_1}{w} \geq 1$$

...

$$a_{1n} \frac{x_1}{w} + \dots + a_{mn} \frac{x_m}{w} \geq 1$$

$$\frac{x_1}{w} + \dots + \frac{x_m}{w} = \frac{1}{w}$$

$$\frac{x_i}{w} \geq 0, 1 \leq i \leq m$$

Sea  $x'_i = \frac{x_i}{w}$ ,  $1 \leq i \leq m$ , y, ya que el problema de maximizar  $w$  es equivalente al de minimizar  $\frac{1}{w}$ , el problema puede ser reescrito a:

Minimiza  $x'_1 + x'_2 + \dots + x'_m$  sujeto a

$$a_{11}x'_1 + a_{21}x'_2 + \dots + a_{m1}x'_m \geq 1$$

...

$$a_{1n}x'_1 + a_{2n}x'_2 + \dots + a_{mn}x'_m \geq 1$$

$$x'_i \geq 0, 1 \leq i \leq m$$

Podemos reescribir esto de manera más concisa de la siguiente forma, sea  $x' = (x'_1, x'_2, \dots, x'_m)^T$ ,  $b = (1, 1, \dots, 1)^T$ , y  $c = (1, 1, \dots, 1)^T$ , donde  $b$  es una vector de dimensión  $m$  y  $c$  es un vector de dimensión  $n$ . De esta forma el problema puede ser reescrito a:

Minimiza  $bx'$  sujeto a

$$A^T x' \geq c, x' \geq 0$$

Resumiendo, la inversa del valor mínimo de la función  $bx'$  es igual a  $v_1$ . Además, ya que  $w(x'_1, \dots, x'_m) = (x_1, \dots, x_m)$ , multiplicar las coordenadas de un  $x'$  donde se alcanza un mínimo nos da la estrategia maximin para el jugador 1.

De una forma similar determinaremos  $v_2$  y la estrategia minimax del jugador 2 y veremos cómo nos lleva al problema dual de la anterior expresión.

Ya que  $v_2 = \min_{y \in Y} \max_{1 \leq i \leq m} A_i y^T$ , para un  $y \in Y$ , el máximo de las  $m$  expresiones  $A_i y^T$  es el menor número real  $z$  que cumple:

$$a_{11}y_1 + a_{12}y_2 + \dots + a_{1n}y_n \leq z$$

...

$$a_{m1}y_1 + a_{m2}y_2 + \dots + a_{mn}y_n \leq z$$

$$y_1 + y_2 + \dots + y_n = 1$$

$$y_j \geq 0, 1 \leq j \leq n$$

El punto en el que se alcanza el valor mínimo es la estrategia minimax para el jugador 2. Como antes, sabemos que  $v_2 > 0$  y, por lo tanto, cualquier  $z$  que satisfaga la expresión anterior ha de ser positiva. Dividamos las expresiones por  $z$  para llevar el problema al siguiente:

Minimiza  $z$  sujeto a

$$a_{11}\frac{y_1}{z} + \dots + a_{1n}\frac{y_1}{z} \leq 1$$

...

$$a_{m1}\frac{y_1}{z} + \dots + a_{mn}\frac{y_n}{z} \leq 1$$

$$\frac{y_1}{z} + \dots + \frac{y_n}{z} = \frac{1}{z}$$

$$\frac{y_j}{z} \geq 0, 1 \leq j \leq n$$

Sea  $y'_j = \frac{y_j}{z}, 1 \leq j \leq n$ . Entonces el problema anterior es equivalente a:

Maximiza  $y'_1 + y'_2 + \dots + y'_n$  sujeto a

$$a_{11}y'_1 + \dots + a_{1n}y'_n \geq 1$$

...

$$a_{m1}y'_1 + \dots + a_{mn}y'_n \geq 1$$

$$y'_j \geq 0, 1 \leq j \leq n$$

Y, como antes, llegamos a:

Maximiza  $c y'$  sujeto a

$$A y' \leq b, y' \geq 0$$

El inverso del valor máximo de la función  $cy'$  es igual a  $v_2$ . Además, ya que  $z(y'_1, \dots, y'_n) = (y_1, \dots, y_n)$ , multiplicando las coordenadas de  $y'$  en las que se alcanza el mínimo por  $v_2$ , llegamos a la estrategia minimax para el jugador 2.

Ahora podemos ver que maximizar  $cy'$  sujeto a  $Ay' \leq b, y' \geq 0$  es el problema primitivo y minimizar  $bx'$  sujeto a  $A^T x' \geq c, x' \geq 0$  es el problema dual. Además, el problema dual ha de tener una solución finita porque la función objetivo  $bx' = x'_1 + \dots + x'_m$  está acotada inferiormente por 0, y ya que todo  $a_{ij}$  es positivo, existen contadas soluciones para el sistema de restricciones  $A^T x' \geq c$ , y, por lo tanto, el problema primitivo también tiene una solución óptima finita. Teniendo en cuenta el Teorema de la Dualidad, ambos problemas tienen solución finita obteniendo el mismo valor óptimo, y por ello existen estrategias óptimas para el jugador 1 y 2, que son el maximin y el minimax, respectivamente, y estos valores son iguales,  $v_1 = v_2$ .

### Caso 2. Caso general

Supongamos que algunos  $a_{ij}$  no son positivos. Sea  $r$  cualquier constante con la propiedad de que  $a_{ij} + r > 0$  para cada  $i, j$ . Sea  $E$  la matriz  $m \times n$  con todos sus coeficientes iguales a 1 y consideremos el juego con la matriz de recompensa  $A + re$ .

Dadas las anteriores condiciones, la recompensa esperada para cada par de estrategias  $(x, y)$  es  $x(A + rE)y^T = xAy^T + rxEy^T$ , pero  $x = (x_1, \dots, x_m)$  y  $y = (y_1, \dots, y_n)$  son estrategias y  $\sum_{i=1}^m x_i = 1, \sum_{j=1}^n y_j = 1$ . De esta forma  $xE$  es igual al vector de dimensión  $n$   $(1, 1, \dots, 1)$  y  $xEy^T = (1, 1, \dots, 1)y^T = 1$ . Por lo tanto, la recompensa esperada es  $xAy^T + r$ , que es la recompensa esperada de la matriz de juego más la constante  $r$ .

Ya que las recompensas esperadas  $xAy^T$  y  $x(A + rE)y^T$  difieren tan solo por la constante  $r$ , es consecuencia que las matrices de recompensa de los juegos  $A$  y  $A + rE$  tendrán la misma estrategia óptima. De esta forma todos los coeficientes de la matriz  $A + rE$  son positivos, y, por lo tanto, podemos aplicar los resultados del caso 1 a este juego. Existiendo así las estrategias maximin y minimax para este juego con matriz de recompensa  $A$  y siendo  $v_1 = v_2$ .

Queda así demostrada la existencia de una estrategia óptima, pero las posibilidades en el ajedrez son del orden de  $10^{120}$ , mayor que el número de átomos en el universo observable, lo que hace que encontrar esta estrategia óptima sea computacionalmente inabordable, teniendo que hacer uso de heurísticas que identifiquen los movimientos más prometedores y cuya eficacia dependerá en gran parte de cuan efectivas sean las heurísticas o métodos usados para guiar el algoritmo.

Veamos ahora el algoritmo minimax que trata de usar estos conceptos para encontrar el mejor movimiento posible:

```

1 Function Minimax(nodo, profundidad, jugadorMaximizando):
2     If profundidad = 0 or nodo es terminal:
3         Return valor heurístico del nodo
4     If jugadorMaximizando:
5         valor ← -∞
6         ForEach hijo de nodo:
7             valor ← max(valor, Minimax(hijo, profundidad - 1, False))
8         Return valor
9     Else:
10        valor ← +∞
11        ForEach hijo de nodo:
12            valor ← min(valor, Minimax(hijo, profundidad - 1, True))
13        Return valor

```

El proceso de Minimax puede ser fácilmente explicado con una imagen que describa el procedimiento (Fig. 2.1).

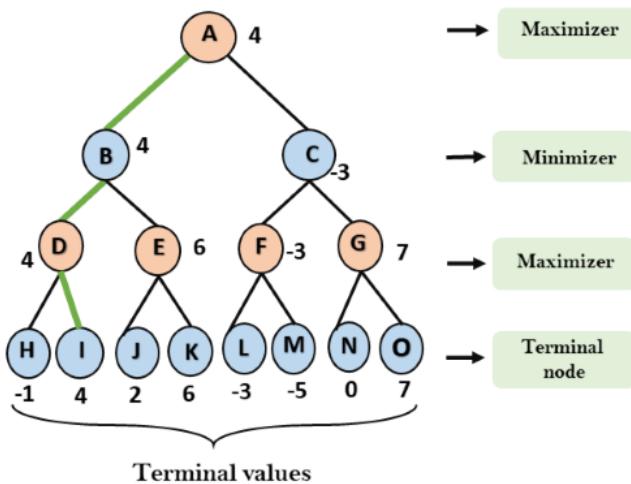


Figura 2.1: Representación gráfica del algoritmo Minimax aplicado a un árbol de estados.

En esta imagen tenemos dos jugadores, uno para el que queremos que su resultado sea maximice, y otro para el que queremos que su resultado sea el mínimo posible.

En el primer paso el algoritmo genera un árbol completo y utiliza una función que le permita calcular el valor de cada nodo terminal.

Una vez hecho esto, buscamos el nodo padre que cumpla la propiedad que deseemos, según el turno del último jugador en realizar una jugada. En nuestro caso, el siguiente turno es del jugador

que queremos que gane, por lo que para cada nodo padre buscamos el nodo hijo con mayor valor, en el caso de que el turno sea del jugador que queremos que pierda, buscamos el nodo hijo con menor valor.

Finalmente, tras repetir el proceso, llegamos al nodo raíz y termina el proceso.

### 2.1.1.1 Poda $\alpha - \beta$

Como se mencionó previamente, el número de estados que explora el algoritmo Minimax puede llegar a ser exponencial respecto al número de movimientos. Sea  $r$  el número de hijos de cada nodo y  $k$  el nivel de profundidad a alcanzar, la complejidad en tiempo es del orden  $\mathcal{O}(r^k)$  y la complejidad en espacio del orden  $\mathcal{O}(rm)$ .

Esto es computacionalmente inabordable, por lo que una forma de reducir el número de nodos a evaluar es mediante la técnica de poda  $\alpha - \beta$ , que busca eliminar partes del árbol que, a priori, sean irrelevantes para nuestro objetivo.

La idea general, como se explica en [29] consiste en determinar en cada momento un intervalo  $(\alpha, \beta)$  de posibles valores que podría tomar el nodo teniendo en cuenta su valor actual y el de su nodo padre.

El algoritmo consiste en lo siguiente:

```

1 Function AlphaBeta(nodo, profundidad, alpha, beta, jugadorMaximizando):
2     If profundidad = 0 or nodo es terminal:
3         Return valor heuristico del nodo
4     If jugadorMaximizando:
5         ForEach hijo de nodo:
6             alpha ← max(alpha, AlphaBeta(hijo, profundidad - 1, alpha, beta,
7                                         False))
8             If beta ≤ alpha:
9                 break
10            Return alpha
11        Else:
12            ForEach hijo de nodo:
13                beta ← min(beta, AlphaBeta(hijo, profundidad - 1, alpha, beta,
14                                         True))
15                If beta ≤ alpha:
16                    break
17            Return beta

```

Tratemos de exemplificarlo a partir de la imágenes. En el primer paso el jugador que maximiza empezará su primer movimiento desde el nodo A donde  $\alpha = -\infty$  y  $\beta = \infty$ . De aquí pasará al nodo

*B*, donde los valores de  $\alpha$  y  $\beta$  son los mismos, y pasará ambos valores al nodo *D*.

En el nodo *D*, el valor de  $\alpha$  se calcula con la función máx, al estar en el turno del jugador a maximizar. Por esto,  $\alpha$  será el valor del máximo entre los valores. Véase figura 2.2

Una vez calculado el valor del nodo *D*, calculamos el valor de  $\beta$  en el nodo *B*, que será el mínimo entre el valor actual de  $\beta$  en *B* y el valor mínimo de *D*, por lo que  $\alpha = -\infty$  y  $\beta = 3$  serán los valores de *B*.

En este paso el algoritmo avanza hacia el nodo *E*, por lo que los valores del nodo *B* pasarán al nodo *E*, en el cual es el turno del jugador a maximizar, por lo que el valor de  $\alpha$  cambiará de nuevo. El valor actual de *E* se compara con 5, así que el máximo entre 5 y  $-\infty$  será 5, y  $\alpha = 5$  y  $\beta = 3$ , donde  $\alpha \geq \beta$ , por lo que el sucesor a la derecha de *E* será podado y el algoritmo no lo recorrerá, fijando el valor del nodo en 5.

Aquí el algoritmo volverá al nodo *A* donde el valor de  $\alpha$  será el máximo entre  $-\infty$  y 3, y  $\beta$  será  $\infty$ , los cuales pasarán a su vez al nodo *C* y *F*. Descrito en la figura 2.3

En el nodo *F*,  $\alpha$  será comparado tanto con 0 como con 1, pero al ser superior a ambos, esté seguirá siendo 3 mientras el valor del nodo *F* pasará a ser 1. Después de esto, al retroceder al nodo *C*, el valor de  $\beta$  se actualiza al mínimo entre  $\infty$  y 1.

Finalmente, como en *C*  $\alpha = 3$  y  $\beta = 1$ ,  $\alpha \geq \beta$ , así que el siguiente nodo hijo de *C*, que es *G*, será podado. *C* devolverá el valor 1, y el mejor valor de *A* será el máximo entre 3 y 1, por lo que 3 será el valor óptimo. Representado en la figura 2.4

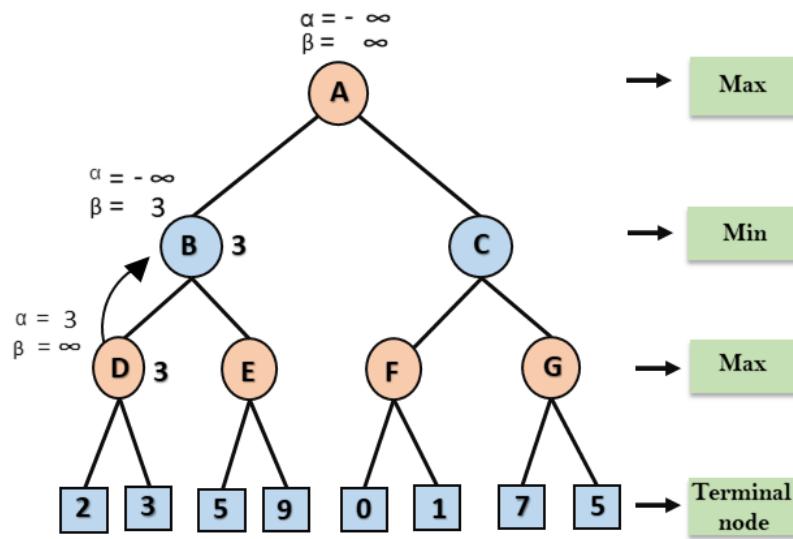


Figura 2.2: Representación gráfica del primer paso de la poda alfa-beta

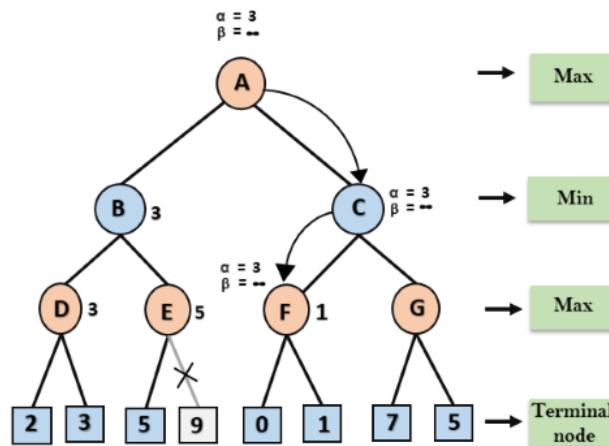


Figura 2.3: Representación gráfica del segundo paso de la poda alfa-beta

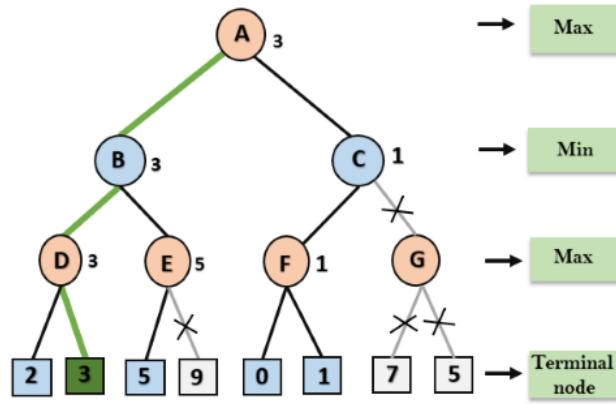


Figura 2.4: Representación gráfica del tercer paso de la poda alfa-beta

## 2.1.2 Búsqueda de Montecarlo

### 2.1.2.1 Problema de las Tragaperras Múltiples

Introduciremos en esta sección este problema descrito en el capítulo 2 de [40] y en [8] con el objetivo de definir la estrategia seguida por el método Montecarlo aplicado a búsquedas. Sustituiremos el papel desempeñado por los movimientos válidos y la recompensa recibida por las diferentes tragaperras y los premios que dan.

Sean  $K$  máquinas tragaperras, cada una con una tasa de retorno distinta, y sea nuestro objetivo buscar una estrategia que permita maximizar la ganancia total cuando las usamos. Supongamos además que no tenemos información de las máquinas, por lo que no podemos saber cuál es la mejor para jugar. La única forma es recopilar información personalmente, buscando un equilibrio para probar las máquinas y usar la que mejor resultados haya obtenido.

Este problema se puede definir de la siguiente manera:

$$\{X_{i,n} : 1 \leq i \leq K, n \geq 1\}$$

donde  $i$  es el índice de cada máquina. Las jugadas sucesivas de cada máquina dan lugar a  $X_{i,1}, X_{i,2}, \dots$ , independientes e idénticamente distribuidas siguiendo una distribución desconocida de media  $\mu_i$ . Además, también exigimos que  $\forall 1 \leq i < j \leq K$  y  $s, t \geq 1$ , se tiene que  $X_{i,s}$  y  $X_{j,t}$  son independientes.

Dada estas condiciones, una estrategia es un algoritmo  $A$  que elige la siguiente máquina en la que jugar basándose en la secuencia de las jugadas anteriores y los resultados obtenidos.

Si  $T_i(n)$  es el número de veces que  $A$  ha seleccionado la máquina  $i$  durante las primera  $n$  jugadas, entonces la pérdida esperada de la estrategia  $A$ , lo que hemos perdido por no haber escogido la mejor máquina, se define como:

$$\mu^*n - \sum_{i=1}^K \mu_i E[T_i(n)]$$

donde  $\mu^* = \max_{1 \leq i \leq K} \mu_i$  representa la mejor jugada posible.

Según algunos resultados de estadística, podemos apoyarnos en la siguiente estrategia, llamada **UCB1 (Upper Confidence Bound)**, que permite construir intervalos estadísticos de confianza para cada máquina de la forma:

$$\left[ \bar{X}_i(n) - \sqrt{\frac{2 \ln(n)}{T_i(n)}}, \bar{X}_i(n) + \sqrt{\frac{2 \ln(n)}{T_i(n)}} \right]$$

donde,  $\bar{X}_i(n)$  es la media de ganancias experimental que ofrece la máquina  $i$  tras los  $n$  pasos.

$$\bar{X}_i(n) = \frac{\sum_{j \leq n} X_{i,j}}{T_i(n)}$$

La ley de los grandes números asegura que  $\lim_{T_i(n) \rightarrow \infty} \bar{X}_i(n) = \mu_i$ . Por tanto, una estrategia válida sería escoger siempre la máquina con mayor límite superior de este intervalo. Cuanto más se use la máquina, el valor medio observado se desplazará al valor real y su intervalo de confianza se reducirá. Llegará un momento en el que alguna de las otras máquinas podrá tener un límite superior mayor al de la máquina actualmente seleccionada, y entonces lo mejor será cambiar a esta.

Esta estrategia verifica que la diferencia entre lo que hubiéramos ganado jugando únicamente en la mejor máquina y las ganancias esperada bajo la estrategia usada crece sólo como  $\mathcal{O}(\log(n))$  (que pierde poco respecto a la mejor jugada posible), que es la misma tasa de crecimiento que la mejor estrategia teórica para este problema, y que tiene el beneficio de ser fácil de calcular, teniendo que llevar solo un contador para cada máquina.

### 2.1.2.2 Algoritmo MCTS

Tras la descripción del problema anterior definimos el MCTS [8], donde el papel que juegan las máquinas tragaperras se cambia por las jugadas válidas, y los premios, por la recompensa de la sucesión de movimientos realizados.

El concepto más importante del Método de búsqueda de Montecarlo es una búsqueda, descrita en la sección 2.3.1 de [32], que es un conjunto de recorridos del árbol. Un recorrido se define como el camino entre el nodo raíz a un nodo que aún no ha sido expandido completamente, o sea, que tiene algún nodo hijo sin haber sido explorado.

Una vez un nodo no completamente explorado es encontrado, uno de sus hijos no explorados es elegido como nodo raíz para una simulación, el resultado de esta es posteriormente retropropagado hasta el actual nodo raíz del árbol, actualizando las estadísticas de los nodos atravesados.

Una vez finalizada la búsqueda, limitada por tiempo o motivos computacionales, el movimiento es elegido basado en las estadísticas recogidas. Definamos con más profundidad los procesos:

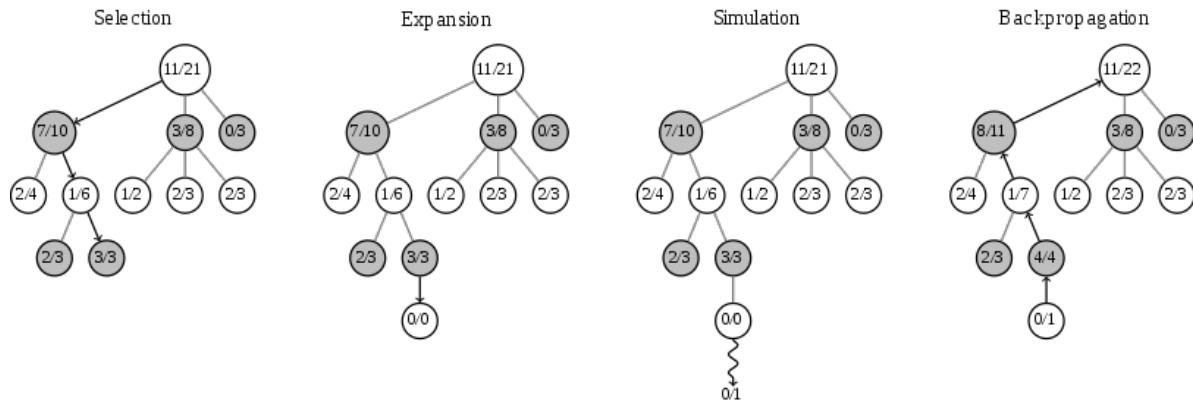


Figura 2.5: Representación gráfica de los distintos procesos del MCTS

El primero que definiremos es la simulación, que es el acto de "jugar una partida", una serie de movimientos que empieza en el nodo actual y termina en un nodo terminal donde el resultado de la partida puede ser valorado. Una simulación no es más que una aproximación de la evaluación del nodo del árbol de juego, calculada mediante una partida hasta cierto punto aleatoria jugada a partir del nodo elegido.

Una duda que puede surgir sería cómo son elegidos estos movimientos durante la simulación, y la respuesta es mediante una función llamada función de política de despliegue:  $s_i \rightarrow a_i$  que devuelve la siguiente acción al recibir un estado de juego. En la práctica está diseñado para que sea fácil de calcular para permitir una rápida simulación. Posteriormente veremos qué particularidad presenta AlphaZero en las simulaciones.

Cuando consideramos un movimiento imaginamos la situación que plantea como consecuencia, por eso, cuando visitamos un nodo, estamos evaluando la situación en la que nos deja tras realizar un movimiento concreto. Los nodos sin visitar son aquellos en los que ni tan siquiera hemos considerado sus movimientos, por lo que su potencial puede quedarse sin descubrir.

Esto nos sirve para definir los términos usados anteriormente, un nodo se considera visitado si al menos una partida ha sido jugada a partir de la situación que describe, y se dice que está completamente explorado si todo nodo hijo es visitado.

En la práctica, en el inicio de cada búsqueda comenzamos con todos los nodos hijos no visitados, uno de ellos es seleccionado y la primera simulación es llevada a cabo.

La retropropagación ya ha sido definida previamente, pero, en resumen, una vez una simulación es finalizada para un nodo recién visitado, su resultado está listo para ser retropropagado hasta el nodo raíz y el nodo donde comenzó la simulación pasa a ser visitado.

La retropropagación es un recorrido que va desde el nodo hoja hasta el nodo raíz en el que el resultado obtenido va actualizando los nodos por los que pasa garantizando que las estadísticas de cada nodo reflejan resultados de las simulaciones llevadas a cabo a partir de nodos descendientes de ellos.

El objetivo de la retropropagación es actualizar el resultado de las recompensas totales simuladas  $Q(v)$  y el número total de visitas  $N(v)$  para cada nodo  $v$  en el camino a retropropagar.  $Q(v)$  es un atributo del nodo  $v$  que define, en su forma más simple, la suma de los resultados obtenidos en todas las simulaciones que pasaban por ese nodo,  $N(v)$  es simplemente un contador de cuantas veces ha pasado por ese nodo un camino al retropropagar.

Estos dos atributos son almacenados para cada nodo visitado, una vez un número de simulaciones suficientes es alcanzado, los nodos visitados almacenarán información indicando cómo de explorado o explotado está. Dicho de otra forma esto último, al mirar las estadísticas de un nodo aleatorio, estos dos valores reflejarán cómo de prometedor es mediante  $Q(v)$ , y cómo de explorado está con  $N(v)$ .

A los nodos con un valor alto en  $Q(v)$  los consideraremos buenos candidatos al estar "explotados", y los nodos con un valor bajo  $N(v)$  serán interesantes a la hora de considerarlos al no haber sido explorados suficientemente.

De esta forma ya solo nos quedaría por explicar de qué forma navegamos desde un nodo totalmente explorado a uno sin visitar.

Empezando desde el inicio de una simulación, todos los nodos sin visitar son elegidos primero, una simulación empieza en cada uno de ellos y, tras eso, los resultados son retropropagados. Una vez finaliza este proceso el nodo raíz pasa a ser considerado totalmente visitado.

Para elegir el siguiente nodo para la siguiente simulación a partir del nodo completamente explorado  $v$  necesitamos información de todos sus nodos hijo y del nodo  $v$  mismo. En esta situación disponemos de la información almacenada del nodo actual y de sus hijos, al haber sido todos explorados, esta información almacenada nos permite hacer uso del UCB mencionado en el problema de las tragaperras. Sea

$$UCB(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c \sqrt{\frac{\log(N(v))}{N(v_i)}}$$

esta función viene definida por un nodo hijo  $v_i$  de un nodo  $v$ . Es la suma de dos componentes, el primero, llamado componente de explotación, que no es nada más que la probabilidad de victoria del nodo hijo, y la segunda, llamado componente de exploración, que nos permite evitar la simulación repetida de aquellos nodos con tan solo 1 victoria y 1 partida simulada o casos parecidos. Además llamamos a  $c$  al factor de exploración, con el que podemos decidir cuán importante será la exploración al decidir los siguientes nodos.

Esto es así en general, pero posteriormente veremos qué hace AlphaZero que lo diferencia de los demás.

Finalmente, una vez el Método de búsqueda de Montecarlo ha finalizado, el mejor movimiento viene dado usualmente por el nodo con mayor número de visitas, con mayor  $N(v_i)$ , pues este nodo ha sido estimado como el mejor (su  $Q(v_i)$  ha de ser un valor alto al haber sido explorado tantas veces). En resumen, una versión simplificada de la definición de las funciones necesarias para el MCTS en la que el proceso de selección de hijos es aleatorio puede ser consultada en [28], y sería la siguiente:

---

**Algorithm 1:** Búsqueda de Montecarlo

---

```

1. function BusquedaMontecarlo(raiz)
2.   while quedenRecursos(tiempo,poderComputacional) do
3.     hoja ← atravesar(raiz)
4.     resultadoSimulacion ← despliegue(hoja)
5.     retropropagacion(hoja, resultadoSimulacion)
6.   end while
7.   return mejorHijo(raiz)
8. end function
9.
10. function atravesar(nodo)
11.   while completamenteExplorado(nodo) do
12.     nodo ← mejorUCB(nodo)
13.   end while
14.   return seleccionaNoVisitado(nodo.hijos) or nodo      # en caso de que sea
   terminal/no tenga hijos
15. end function
16.
17. function despliegue(nodo)
18.   while noTerminal(nodo) do
19.     nodo ← desplieguePolitica(nodo)
20.   end while
21.   return resultado(nodo)
22. end function
23.
24. function desplieguePolitica(nodo)
25.   return seleccionaAleatorio(nodo.hijos)
26. end function
27.
28. function retropropagacion(nodo, resultado)
29.   if esRaiz(nodo) then return
30.   end if
31.   nodo.estadisticas ← actualizarEstadisticas(nodo, resultado)
32.   retropropagacion(nodo.padre)
33. end function
34.
35. function mejorHijo(nodo)
36.   return escoge hijo con mayor número de visitas
37. end function

```

---

## 2.2 Redes Neuronales

Las **Redes Neuronales Artificiales** [10] son parte del aprendizaje automático, su nombre y estructura se inspiran en el cerebro e imitan el funcionamiento entre las neuronas que lo componen.

Las redes neuronales están formadas por capas de nodos, una capa de entrada, una de salida, y una o varias capas ocultas. Cada nodo se conecta con otro y tiene un peso y umbral asociado, activándose en caso de que la salida de este esté por encima del valor umbral, enviando así datos a la siguiente capa de red.

Estas se basan en entrenar datos para aprender y mejorar su precisión, además, tras un tiempo de aprendizaje, pueden convertirse en potentes herramientas capaces de clasificar y agrupar datos a gran velocidad, mejorando incluso la velocidad del reconocimiento de voz o imágenes efectuado por expertos humanos. Todos los conceptos definidos posteriormente están recogidos en [6].

### 2.2.1 Nodos y capas

Un **nodo** o **perceptrón** es una unidad computacional que tiene una o más entradas (o *inputs*), una función que combina esos inputs de cierta forma, y una salida (o *output*). La unión organizada de estos nodos se llama capa, y la unión organizada de estas se denomina red neuronal.

La forma en la que un nodo combina estos inputs para decidir su output es mediante un conjunto de coeficientes, o pesos, que pueden amplificar o reducir esos inputs, cuantificándolos según el objetivo. Esto podría ser, en el caso de querer identificar si es de noche en una imagen, un nodo de la penúltima capa que se encarga de ver si en el modelo CMYK la mayoría de los píxeles tienen un componente *K* alto, refiriéndose al color negro, mandaría una señal al nodo de la capa final y este tendría un coeficiente alto, ya que para dictaminar si una imagen es de noche este es un dato importante.

Generalmente al inicio los coeficientes son aleatorios, por lo que las respuestas de la red son erróneas, es por eso que la red aprende a través del entrenamiento. Continuamente se le muestra a la red ejemplos en los que el resultado es conocido, y, mediante la comparación de los resultados y las respuesta dadas, es capaz de ir ajustando los coeficientes. A medida que va comparando los resultados con las respuestas, estas terminan siendo más fiables, permitiendo así su posterior uso a la hora de predecir casos en los que no conocemos de antemano el resultado.

#### 2.2.1.1 Función de activación

Una pregunta que podría surgir sería *¿cuál es el baremo que se toma a la hora de decidir si el nodo se activa o no?* Esto depende de la red elegida, y se denomina función de activación. Supongamos

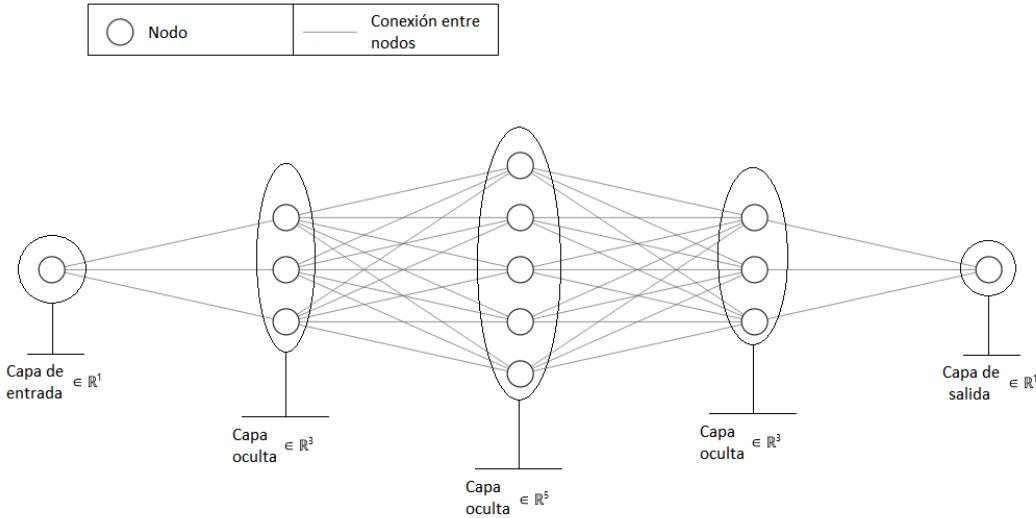


Figura 2.6: Representación gráfica de capas, nodos y conexiones en una red neuronal generada con NN-SVG

que un nodo de nuestra red neuronal tiene tres inputs  $x_i$ , con sus tres respectivos coeficientes  $w_i$ , y un output  $z = \sum_{i=1}^3 (x_i * w_i) + b \in \mathbb{R}$ , llamamos  $f(z)$  función de activación a aquella función limitadora que modifica el valor resultado o impone un límite que se debe superar para pasar al próximo nodo.

**Algoritmo 2.1 (Descenso del gradiente).** Llamaremos gradiente de una función de activación a una generalización de la derivada de la función. Esto es:

$$\nabla f(p) = \begin{vmatrix} \frac{\partial f}{\partial x_1}(p) \\ \frac{\partial f}{\partial x_2}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{vmatrix} \quad (2.1)$$

Se encarga de encontrar el mínimo local de una función y los requisitos para su uso son que la función ha de ser derivable y debe ser convexa. El algoritmo consiste en lo siguiente:

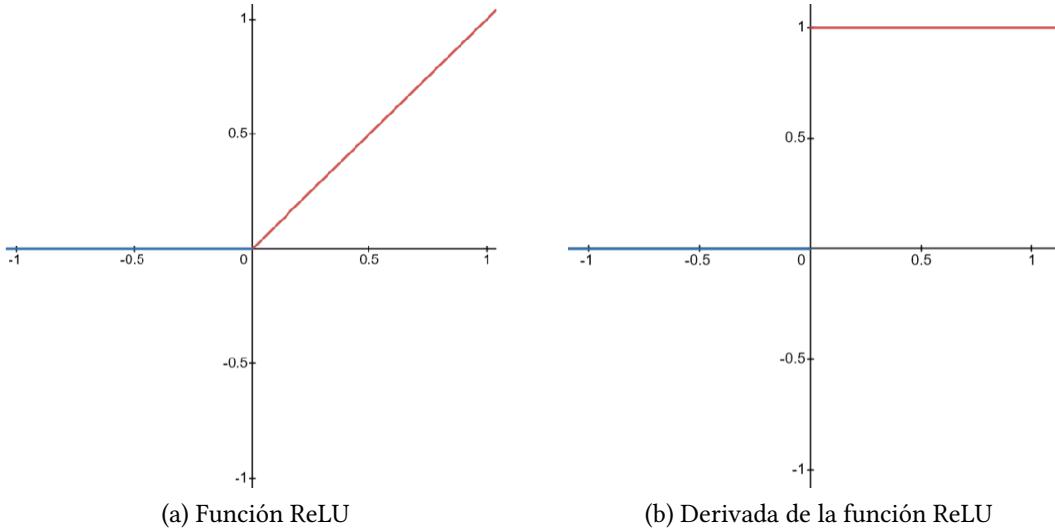
1. Se escoge un punto de salida aleatoriamente,
2. Se calcula el gradiente en ese punto,
3. Se calcula el siguiente punto de la siguiente forma:  $p_{n+1} = p_n - \gamma \nabla f(p_n)$ ,
4. Se detiene el algoritmo con alguna condición; número máximo de iteraciones o tamaño del paso dado menor que umbral.

Las funciones de activación más frecuentes recogidas en [4] son:

**ReLU o unidad lineal rectificada,**

$$f(z) = \begin{cases} z & \text{si } z \geq 0 \\ 0 & \text{resto} \end{cases}$$

ya que solo un número limitado de neuronas son activadas, la función ReLU es computacionalmente muy eficiente, además acelera la convergencia del método de descenso de gradiente hacia el mínimo global debido a su carácter lineal. Una de sus limitaciones es el problema generado al  $f(z)$  ser igual a 0 para los valores negativos, debido a esto, durante el proceso de retropropagación, los coeficientes de algunas neuronas no son actualizadas, lo que puede ocasionar nodos que nunca se activan.



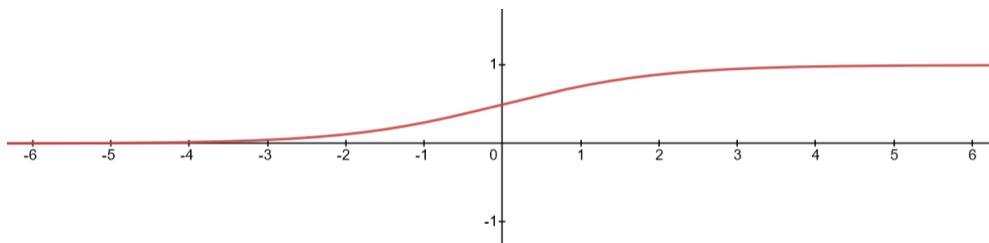
(a) Función ReLU

(b) Derivada de la función ReLU

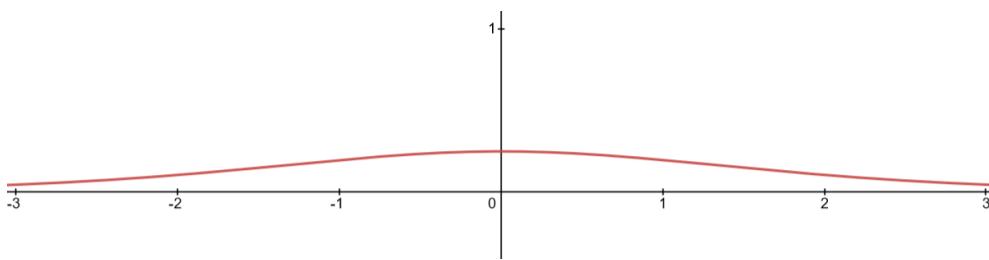
**Sigmoide,**

$$f(z) = \frac{1}{1 + e^{-z}}$$

es comúnmente usada para modelos donde es necesario predecir la probabilidad como output, por ello es muy conveniente que su rango sea entre 0 y 1. Además, es diferenciable y su gradiente no experimenta cambios bruscos, evitando saltos en el valor del output. La derivada de la función sigmoide es  $f'(z) = f(z) * (1 - f(z))$ , y los valores mayores que 3 y menores que -3 tienen un gradiente muy bajo, por lo que la red para de aprender y sufre del "Problema de desvanecimiento de gradiente".



(a) Función sigmoide

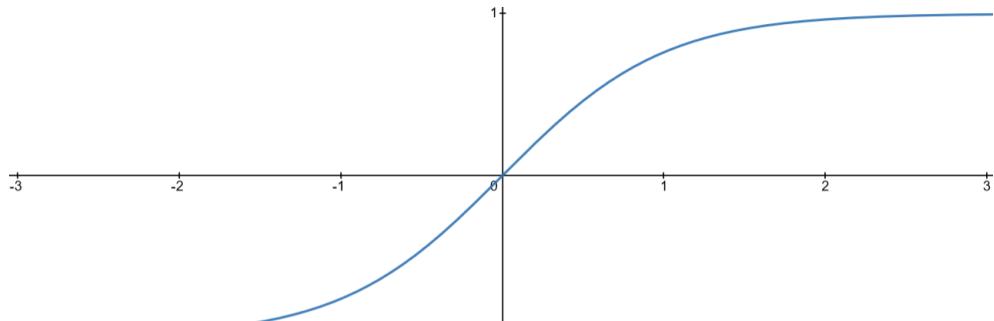


(b) Derivada de la función sigmoide

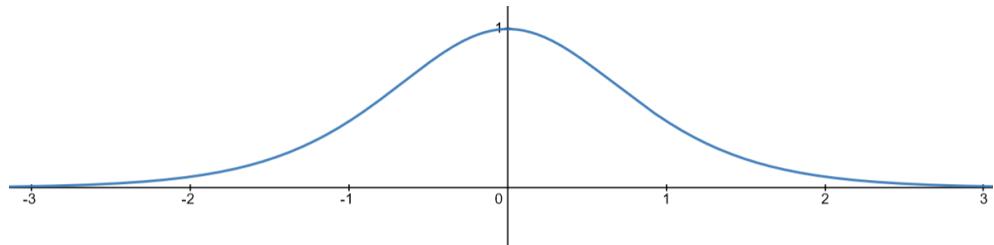
### Tangente hiperbólica,

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

es muy parecida a la función sigmoide con la diferencia de que su rango está entre -1 y 1, además, es muy útil ya que la función está centrada en el 0, permitiendo identificar los valores como muy negativos, neutros o muy positivos. Su uso se da frecuentemente en las capas ocultas, como su rango está entre -1 y 1, la media suele ser cercana al 0, ayudando al centrado de los datos y habilitando un aprendizaje en la próxima capa más eficaz. La derivada de la función es  $f'(z) = 1 - \tanh^2(z)$ , por lo que también sufre del **Problema de desvanecimiento de gradiente**.



(c) Función tangente hiperbólica



(d) Derivada de la función tangente hiperbólica

**Softmax,**

$$f(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \text{ para } j = 1, \dots, K$$

se usa para *comprimir* un vector  $z$  de dimensión  $K$  de valores reales, en un vector  $f(z)$  de dimensión  $K$  de valores reales en el rango  $[0, 1]$ . Suele ser usada en redes neuronales para normalizar el output a una distribución de probabilidad. Debido a su alto coste computacional, ya que  $K$  suele ser un número grande, suele ser usado exclusivamente en la última capa.

Su primera derivada es la siguiente:

$$\frac{\partial f(z_k)}{\partial z_j} = \begin{cases} f(z_k)(1 - f(z_j)) & \text{si } k = j \\ -f(z_j)f(z_k) & \text{si } k \neq j \end{cases}$$

### 2.2.1.2 Propagación hacia atrás

El modo en el que los coeficientes son actualizados para mejorar la red neuronal es mediante el algoritmo de propagación hacia atrás de errores, o retropropagación [45]. La propagación hacia atrás es un algoritmo de aprendizaje supervisado que se usa para entrenar redes neuronales previamente alimentadas.

Para entender cómo funciona el algoritmo de retropropagación o backpropagation haremos uso del ejercicio descrito en [19], consideremos una red con 3 capas, una de entrada, una oculta y una de salida, usando la función sigmoide como función de activación en la capa oculta y la función softmax en la capa de salida y definamos  $x_i$  como la respuesta del nodo  $i$ -ésimo de la capa de entrada,  $y_i$  la respuesta del nodo  $i$ -ésimo de la capa oculta y  $z_i$  la respuesta del nodo  $i$ -ésimo de la capa de salida.

Definamos una función de pérdida que nos permita encontrar el conjunto de coeficientes que minimice el error entre la salida de nuestra red y el resultado esperado. Sea esta función

$$J(w) = - \sum_{k=1}^K t_k \log(z_k)$$

donde  $t = [t_1, \dots, t_K]^T$  es la salida deseada en formato *one hot encoding*, que es tan solo una forma de asignar valores numéricos a diferentes categorías,  $z = [z_1, \dots, z_K]^T$  el vector de respuestas en la capa de salida, y  $w$  representando todos los coeficientes de la red neuronal.

La propagación hacia atrás se basa en descenso de gradiente, para ello, los coeficientes son inicializados aleatoriamente y cambian en la dirección que reduce el error mediante lo que denominamos la regla delta:

$$\Delta w = -\mu \frac{\partial J}{\partial W}$$

donde  $\mu$  es la tasa de aprendizaje. Por lo tanto, la regla de actualización de la  $t$ -ésima iteración sería:

$$w(t+1) = w(t) + \Delta w(t)$$

Ahora veamos cómo podemos actualizar los coeficientes de la capa oculta hacia la de salida. Para ello es necesario derivar la función de pérdida  $J(w)$  con respecto a los coeficientes de la capa oculta hacia la de salida, haremos uso de la regla de la cadena:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial u_{kj}} \frac{\partial u_{kj}}{\partial w_{kj}}$$

donde  $u_{kj} = y_j w_{kj}$ , es decir, la respuesta del  $j$ -ésimo nodo oculto multiplicada por el coeficiente  $w_{kj}$  que lo conecta con el  $k$ -ésimo nodo de salida. Esto es lo mismo a:

$$\frac{\partial J}{\partial w_{jk}} = \underbrace{\frac{\partial J}{\partial u_{kj}}}_A \underbrace{\frac{\partial u_{kj}}{\partial w_{kj}}}_B$$

Teniendo en cuenta la expresión de la función de pérdida tenemos:

$$\frac{\partial J}{\partial u_{k_j}} = \frac{\partial}{\partial u_{k_j}} \left( - \sum_{k=1}^K t_k \log z_k \right) = - \sum_{k=1}^K t_k \frac{\partial}{\partial u_{k_j}} (\log z_k) = - \sum_{k=1}^K \underbrace{\frac{t_k}{z_k} \frac{\partial z_k}{\partial u_{k_j}}}_{a}$$

La derivada de la función softmax que fue calculada anteriormente coincide con  $a$  por lo que:

$$\begin{aligned} \frac{\partial J}{\partial u_{k_j}} &= - \left[ \frac{t_k}{z_k} \frac{\partial z_k}{\partial u_{k_j}} + \sum_{k=1, k \neq j}^K \frac{t_j}{z_j} \frac{\partial z_j}{\partial u_{k_j}} \right]. \\ \frac{\partial J}{\partial u_{k_j}} &= - \left[ \frac{t_k}{z_k} z_k (1 - z_k) + \sum_{k=1, k \neq j}^K \frac{t_j}{z_j} (-z_k z_j) \right] = -t_k (1 - z_k) + \sum_{k=1, k \neq j}^K t_j z_k = \\ &\quad \underbrace{z_k \left( t_k + \sum_{k=1, k \neq j}^K t_j \right)}_b - t_k \end{aligned}$$

Nótese que  $b$  es equivalente a  $\sum_{k=1}^K t_k = 1$  ya que  $t_k$  es la salida deseada en formato one hot encoding. Por tanto,

$$\boxed{\frac{\partial J}{\partial u_{k_j}} = z_k - t_k}$$

Además, como  $u_{k_j} = y_j w_{k_j}$ , en  $B$  podemos llegar a:

$$\frac{\partial u_{k_j}}{\partial w_{k_j}} = \frac{\partial}{\partial w_{k_j}} (y_j w_{j_k}) = y_j \frac{\partial w_{j_k}}{\partial w_{k_j}} = \boxed{y_j \cdot}$$

Por lo que tendríamos:

$$\frac{\partial J}{\partial w_{k_j}} = \frac{\partial J}{\partial u_{k_j}} \frac{\partial u_{k_j}}{\partial w_{j_k}} = (z_k - t_k) y_j.$$

Obteniendo así la regla delta para actualizar los coeficientes de la capa oculta hacia la capa de salida:

$$\Delta w_{k_j} = -\mu \underbrace{(z_k - t_k)}_{\delta_k} y_j$$

Finalmente, para actualizar los coeficientes de la capa de entrada hacia la capa oculta sería necesario derivar la función de pérdida  $J(w)$  con respecto a los coeficientes de la capa de entrada hacia la oculta:

$$\frac{\partial J}{\partial w_{j_i}} = \underbrace{\frac{\partial J}{\partial y_j}}_A \underbrace{\frac{\partial y_j}{\partial u_{j_i}}}_B \underbrace{\frac{\partial u_{j_i}}{\partial w_{j_i}}}_C,$$

con  $u_{j_i} = x_i w_{j_i}$ , la  $i$ -ésima variable de la entrada  $x = [x_1, \dots, x_{K'}]^T$  multiplicada por el coeficiente  $w_{j_i}$  que lo conecta con el  $j$ -ésimo nodo oculto. Tenemos la siguiente expresión por la regla de la cadena:

$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{K'} \frac{\partial J}{\partial u_{k_j}} \frac{\partial u_{k_j}}{\partial y_j},$$

la respuesta del  $j$ -ésimo nodo oculto se distribuye a los  $K'$  nodos de salida, donde  $\frac{\partial J}{\partial u_{k_j}} = \delta_k$  y, por tanto:

$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{K'} \delta_k \frac{\partial u_{k_j}}{\partial y_j}.$$

Ya que  $u_{k_j} = y_j w_{k_j}$  tenemos:

$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{K'} \delta_k \frac{\partial}{\partial y_j} (y_j w_{k_j}) = \sum_{k=1}^{K'} \delta_k w_{k_j} \frac{\partial y_j}{\partial y_j} = \sum_{k=1}^{K'} \delta_k w_{k_j}.$$

En el término  $B$ ,  $y_j$  representa la respuesta del  $j$ -ésimo nodo oculto activado por la función sigmoide, a partir de ahora nos referiremos a esta con  $\sigma(x)$ . Por lo que tenemos  $y_j = \sigma(u_{j_i})$ , y sustituyendo en B llegamos a  $\frac{\partial y_j}{\partial u_{j_i}} = \frac{\partial}{\partial u_{j_i}} \sigma(u_{j_i})$ , que al sustituir con la derivada de la función sigmoide queda como:

$$\frac{\partial y_j}{\partial u_{j_i}} = \sigma(u_{j_i})(1 - \sigma(u_{j_i})).$$

Teniendo en cuenta  $C$  y como  $u_{j_i} = x_i w_{j_i}$ , tenemos:

$$\frac{\partial u_{j_i}}{\partial w_{j_i}} = \frac{\partial}{\partial w_{j_i}}(x_i w_{j_i}) = x_i \frac{\partial w_{j_i}}{\partial w_{j_i}} = \boxed{x_i}$$

Por último, sustituyendo  $A, B, C$  tenemos:

$$\frac{\partial J}{\partial w_{j_i}} = (\sum_{k=1}^{K'} \delta_k w_{k_j})(\sigma(u_{j_i}))(1 - \sigma(u_{j_i}))x_i.$$

Obteniendo así la regla delta para actualizar los coeficientes de la capa entrada hacia la oculta:

$$\Delta w_{j_i} = -\mu \underbrace{\left( \sum_{k=1}^{K'} \delta_k w_{k_j} (\sigma(u_{j_i}))(1 - \sigma(u_{j_i})) \right)}_{\delta_j} x_i.$$

A grandes rasgos el algoritmo consiste en lo siguiente:

*Algoritmo 2.2 (Algoritmo de retropropagación).* Llamaremos  $o^i$  a la  $i$ -ésima capa de la red neuronal y  $W^i$  a la matriz cuyos valores son los coeficientes de la conexión entre  $o_j^{i-1}$  y  $o_{j'}^i$ , para  $i = 1, \dots, N$ .

1. Cálculo de la salida de la red neuronal a partir de uno de los conjuntos de valores de prueba,
  2. Comparación de la salida esperada con el valor calculado y cálculo del error,
  3. Cálculo de las derivadas parciales del error con respecto a los coeficientes  $W^i$  para  $i = 1, \dots, N - 1$  para una red con  $n$  capas,
  4. Ajuste de los coeficientes de cada nodo con el objetivo de disminuir el error,
  5. Repetir el proceso varias veces por cada par entrada-salida de prueba
- 

## 2.2.2 Red Neuronal Convolutacional

Una **Red Neuronal Convolutacional** (CNN, [27]) es uno de los tipos de redes neuronales más usados a día de hoy, en ella las neuronas artificiales corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico. Debido a que su aplicación es realizada en matrices bidimensionales, son muy efectivas para tareas como la clasificación y segmentación de imágenes.

Las redes neuronales convolucionales tienen 3 tipos de capas, la capa convolucional, la capa de agrupación y la capa completamente conectada. La capa convolucional es la primera capa de una

CNN y puede ser seguida por capas convolucionales adicionales o capas de agrupación, mientras que la capa completamente conectada es la capa final.

Cuantas más capas, más crece en complejidad la CNN, siendo capaz de identificar características más específicas de una imagen, las primeras capas se centran en rasgos más generales, como colores o bordes, conforme las capas avanzan, empiezan a reconocer elementos más grandes o formas hasta ser capaces de identificar el objeto.

La capa convolucional es la pieza fundamental de una CNN y es donde la mayoría de la fuerza computacional se emplea. Requiere varios elementos, datos de entrada, filtros, y una máscara de atributos. Por ejemplo, en el caso de una imagen a color tendremos unos datos de entrada en 3 dimensiones, correspondiente al valor del píxel en la escala R, G y B. Además, tenemos un detector de atributos, conocido como kernel o filtro, que se moverá a través de la imagen reconociendo si dicho atributo está presente, a este proceso le llamamos convolución. Más información sobre el proceso de convolución puede ser consultado en [42]

El detector de atributos es 2-dimensional, un vector de coeficientes que representa parte de la imagen, y, aunque puede variar en tamaño, normalmente es una matriz 3x3. Este filtro es aplicado a parte de la imagen y mediante un producto entre los píxeles de entrada y el filtro se llega al vector output.

Después de cada proceso de convolución, la CNN aplica una transformación ReLU a la máscara de atributos, introduciendo no linealidad al modelo. Como se mencionó anteriormente, una capa convolucional extra puede seguir a la inicial, cuando esto pasa, la estructura de la CNN puede jerarquizarse debido a que recibe los datos ya procesados por las anteriores capas.

Todo esto podría verse como, suponiendo el caso de una imagen de una bicicleta, una capa inicial fuese capaz de distinguir los bordes, y una segunda capa, tras recibir esos bordes, sea capaz de distinguir la forma de una rueda, manillar,...

Las capas de agrupación lleva a cabo la reducción de dimensión, reduciendo así el número de parámetros en el input. De forma similar a la capa convolucional, la capa de agrupación pasa un filtro a través del input, pero con la diferencia de que el filtro no tiene coeficientes, si no que emplea una función de agregación a los valores recibidos según los atributos seleccionados. Existen varias formas de agrupar, la primera, que selecciona el máximo entre los píxeles a la hora de mandarlo al output, y la segunda, que selecciona la media entre los píxeles. Es evidente que gran cantidad de información se pierde, pero ayuda a reducir la complejidad y el riesgo de sobreajuste.

Por último tenemos la capa completamente conectada, en esta capa cada nodo en la capa de salida se conecta directamente con los nodos de la capa anterior. Esta capa se encarga de clasificar basado en atributos extraídos de capas anteriores y sus diferentes filtros. Esta capa suele usar la función de activación softmax, a diferencia de las anteriores que usan la ReLU, produciendo así una probabilidad entre 0 y 1.

Todo esto queda representado en la figura 2.7

## ARQUITECTURA DE UNA CNN

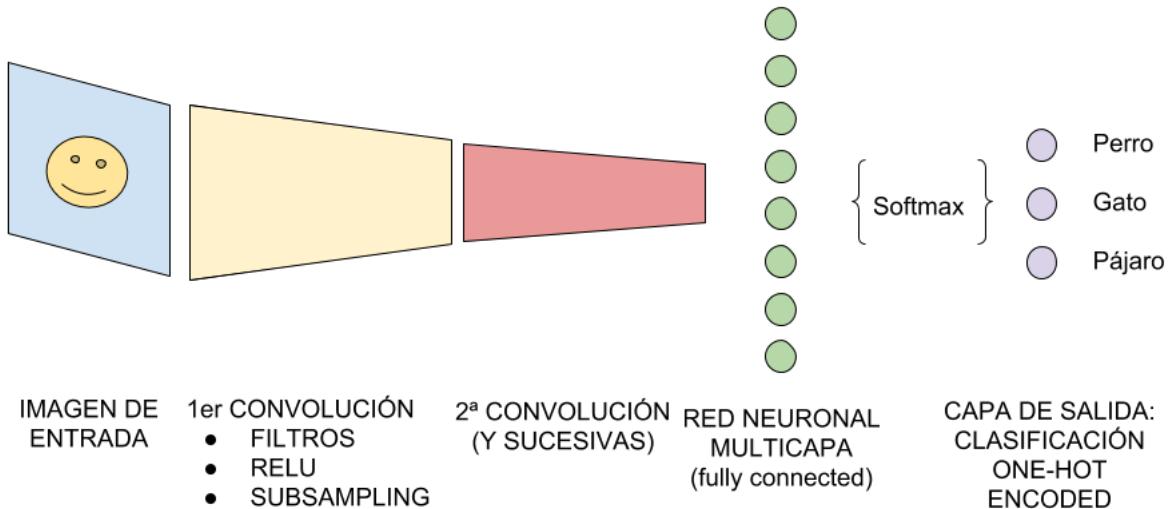


Figura 2.7: Representación gráfica de una red neuronal convolucional

### 2.2.3 Red Neuronal Residual

Tras la primera arquitectura basada en redes neuronales convolucionales (AlexNet) que ganó la competición *ImageNet 2012*, cada arquitectura ganadora posterior simplemente ha usado más capas para reducir la relación de error. Esta tendencia ha sido efectiva para cantidades pequeñas de capas pero, conforme se incrementaron, apareció un problema común en el aprendizaje profundo llamado el problema del desvanecimiento/exploración del gradiente, que provoca que este sea cero o demasiado grande.

Por esto, un incremento de capas no siempre resulta en una tasa de error menor. Debido a esto, fue propuesto en 2015 por parte de investigadores de Microsoft Research una nueva arquitectura llamada red residual.

La **Red Neuronal Residual** [22], con el objetivo de reducir el problema del desvanecimiento/exploración de gradiente, introdujo el concepto de bloques residuales. En esta red se usa un concepto denominado *salto de conexiones*, que permite a una capa activar capas lejanas sin necesidad de activar las que están entre ellas, formando así un bloque residual. Las rednets son creadas apilando estos bloques residuales juntos.

La intención detrás de esto es que la red en vez de aprender el mapeo subyacente, se ajuste al mapeo residual. Así que, en vez de aprender el mapeo inicial, digamos  $H(x)$ , permitir que la red mapee lo siguiente:

$$F(x) := H(x) - x \text{ que nos lleva a } H(x) := F(x) + x$$

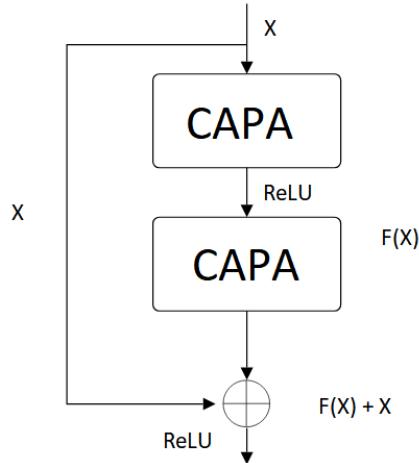


Figura 2.8: Esquema de bloque residual de red neuronal residual

La ventaja de añadir este tipo de conexiones es que si alguna capa perjudica el desempeño de la arquitectura, entonces será saltada. Esto da como resultado una red neuronal muy profunda sin los problemas causados por el desvanecimiento/explosión de gradiente.

## 2.2.4 Teorema de Aproximación Universal

Después de revisar todos estos conceptos sobre las redes neuronales es evidente que pueden llegar a ser muy útiles, pero, ¿qué nos asegura que sea capaz de describir cualquier función? ¿Existen funciones que no sean descriptibles por una red neuronal lo suficientemente grande? Todo esto puede ser contestado mediante lo que se denomina el Teorema de aproximación universal.

El **Teorema de Aproximación Universal**, en nuestro caso aplicado a las redes neuronales convolucionales [3] que son las usadas por AlphaZero, prueba que dada una función definida en un conjunto compacto en un espacio n-dimensional, existe una red neuronal convolucional que aproxima dicha función. Esto tan solo prueba su existencia, pero no nos proporciona un método capaz de construir dicha red neuronal.

**Demostración.** Para empezar la demostración, para la que nos basaremos en [46], empezaremos definiendo la red neuronal que consideraremos y aprovecharemos para demostrar la eficiencia de las redes neuronales convolucionales. Como en AlphaZero, nuestra red neuronal convolucional tendrá una ReLU, definida como función  $\sigma$  no lineal dada por  $\sigma(u) = (u)_+ = \max\{u, 0\}$ ,  $u \in \mathbb{R}$  y una

serie de máscaras de filtro convolucionales  $w = \{w^{(j)}\}_j$ . Aquí una máscara de filtro  $w = (w_k)_{k=-\infty}^{\infty}$  representa una serie de coeficientes de filtros. Usamos una longitud de filtrado fijada  $s \geq 2$  para controlar la dispersión, y asumir que  $w_k^{(j)} \neq 0$  solo para  $0 \leq k \leq s$ . La convolución de tal máscara de filtro  $w$  con otra serie  $v = (v_0, \dots, v_D)$  es una serie  $w * v$  dada por  $(w * v)_i = \sum_{k=0}^D w_{i-k} v_k$ . Esto nos lleva a una matriz convolucional del tipo Toeplitz con diagonales constantes de tamaño  $(D + s) \times D$ .

$$T = \begin{bmatrix} w_0 & 0 & 0 & 0 & \dots & 0 \\ w_1 & w_0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ w_s & w_{s-1} & \dots & w_0 & 0 & \dots & 0 \\ 0 & w_s & \dots & w_1 & w_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \dots & \dots & 0 & w_s & \dots & w_0 \\ \dots & \dots & \dots & 0 & w_s & \dots & w_1 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & 0 & w_s & w_{s-1} \\ 0 & \dots & \dots & \dots & \dots & 0 & w_s \end{bmatrix}$$

Nótese que el número de filas de  $T$  es el número de columnas  $+s$ . Esto nos lleva a tomar una serie de incrementos lineales de amplitud  $\{d_j = d + js\}$  para la red, que permite a la red neuronal convolucional representar funciones más complejas.

Empezando con el efecto de la matriz  $T$  en el vector de los datos de entrada  $x \in \mathbb{R}^d$ , podemos definir una red neuronal convolucional de profundidad  $J$  como una serie de  $J$  vectores  $h^{(j)}(x)$  de funciones en  $\mathbb{R}^d$  dadas iterativamente por  $h^{(0)}(x) = x$  y

$$h^{(j)}(x) = \sigma(T^{(j)}h^{(j-1)}(x) - b^{(j)}), j = 1, 2, \dots, J,$$

donde  $T^{(j)} = (w_{i-k}^{(j)})$  es una matriz convolucional de tamaño  $d_j \times d_{j-1}$ ,  $\sigma$  función y  $b$  es una serie de vectores de sesgo  $b^{(j)}$ . Excepto la última iteración, tomamos  $b^{(j)}$  de la forma

$$[b_1 \dots b_s b_{s+1} b_{s+1} \dots b_{s+1} b_{d_j-s+1} \dots b_{d_j}]^T$$

con las  $d_j - 2s$  componentes repetidas en el medio. La dispersión de  $T^{(j)}$  y la forma de  $b^{(j)}$  nos señalan que la  $j$ -ésima iteración de la red neuronal convolucional toma  $3s + 2$  variables libres. Así que, junto a las  $2d_J + s + 1$  variables libres para  $b^{(J)}, c \in \mathbb{R}^{d_J}, w^{(J)}$ , el total de variables libres en la red neuronal convolucional es de  $(5s + 2)J + 2d - 2s - 1$ , mucho menor que en una clásica red completamente conectada multicapa con matrices completamente conectadas  $T^{(j)}$  con  $d_j d_{j-1}$

variables libres. De esta forma evidenciamos la eficiencia computacional de las redes neuronales convolucionales.

El espacio de hipótesis de un algoritmo de aprendizaje es el conjunto de todas las funciones posibles que pueden ser representadas o producidas por el algoritmo. Para la red neuronal convolucional de profundidad  $J$  considerada aquí, este es el espacio de funciones definido por:

$$\mathcal{H}_J^{w,b} = \left\{ \sum_{k=1}^{d_J} c_k h_k^{(J)}(x) : c \in \mathbb{R}^{d_J} \right\}.$$

El espacio de hipótesis y su habilidad de aproximación dependen completamente de la serie de máscaras de filtro convolucional  $w = \{w^{(j)}\}_{j=1}^J$  y la serie de vectores de sesgos  $b = \{b^{(j)}\}_{j=1}^J$ . Cada función del espacio  $\mathcal{H}_J^{w,b}$  es una función a trozos continua y lineal (spline lineal) en cualquier subconjunto compacto  $\Omega$  de  $\mathbb{R}^d$ .

Para enunciar los resultados primero debemos definir el espacio de Sobolev brevemente:

**| Definición 2.1.** *El espacio de Sobolev es un espacio vectorial normado de funciones que puede verse como un subespacio de un espacio  $L^p$ . Un espacio de Sobolev es un subespacio vectorial del espacio  $L^p$  formado por clases de funciones tales que sus derivadas hasta orden  $m$  también pertenecen a  $L^p$ .*

Dado un dominio  $\Omega \subset \mathbb{R}^n$  el espacio de Sobolev  $W^{m,p}(\Omega)$  es definido como:

$$W^{m,p}(\Omega) = \{f \in L^p(\Omega) | D^\alpha f \in L^p(\Omega), \forall \alpha \in \mathbb{N}^n : |\alpha| \leq m\} \subset L^p(\Omega)$$

Donde  $D^\alpha f$  es la notación multi-índice para las derivadas parciales. Este espacio está formado por clases de equivalencia de funciones.

La norma del espacio de Sobolev se define a partir de la norma  $\|\cdot\|_{L^p(\Omega)}$  de  $L^p$ :

$$\|f\|_{m,p,\Omega} = \left( \sum_{|\alpha| \leq m} \|D^\alpha f\|_{L^p(\Omega)^p} \right)^{\frac{1}{p}}, \quad 1 \leq p < \infty$$

$$\|f\|_{m,\infty,\Omega} = \max_{\alpha \leq m} \|D^\alpha f\|_{L^\infty(\Omega)}$$

El primer resultado que probaremos que verifica la universalidad de las redes neuronales convolucionales, teniendo en cuenta que cualquier función  $f \in C(\Omega)$ , el espacio de funciones continuas de  $\Omega$  con norma  $\|f\|_{C(\Omega)} = \sup_{x \in \Omega} |f(x)|$ , puede ser aproximado por  $\mathcal{H}_J^{w,b}$  a una arbitraria precisión cuando la profundidad  $J$  es suficiente, es el siguiente.

**| Teorema 2.1.** *Sea  $2 \leq s \leq d$ . Para cualquier subconjunto compacto  $\omega$  de  $\mathbb{R}^d$  y cualquier  $f \in C(\Omega)$ , existen series  $w$  de máscaras de filtros,  $b$  de vectores de sesgos y  $f_J^{w,b} \in \mathcal{H}_J^{w,b}$  tal que*

$$\lim_{J \rightarrow \infty} \|f - f_J^{w,b}\|_{C(\Omega)} = 0$$

**Demostración.** Para aproximar en  $C(\Omega)$  podemos considerar tan solo los espacios de Sobolev que pueden ser sumergidos en el espacio de funciones continuas, que son aquellos con un índice de regularidad  $r > \frac{d}{2}$ . Para establecer relaciones de aproximación requeriremos que  $r > \frac{d}{2} + 2$  en el siguiente teorema, en cuyo caso, el conjunto  $H^r(\Omega)$  es denso en  $C(\Omega)$ , por lo que la prueba del teorema se dará en la del teorema 2.2 al ser un caso particular. |

El segundo resultado presenta la razón de aproximación hechas por redes neuronales convolucionales para funciones en el espacio de Sobolev  $H^r(\Omega)$  con un índice  $r > 2 + d/2$ . Dicha función  $f$  es la restricción de  $\Omega$  de una función  $F$  del espacio de Sobolev  $H^r(\mathbb{R}^d)$  en  $\mathbb{R}^d$  significando que  $F$  y todas sus derivadas parciales hasta las de orden  $r$  son funciones de cuadrado integrable en  $\mathbb{R}^d$ .

**Teorema 2.2.** Sea  $2 \leq s \leq d$  y  $\Omega \subseteq [-1, 1]^d$ . Si  $J \geq 2d/(s-1)$  y  $f = F|_{\Omega}$  con  $F \in H^r(\mathbb{R}^d)$  y un índice  $r > 2 + d/2$ , entonces existe  $w, b$  y  $f_J^{w,b} \in \mathcal{H}_J^{w,b}$  tal que

$$\|f - f_J^{w,b}\|_{C(\Omega)} \leq c \|F\| \sqrt{\log J} (1/J)^{\frac{1}{2} + \frac{1}{d}}$$

donde  $c$  es una constante no negativa y  $\|F\|$  denota la norma de Sobolev de  $F \in H^r(\mathbb{R}^d)$ .

**Demostración.** Sea  $J \geq \frac{2d}{s-1}$  y  $m$  la parte entera de  $\frac{(s-1)J}{d} - 1 \geq 1$ . Sea  $f = F|_{\Omega}$  para alguna función  $F \in H^r(\mathbb{R}^d)$  con la transformación de Fourier  $\hat{F}(w)$  y con la norma  $\|F\| = \|(1 + |w|^2)^{\frac{r}{2}} \hat{F}(w)\|_{L^2}$ . Por la desigualdad de Schwarz y la condición  $r > \frac{d}{2} + 2$ ,  $v_{F,2} := \int_{\mathbb{R}^d} \|w\|_1^2 |\hat{F}(w)| dw \leq c_{d,r} \|F\|$  donde  $c_{d,r}$  es la constante finita  $\||w\|_1^2 (1 + |w|^2)^{\frac{r}{2}}\|_{L^2}$ . Entonces, aplicando un resultado de [31] de aproximación de ridge a  $F_{[-1,1]^d}$  y sabemos que existe una combinación lineal de funciones de ramp ridge de la forma

$$F_m(x) = \beta_0 + \alpha_0 x + \frac{v}{m} \sum_{k=1}^m \beta_k (\alpha_k x - t_k)_+$$

con  $\beta_k \in [-1, 1]$ ,  $\|\alpha_k\|_1 = 1$ ,  $t_k \in [0, 1]$ ,  $\beta_0 = F(0)$ ,  $\alpha_0 = \nabla F(0)$  y  $|v| \leq 2v_{F,2}$  tal que

$$\|F - F_m\|_{C([-1,1]^d)} \leq c_0 v_{F,2} \max\{\sqrt{\log m}, \sqrt{d}\} m^{\frac{-1}{2} - \frac{1}{d}}$$

para alguna constante universal  $c_0 > 0$ .

Ahora tratamos de construir la serie de máscaras de filtro  $w$ . Definiremos una serie  $W$  con soporte en  $\{0, \dots, (m+1)d-1\}$  apilando los vectores  $\alpha_0, \alpha_1, \dots, \alpha_m$  (con componentes al revés) con

$$[W_{(m+1)d-1} \dots W_1 W_0] = [\alpha_m^T \dots \alpha_1^T \alpha_0^T].$$

Aplicando el teorema 2.3, que se enunciará posteriormente, a la serie  $W$  con soporte en  $\{0, 1, \dots, (m+1)d\}$  y, junto a una serie de máscaras de filtro  $w = \{w^{(j)}\}_{j=1}^{\hat{J}}$  con soporte en  $\{0, 1, \dots, s\}$  y con  $\hat{J} < \frac{(m+1)d}{s-1} + 1$  tal que  $W = w^{(\hat{J})} * w^{(\hat{J}-1)} * \dots * w^{(2)} * w^{(1)}$ . La elección de  $m$  ha de ser tal que  $\frac{(m+1)d}{s-1} \leq J$ . Así que  $\hat{J} \leq J$  y tomando  $w^{(\hat{J}+1)} = \dots = w^{(J)}$  como serie delta, tenemos  $W = w^{(J)} * w^{(J-1)} * \dots * w^{(2)} * w^{(1)}$ . Esto nos sugiere que

$$T^{(J)} \dots T^{(1)} = T^W$$

donde  $T^W$  es la matriz de dimensión  $d_J \times d$  dada por  $[W_{l-k}]_l = 1, \dots, d_J, k = 1, \dots, d$ . Por la definición de  $W$  podemos ver que para  $k = 0, 1, \dots, m$ , la  $(k+1)d$ -ésima fila de  $T^W$  es exactamente la traspuesta de  $\alpha_k$ . Además, ya que  $J_s \geq (m+1)d$ , tenemos que  $W_{J_s} = 0$  y la última fila de  $T^W$  es una fila de ceros.

Entonces construimos  $b$ . Sea  $\|w\|_1 = \sum_{k=-\infty}^{\infty} |w_k|$ ,  $B^{(0)} = \max_{x \in \Omega} \max_{k=1, \dots, d} |x_k|$  y  $B^{(j)} = \|w^{(j)}\|_1 \dots \|w^{(1)}\|_1 B^{(0)}$  para  $j \geq 1$ . De esta forma tenemos:

$$\|(T^{(j)} \dots T^{(1)} x)_k\|_{C(\Omega)} \leq B^{(j)}, \forall k = 1, \dots, d_j.$$

Tomamos  $b^{(1)} = -B^{(1)} 1_{d_1} := -B^{(1)}(1, \dots, 1)^T$ , y

$$b^{(j)} = B^{(j-1)} T^{(j)} 1_{d_{j-1}} - B^{(j)} 1_{d_j}, j = 1, \dots, J-1.$$

Entonces, para  $j = 1, \dots, J-1$ , tenemos:

$$h^{(j)}(x) = T^{(j)} \dots T^{(1)} x + B^{(j)} 1_{d_j}$$

y  $b_l^{(j)} = B^{(j-1)} \sum_{k=0}^s w_k^{(j)} - B^{(j)} = b_{s+1}^{(j)}$  para  $l = s+1, \dots, d_j - s$ . Finalmente, tomamos el vector de sesgo  $b^{(j)}$  ajustando  $b_l^{(j)}$  para ser

$$\begin{cases} B^{(J-1)}(T^{(J)} 1_{dJ-1})_l - B^{(J)} & \text{si } l = d, d+J, \\ B^{(J-1)}(T^{(J)} 1_{dJ-1})_l + t_k & \text{si } l = (k+1)d, 1 \leq k \leq m, \\ B^{(J-1)}(T^{(J)} 1_{dJ-1})_l + B^{(J)} & \text{en otro caso} \end{cases}$$

Sustituyendo el vector de sesgo y la expresión por  $h^{(J-1)}(x)$  en la la relación iterativa de la red neuronal convolucional que podemos ver de la identidad  $T^{(J)} \dots T^{(1)} = T^W$  y la definición de la serie  $W$ , que el  $l$ -ésimo componente  $h_l^{(J)}(x)$  de  $h^{(J)}(x)$  es igual a:

$$\begin{cases} \alpha_0 x + B^{(J)}, & \text{si } l = d, \\ B^{(J)} & \text{si } l = d+J, \\ (a_k x - t_k)_+, & \text{si } l = (k+1)d, 1 \leq k \leq m, \\ 0, & \text{en otro caso.} \end{cases}$$

Por lo tanto, podemos tomar  $f_J^{w,b} = F_m|_{\Omega} \in \text{span}\{h_k^{(J)}(x)\}_{k=1}^{d_J} = \mathcal{H}_J^{w,b}$  y sabemos que el error  $\|f - f_J^{w,b}\|_{C(\Omega)} \leq \|F - F_m\|_{C([-1,1]^d)}$  puede ser acotado por:

$$\|f - f_J^{w,b}\|_{C(\Omega)} \leq c_0 v F, 2 \max\{\sqrt{\log m}, \sqrt{d}\} m^{-\frac{1}{2} - \frac{1}{d}}$$

Pero  $\frac{1}{2}(s-1)J < md < (s-1)J$  y  $2r-d-4 \geq 1$ . Mediante una transformación por coordenadas polares,  $c_{d,r} d^{1+\frac{1}{d}} \leq \sqrt{\frac{d^6 \pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)}} (1 + \frac{1}{\sqrt{2r-d-4}})$  que puede ser acotado por una constante no negativa  $c' := \max_{l \in \mathbb{N}} 2 \sqrt{l^6 \pi^{l/2} / \Gamma(\frac{l}{2}+1)}$ . Por ello:

$$\|f - f_J^{w,b}\|_{C(\Omega)} \leq 2c_0 c' \|F\| \sqrt{\log J} J^{-\frac{1}{2} - \frac{1}{d}}.$$

Esto prueba el teorema 2.2 tomando  $c = 2c_0 c'$ .

Según el teorema 2.2, si tomamos  $s = \lceil 1 + d^\tau / 2 \rceil$  y  $J = \lceil 4d^{1-\tau} \rceil L$  con  $0 \leq \tau \leq 1$  y  $L \in \mathbb{N}$  donde  $\lceil u \rceil$  denota el menor entero no menor que  $u$ , entonces tenemos

$$\|f - f_J^{w,b}\|_{C(\Omega)} \leq c \|F\| \sqrt{\frac{(1-\tau) \log d + \log L + \log 5}{4d^{1-\tau} L}}$$

mientras la anchura de la red neuronal convolucional esté acotada por  $12L_d$  y el número total de variables libres cumpla  $(5s+2)J + 2d - 2s - 1 \leq (73L+2)d$ . Podemos incluso tomar  $L = 1$  y  $\tau = 1/2$  para tener una cota del error relativo

$$\frac{\|f - f_J^{w,b}\|_{C(\Omega)}}{\|F\|} \leq \frac{c}{2} d^{-\frac{1}{4}} \sqrt{\log(5\sqrt{d})}$$

conseguido por la red neuronal convolucional de profundidad  $\lceil 4\sqrt{d} \rceil$  y como mucho  $75d$  variables libres, que disminuyen conforme  $d$  crece. Esto permite explicar la habilidad de aproximación tan fuerte de las redes neuronales convolucionales.

El tercer y último resultado nos da un resultado más general que mejora la acotación de  $J$  en [47] y elimina las restricciones especiales en  $W_0$  y  $W_s$  para las que  $W_0 > 0$  y  $W_s \neq 0$ .

**Teorema 2.3.** *Sea  $s \geq 2$  y  $W = (W_k)_{-\infty}^{\infty}$  una serie con soporte en  $\{0, \dots, M\}$  con  $M \geq 0$ . Entonces existe una serie finita de máscaras de filtro  $\{w^{(j)}\}_{j=1}^J$  con soporte en  $\{0, \dots, s\}$  con  $J < \frac{M}{s-1} + 1$  tal que la factorización convolucional  $W = w^{(J)} * \dots * w^{(2)} * w^{(1)}$  es cierta.*

**Demostración.** Para la prueba del teorema 2.3 usaremos un concepto de las *ondículas*, descrito en [13], la variable  $\tilde{w}$  de una serie con soporte finito en el conjunto de los enteros no negativos, definido como polinomio en  $\mathbb{C}$  por  $\tilde{w}(z) = \sum_{k=0}^{\infty} w_k z^k$ . La serie convolucionada  $a * b$  viene dada por  $a * b(z) = \tilde{a}(z)\tilde{b}(z)$ .

Por ello,  $\tilde{W}$ , de la serie  $W$  con soporte en  $\{0, \dots, M\}$  es un polinomio de grado  $M$  con coeficientes reales para algún  $M$  entre 0 y  $M$ . Sabemos que las raíces complejas  $z_k = x_k + iy_k$  de  $\tilde{W}$  con  $x_k \neq 0$  aparece en pares y ya que  $(z - z_k)(z - \bar{z}_k) = z^2 - 2x_k z + (x_k^2 + y_k^2)$ , el polinomio  $\tilde{W}(z)$  puede ser completamente factorizado como:

$$\tilde{W}(z) = W_M \prod_{k=1}^K \{z^2 - 2x_k z + (x_k^2 + y_k^2)\} \prod_{k=2K+1}^M (z - x_k),$$

donde  $2K$  es el número de raíces complejas contando multiplicidad y  $M - 2K$  es el número de raíces reales contando multiplicidad. Tomando grupos de  $s/2$  factores cuadráticos y  $s$  factores lineales en la anterior factorización, tenemos  $\tilde{W}(z) = \tilde{w}^{(J)}(z) \dots \tilde{w}^{(1)}(z)$ , una factorización de  $\tilde{W}$  en polinomios de grado hasta  $s$ , que tomamos como factorización convolucional  $W = w^{(J)} * w^{(J-1)} * \dots * w^{(1)}$ . |

Y aquí, tras haber demostrado los 3 resultados, finaliza la prueba del Teorema de Aproximación Universal. |

## 2.2.5 Teorema No Free Lunch

En el mundo de la Inteligencia Artificial es muy común encontrarnos con el conjunto de **Teoremas No Free Lunch** pero, ¿hasta qué punto es el resultado aplicable en la práctica?

| **Teorema 2.4.** *Para cualquier algoritmo de optimización, cualquier mejora en el desempeño sobre una clase de problemas se compensa con un desempeño inferior en otra clase, es decir, no existe un algoritmo óptimo universal para todos los problemas de optimización*

**Demostración.** Probaremos el resultado para el caso en el que el output es una función binaria, o sea, para cualquier algoritmo de aprendizaje que devuelva una función binaria existe una distribución que provocará que devuelva una hipótesis con una tasa de error alto, incluso cuando existe otra función que puede alcanzar un error nulo en la distribución.

Sea  $A$  algoritmo de aprendizaje que devuelve un clasificador binario en el dominio  $X$  y definamos la función de error que marcará cómo de bien nuestra hipótesis de salida lo hace en la distribución actual. Usaremos la  $L$ -función binaria **1** que, dados dos inputs, devuelve 0 si los dos valores son iguales, y 1 en cualquier otro caso.

Entonces, existe una distribución  $D$  en  $X \times \{0, 1\}$  tal que:

1. Existe una función  $f : X \rightarrow \{0, 1\}$  con  $L_D(f) = 0$ . O sea, hay alguna función binaria en  $X$  tal que la pérdida esperada sobre  $D$  es 0, una función que clasificará correctamente todos los puntos en nuestra distribución correctamente.
2. Con probabilidad de al menos  $\frac{1}{7}$  sobre la elección de  $S \sim D^m$   $L_D(A(S)) \geq \frac{1}{8}$ . En otras palabras, con al menos una probabilidad de  $\frac{1}{7}$  de todos los posibles conjuntos de aprendizaje de tamaño  $m$  en  $D$ , la pérdida esperada del output del algoritmo de aprendizaje es de, al menos,  $\frac{1}{8}$ . ( Nótese que nuestra función de error está definida aquí como la pérdida promedio de 1 entre todos los puntos de la distribución).
3. La distribución  $D$  es definida en más de  $2m$  puntos, así que nuestro conjunto de aprendizaje consistirá en al menos  $m$  valores únicos.

Para probar esto, probaremos algo equivalente, que es:

$$\max_{i \in \{T\}} \mathbb{E}_{S \sim D_i^m} [L_{D_i}(A(S))] \geq \frac{1}{4}$$

Por lo que, si ejecutamos nuestro algoritmo de aprendizaje en conjuntos de aprendizaje de tamaño  $m$  sobre una familia de distribuciones  $D_1 - D_T$ , tomará al menos  $\frac{1}{4}$  como pérdida esperada en al menos uno, por lo que al menos una distribución no será bien aproximada por nuestro algoritmo.

Para ver que estos dos resultados son equivalentes tomamos una versión modificada de la desigualdad de Markov,  $P(X \geq a) \geq \frac{\mathbb{E}[X] - a}{1 - a}$ . Aplicando esto último a nuestra desigualdad llegamos a:

$$P(L_{D_i}(A(S)) \geq \frac{1}{8}) \geq \frac{\frac{1}{4} - \frac{1}{8}}{\frac{7}{8}} \geq \frac{1}{7}$$

Empecemos ahora con la prueba, sea  $C \subseteq X$  tal que  $|C| = 2m$  para que sea un subconjunto de nuestro espacio de entrada de tamaño  $2m$ .

Teniendo en cuenta que hay  $T = 2^{2m}$  funciones posibles desde  $C$  a  $\{0, 1\}$ , sea  $D_i$  la distribución sobre  $D$  que solo asigna probabilidad a los pares  $(x, f_i(x))$  para la función correspondiente. Esto refleja la idea de que nuestro espacio de inputs  $C$  es del doble de tamaño que nuestro conjunto de aprendizaje, y que cada función es igualmente probable.

Cada función es ahora una distribución de probabilidad que toma como valor  $\frac{1}{|C|}$  si el punto  $(x, y)$  es uno de los pares de input/output de la función, y 0 en cualquier otro caso. De esta manera, cada función única tiene una distribución asociada con ella única.

Para cada distribución,  $\mathbb{E}_{S \sim D_i^m} [L_{D_i}(A(S))]$  es la pérdida para un conjunto de aprendizaje dado, tomada la media sobre todas las opciones posibles de datos de aprendizaje.

Si nuestro espacio de input consiste en  $2m$  puntos y nuestros datos de aprendizaje consisten en una sucesión de  $m$  puntos, entonces hay  $k = (2m)^m$  sucesiones posibles.

Sea  $S_j^i$  la  $i$ -ésima posible sucesión de datos de aprendizaje, clasificada por la  $j$ -ésima posible función. Dicho esto, sabemos que  $\mathbb{E}_{S \sim D_i^m} [L_{D_i}(A(S))] = \frac{1}{k} \sum_{j=1}^k L_{D_i}(A(S_j^i))$ , para cada distribución de datos dada, la pérdida esperada de nuestro algoritmo de aprendizaje (a través de todos los conjuntos de datos de aprendizaje) es igual a la pérdida esperada si tomamos  $m$  puntos de aprendizaje, tomada la media sobre todas las funciones posibles que podían clasificar los datos.

Teniendo en cuenta que el error máximo sobre todas las posibles distribuciones es, al menos, tan grande como la pérdida media entre todas las distribuciones. Por lo que sabemos que:

$$\max_{i \in [T]} \frac{1}{k} \sum_{j=1}^k L_{D_i}(A(S_j^i)) \geq \frac{1}{T} \sum_{i=1}^T \frac{1}{k} \sum_{j=1}^k L_{D_i}(A(S_j^i))$$

No importa realmente si tomamos la media sobre todas las distribuciones primero o la media sobre todos los conjuntos de aprendizaje primero, por lo tanto, podemos cambiar el orden de las sumas para obtener el mismo valor:

$$\frac{1}{T} \sum_{i=1}^T \frac{1}{k} \sum_{j=1}^k L_{D_i}(A(S_j^i)) = \frac{1}{k} \sum_{j=1}^k \frac{1}{T} \sum_{i=1}^T L_{D_i}(A(S_j^i))$$

Si tomamos la media de pérdida sobre todos los datos de aprendizaje, este será al menos tan grande como la pérdida dada por los datos de aprendizaje, que nos lleva a la pérdida media menor.

Dicho esto, podemos sustituir nuestra desigualdad por una más simple. Nuestra desigualdad tiene que ser al menos tan grande como la pérdida media (sobre todas las funciones) cuando es aplicada a ese conjunto de aprendizaje específico.

$$\frac{1}{k} \sum_{j=1}^k \frac{1}{T} \sum_{i=1}^T L_{D_i}(A(S_j^i)) \geq \min_{j \in [k]} \frac{1}{T} \sum_{i=1}^T L_{D_i}(A(S_j^i))$$

Hasta ahora hemos estado hablando de forma abstracta de la función de pérdida. ¿Qué significa  $L_{D_i}(h)$  realmente? Queremos que sea un cuantificador de cómo bien nuestra hipótesis actúa en una distribución específica  $D_i$ . Una forma fácil de formalizar esto es evaluar todos los pares  $(x, y)$  en  $D_i$  y tomar la **1**-pérdida media entre  $y$  y  $h(x)$ :

$$LD_i(h) = \frac{1}{2m} \sum_{x \in C} \mathbf{1}[h(x) \neq f_i(x)]$$

Fijemos algún  $j \in [k]$ , que tiene  $m$  ejemplos de aprendizaje dados por  $S_j = (x_1, \dots, x_m)$ . Sean  $(v_1, \dots, v_p)$  los ejemplos en  $C$  (todos los puntos de los datos) que no están en  $S_j$ . Podemos ver que  $p \geq m$  porque como máximo tenemos la mitad de los datos de la distribución, que es cuando no tenemos duplicados. Por lo tanto,

$$L_{D_i}(h) = \frac{1}{2m} \sum_{x \in C} \mathbf{1}[h(x) \neq f_i(x)] \geq \frac{1}{2p} \sum_{j=1}^p \mathbf{1}[h(v_j) \neq f_i(v_j)]$$

La pérdida esperada sobre todos los puntos en la distribución es al menos la mitad de la pérdida esperada sobre todos los puntos que no están en el conjunto de aprendizaje.

Ahora podemos sustituir esto por  $L_{D_i}$  en nuestra desigualdad anterior para llegar a:

$$\frac{1}{T} \sum_{i=1}^T L_{D_i}(A(S_j^i)) \geq \frac{1}{T} \sum_{i=1}^T \frac{1}{2p} \sum_{j=1}^p \mathbf{1}[h(v_j) \neq f_i(v_j)]$$

Así que ahora estamos tomando la pérdida sobre todas las funciones, sobre todos los puntos en  $(v_1, \dots, v_p)$  y, como antes, podemos intercambiar el orden de las sumas:

$$\frac{1}{T} \sum_{i=1}^T \frac{1}{2p} \sum_{j=1}^p \mathbf{1}[h(v_j) \neq f_i(v_j)] = \frac{1}{2p} \sum_{j=1}^p \frac{1}{T} \sum_{i=1}^T \mathbf{1}[h(v_j) \neq f_i(v_j)]$$

De nuevo, podemos acotar esta desigualdad teniendo en cuenta que la pérdida media sobre todos los puntos en  $(v_1, \dots, v_p)$  y sobre todas las funciones tiene que ser al menos la pérdida del punto  $v_r$  que asumiremos que es el punto con pérdida mínima:

$$\frac{1}{2p} \sum_{j=1}^p \frac{1}{T} \sum_{i=1}^T \mathbf{1}[h(v_j) \neq f_i(v_j)] \geq \frac{1}{2} \min_{r \in [p]} \frac{1}{T} \sum_{i=1}^T \mathbf{1}[h(v_r) \neq f_i(v_r)]$$

Podemos tomar una partición de  $[T]$  en pares  $(f_a, f_b)$  tales que  $f_a$  y  $f_b$  toman siempre el mismo valor excepto en  $v_r$ . Podemos hacer esto ya que  $[T]$  es el conjunto de todas las funciones, así que podemos fijar  $f_a(v_r) = 0$  y dejar que el resto de valores varíen, como también podemos fijar  $f_b(v_r) = 1$  y hacer lo mismo. Es claro que esto nos dará dos conjuntos del mismo tamaño y que permite construir esos pares con las condiciones dadas.

Esto significa que:

$$\mathbf{1}[h(v_j) \neq f_a(v_j)] + \mathbf{1}[h(v_j) \neq f_b(v_j)] = 1$$

por lo que la pérdida media del par de funciones es  $\frac{1}{2}$  ya que nuestra hipótesis solo puede acertar una.

Finalmente, y debido a esto, podemos sustituir  $\frac{1}{2}$  por  $\sum_{i=1}^T \mathbf{1}[h(v_j) \neq f_i(v_j)]$  en nuestra desigualdad anterior y tener:

$$\frac{1}{2} \min_{r \in [p]} \frac{1}{T} \sum_{i=1}^T \mathbf{1}[h(v_j) \neq f_i(v_j)] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Y como solo hemos estado acotando superiormente, nuestro valor final  $\frac{1}{4}$  aplica a nuestra desigualdad inicial.

Por lo tanto,

$$\max_{i \in [T]} \frac{1}{k} \sum_{j=1}^k L_{D_i}(A(S_j^i)) \geq \frac{1}{4}$$

|

Pero esto contradice lo que a día de hoy podemos observar en la práctica, ¿por qué parece que unos algoritmos funcionan mejor que otros?

Un planteamiento popular considera que es importante observar la mayor diferencia entre los resultados empíricos y el teorema NFL, y este es que nuestro espacio de búsqueda está bastante más restringido que lo que creemos, nunca buscamos en el conjunto de todas las funciones posibles. La vida real no nos proporciona distribuciones adversarias y, por esto, tenemos información previa sobre a lo que el resultado final debería parecerse.

Para información más detallada acerca de este teorema se puede consultar [24] donde se explica desde otra perspectiva o en el capítulo 5 de [36].

## 2.3 Aprendizaje por Refuerzo

El *Aprendizaje por Refuerzo*, ([40] y [34] capítulo 17), es un área del aprendizaje automático inspirado en la psicología conductista, su objetivo es encontrar qué acciones debe escoger un agente de software en un entorno específico para maximizar la recompensa.

Hay 3 elementos básicos dentro del aprendizaje por refuerzo, el agente, que puede elegir qué acción ejecutar en el estado actual, el entorno, el cual responde a las acciones y sirve de input para el agente, y la recompensa, que es un mecanismo acumulativo que devuelve el entorno.

Para formalizar estos conceptos se usan los denominados procesos de decisión de Markov, que consta de lo siguiente:

- Un conjunto de estados  $S$ , puede ser infinito.
- Un estado inicial  $s_0 \in S$ .
- Un conjunto de acciones  $A$ , puede ser infinito.
- Una función de probabilidad de transición de estados  $\mathbb{P}[s'|s, a]$ : distribuida sobre los estados destino  $s' = \delta(s, a)$ .
- Una función de probabilidad de recompensas  $\mathbb{P}[r'|s, a]$ : distribuida sobre las recompensas devueltas  $r' = r(s, a)$ .

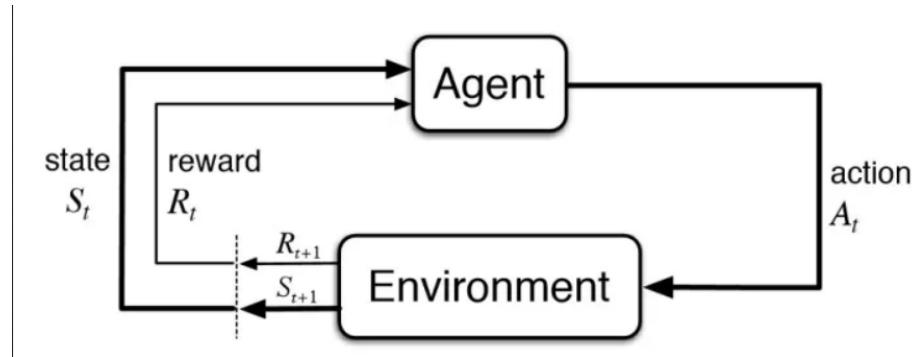


Figura 2.9: Esquema del funcionamiento de un algoritmo de aprendizaje por refuerzo

El modelo se denomina así ya que tanto la función de transición como la de recompensa dependen tan solo del estado actual  $s$  y no de los anteriores estados y acciones escogidos. Esta definición de procesos de decisión de Markov puede ser extendida a casos de estados y acciones no discretos, aunque en nuestro caso nos centraremos en el primero ya que está dividido en turnos.

En un modelo de tiempo discreto las acciones son tomadas de un conjunto de decisiones temporal  $\{0, \dots, T\}$ . Cuando  $T$  es finito, se dice que el modelo de procesos de decisión de Markov tiene horizonte finito, e independientemente de  $T$ , se califica como finito si tanto  $\mathcal{S}$  como  $\mathcal{A}$  son conjuntos finitos.

En nuestro caso la recompensa  $r(s, a)$  en el estado  $s$  cuando se toma la acción  $a$  viene dada por una variable aleatoria. En muchos casos, la recompensa puede ser una función determinista para el par  $(s, a)$ .

Ahora definiremos una política de acción o una táctica, que define cómo nuestro agente determina la acción a tomar en cada estado.

**| Definición 2.2.** Una táctica es un mapeo  $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ , donde  $\Delta(\mathcal{A})$  es un conjunto de distribuciones de probabilidad sobre  $\mathcal{A}$ . Una táctica  $\pi$  es determinista si para cada  $s$  existe un único  $a \in \mathcal{A}$  tal que  $\pi(s)(a) = 1$ . En este caso, podemos identificar  $\pi$  con un mapeo desde  $\mathcal{S}$  a  $\mathcal{A}$  y usar  $\pi(s)$  para referirnos a esta acción.

De hecho, esto es la definición de táctica estacionaria, ya que la elección de la distribución de las acciones no depende del tiempo. Podríamos definir una política no estacionaria como secuencia de mapeos  $\pi_t : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ . En particular, para el caso de horizonte finito, una política no estacionaria es necesaria generalmente para optimizar la recompensa.

El objetivo del agente es encontrar una táctica que maximice la recompensa esperada. La recompensa devuelta si  $\pi$  es una táctica determinista sobre una específica secuencia de estados  $s_0, \dots, s_T$  es definida de la siguiente forma:

- Para un horizonte finito:  $\sum_{t=0}^T r(s_t, \pi(s_t))$

- Para un horizonte infinito:  $\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t))$ , donde  $\gamma \in [0, 1)$  constante.

Nótese que se devuelve un único escalar para una posible serie infinita de recompensas. En el caso de horizonte infinito, las recompensas iniciales son más consideradas que las posteriores, lógico ya que acciones más próximas a una recompensa son las que más relevancia tienen.

Una vez definida la táctica es coherente la siguiente definición de su valor para cada estado:

**Definición 2.3.** *El valor  $V_\pi(s)$  de una táctica  $\pi$  en un estado  $s \in S$  es definido como la recompensa esperada devuelta cuando se empieza en  $s$  usando la táctica  $\pi$ :*

- Para un horizonte finito:  $V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} [\sum_{t=0}^T r(s_t, a_t) | s_0 = s]$ ;
- Para un horizonte infinito:  $V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} [\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s]$ ;

donde la recompensa esperada es sobre una selección aleatoria de una acción  $a_t$  según la distribución  $\pi(s_t)$  y sobre los estados aleatorios  $s_t$  alcanzados y los valores de recompensa  $r(s_t, a_t)$ .

Empezando en un estado  $s \in S$ , para maximizar su recompensa, un agente busca una táctica  $\pi$  con el mayor valor  $V_\pi(s)$ . Veamos que para cualquier proceso de decisión de Markov finito con horizonte infinitos, existe una táctica que es óptima para cualquier estado inicial, su definición es la siguiente:

**Definición 2.4.** *Una táctica  $\pi^*$  es óptima si su valor es maximal para cada estados  $s \in S$ , o lo que es lo mismo, para cada táctica  $\pi$  y estado  $s \in S$ ,  $V_{\pi^*}(s) \geq V_\pi(s)$*

Mostremos que para cualquier proceso de decisión de Markov existe una táctica óptima determinista. Para ello, definiremos la siguiente función.

**Definición 2.5.** *La función de valor de estado-acción  $Q$  asociada a una táctica  $\pi$  se define para todo  $(s, a) \in S \times A$  como el retorno esperado por tomar la acción  $a \in A$  en el estado  $s \in S$  y siguiendo la táctica  $\pi$ :*

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}[r(s, a)] + \mathbb{E}_{a_t \sim \pi(s_t)} \left[ \sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a \right] \\ &= \mathbb{E} [r(s, a) + \gamma V_\pi(s_1) \mid s_0 = s, a_0 = a] \end{aligned}$$

Véase que  $\mathbb{E}_{a \sim \pi(s)} [Q_\pi(s, a)] = V_\pi(s)$  (*Ecuaciones de Bellman*).

Busquemos ahora una condición que permita asegurar la optimalidad. Para ello, demostraremos primero el siguiente teorema:

**Teorema 2.5 (Teorema de mejora de tácticas).** *Para cualesquiera dos políticas  $\pi$  y  $\pi'$ , la siguiente expresión se cumple:*

$$(\forall s \in \mathcal{S}, \mathbb{E}_{a \sim \pi'(s)}[Q_\pi(s, a)] \geq \mathbb{E}_{a \sim \pi(s)}[Q_\pi(s, a)]) \Rightarrow (\forall s \in \mathcal{S}, V_{\pi'}(s) \geq V_\pi(s))$$

Además, una desigualdad estricta para cualquier  $s \in \mathcal{S}$  en la parte izquierda implica una desigualdad estricta para al menos una  $s$  en la parte derecha.

**Demostración.** Asumamos que tanto  $\pi$  como  $\pi'$  cumplen la expresión de la izquierda. Para cada  $s \in \mathcal{S}$  tenemos:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_{a \sim \pi(s)}[Q_\pi(s, a)] \\ &\leq \mathbb{E}_{a \sim \pi'(s)}[Q_\pi(s, a)] \\ &= \mathbb{E}_{a \sim \pi'(s)}[r(s, a) + \gamma V_\pi(s_1) | s_0 = s] \\ &= \mathbb{E}_{a \sim \pi'(s)}[r(s, a) + \gamma \mathbb{E}_{a_1 \sim \pi(s_1)}[Q_\pi(s_1, a_1)] | s_0 = s] \\ &\leq \mathbb{E}_{a \sim \pi'(s)}[r(s, a) + \gamma \mathbb{E}_{a_1 \sim \pi'(s_1)}[Q_\pi(s_1, a_1)] | s_0 = s] \\ &= \mathbb{E}_{a \sim \pi'(s), a_1 \sim \pi'(s_1)}[r(s, a) + \gamma r(s_1, a_1) + \gamma^2 V_\pi(s_2) | s_0 = s] \end{aligned}$$

Continuando de esta forma podemos ver que para cualquier  $T \geq 1$ :

$$V_\pi(s) \leq \mathbb{E}_{a_t \sim \pi'(s_t)}[\sum_{t=0}^T \gamma^t \mathbb{E}[r(s_t, a_t)] + \gamma^{T+1} V_\pi(s_{T+1}) | s_0 = s]$$

Ya que  $V_\pi(s_{T+1})$  está acotado, tomando el límite cuando  $T \rightarrow \infty$  nos da:

$$V_\pi(s) \leq \mathbb{E}_{a_t \sim \pi'(s_t)}[\sum_{t=0}^{\infty} \gamma^t \mathbb{E}[r(s_t, a_t)] | s_0 = s] = V_{\pi'}(s)$$

Y de nuevo, cualquier desigualdad en la parte izquierda resulta en una desigualdad estricta en la cadena de desigualdades de arriba. |

Veamos ahora la condición de optimalidad de Bellman que usaremos para probar la existencia de una táctica determinista óptima posteriormente:

**| Teorema 2.6 (Condición de optimalidad de Bellman).** Una táctica  $\pi$  es óptima si y sólo si para cualquier par  $(s, a) \in \mathcal{S} \times \mathcal{A}$  con  $\pi(s)(a) > 0$  se cumple la siguiente expresión:

$$a \in \operatorname{argmax}_{a' \in \mathcal{A}} Q_\pi(s, a')$$

**Demostración.** Por el teorema de mejora de tácticas, si la condición de arriba no se cumple para algún  $(s, a)$  con  $\pi(s)(a) > 0$ , entonces la táctica  $\pi'$  no es óptima. Esto es ya que  $\pi$  entonces puede

ser mejorado definiendo  $\pi'$  tal que  $\pi'(s') = \pi(s)$  para  $s' \neq s$  y  $\pi'(s)$  basado en cualquier elemento de  $\text{argmax}_{a' \in \mathcal{A}} Q_\pi(s, a')$ .  $\pi'$  verifica que  $\mathbb{E}_{a \sim \pi'(s)}[Q_\pi(s, a)] = \mathbb{E}_{a \sim \pi(s)}[Q_\pi(s', a)]$  para  $s' \neq s$  y  $\mathbb{E}_{a \sim \pi'(s)}[Q_\pi(s, a)] > \mathbb{E}_{a \sim \pi(s)}[Q_\pi(s, a)]$ . Por lo que, por el teorema de mejoras tácticas,  $V_{\pi'}(s) > V_\pi(s)$  para al menos un  $s$  y  $\pi$  no es óptimo.

A la inversa, sea  $\pi'$  una táctica no óptima. Entonces existe una táctica  $\pi$  y al menos un estado  $s$  para el que  $V_{\pi'}(s) < V_\pi(s)$ . Por el teorema de mejoras tácticas esto implica que existe algún estado  $s \in \mathcal{S}$  con  $\mathbb{E}_{a \sim \pi'(s)}[Q_\pi(s, a)] < \mathbb{E}_{a \sim \pi(s)}[Q_\pi(s, a)]$ . Por lo tanto,  $\pi'$  no puede cumplir la condición de arriba. |

Ya tenemos todas las herramientas para demostrar la existencia de una táctica determinista óptima, veámosla:

**| Teorema 2.7 (Existencia de una táctica óptima determinista).** *Cualquier proceso de decisión de Markov finito admite una táctica óptima determinista*

**Demostración.** Sea  $\pi^*$  una táctica determinista que maximiza  $\sum_{s \in \mathcal{S}} V_\pi(s)$ .  $\pi^*$  existe ya que existen finitas tácticas deterministas. Si  $\pi^*$  no fuera óptima, por el teorema de condición de optimalidad de Bellman, existiría un estado  $s$  con  $\pi(s) \notin \text{argmax}_{a' \in \mathcal{A}} Q_\pi(s, a')$ . Por el teorema de mejora de tácticas,  $\pi^*$  podría ser mejorado eligiendo una táctica  $\pi$  con  $\pi(s) \in \text{argmax}_{a' \in \mathcal{A}} Q_\pi(s, a')$  y  $\pi$  coincidiendo con  $\pi^*$  para todos los demás estados. Pero entonces  $\pi$  verificaría  $V_{\pi^*}(s) \leq V_\pi(s)$  con una desigualdad estricta para al menos un estado. Esto contradice el hecho de que  $\pi^*$  maximice  $\sum_{s \in \mathcal{S}} V_\pi(s)$ . |

Tras demostrar la existencia de una táctica óptima determinista, consideremos ahora tan solo tácticas deterministas. Sea  $\pi^*$  una táctica óptima determinista, y sean  $Q^*$  y  $V^*$  sus correspondientes función de valor de estado-acción y función de valor. Por el teorema de condición de optimalidad de Bellman, podemos escribir la siguiente expresión:

$$\forall s \in \mathcal{S}, \pi^*(s) = \text{argmax}_{a \in \mathcal{A}} Q^*(s, a)$$

De esta forma, el conocimiento que brinda la función de valor de estado-acción  $Q^*$  es suficiente para que el agente determine la táctica óptima, sin ningún conocimiento directo de la recompensa o de las probabilidades de transición. Sustituyendo  $Q^*$  por su definición nos da el siguiente sistema de ecuaciones para los valores de la táctica óptima  $V^*(s) = Q^*(s, \pi^*(s))$ :

$$\forall s \in \mathcal{S}, V^*(s) = \max_{a \in \mathcal{A}} \{ \mathbb{E}[r(s, a)] + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}[s'|s, a] V^*(s') \}$$

Conocido como ecuaciones de Bellman, nótese que es un sistema de ecuaciones no lineal, debido a la presencia del operador max.

Veamos formas de evaluar las tácticas:

**| Teorema 2.8 (Ecuaciones de Bellman).** Los valores  $V_\pi(s)$  de una táctica  $\pi$  en estados  $s \in \mathcal{S}$  para un modelo de procesos de decisión de Markov de horizonte infinito cumplen el siguiente sistema de ecuaciones lineales:

$$\forall s \in \mathcal{S}, V_\pi(s) = \mathbb{E}_{a_1 \sim \pi(s)}[r(s, a_1)] + \gamma \sum_{s'} \mathbb{P}[s'|s, \pi(s)]V_\pi(s')$$

**Demostración.** Podemos descomponer la expresión del valor de la táctica como la suma del primer término y del resto, que admiten  $\gamma$  como multiplicador:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) | s_0 = s] \\ &= \mathbb{E}[r(s, \pi(s))] + \gamma \mathbb{E}[\sum_{t=1}^{\infty} \gamma^t r(s_{t+1}, \pi(s_{t+1})) | s_0 = s] \\ &= \mathbb{E}[r(s, \pi(s))] + \gamma \mathbb{E}[\sum_{t=1}^{\infty} \gamma^t r(s_{t+1}, \pi(s_{t+1})) | s_1 = \delta(s, \pi(s))] \\ &= \mathbb{E}[r(s, \pi(s))] + \gamma \mathbb{E}[V_\pi(\delta(s, \pi(s)))] \end{aligned}$$

Este sistema puede ser reescrito como  $V = R + \gamma PV$ , con  $P$  denotando la matriz de probabilidades de transición definida por  $P_{s,s'} = \mathbb{P}[s'|s, \pi(s)]$  para todo  $s, s' \in \mathcal{S}$ .  $V$  es la matriz columna de valores cuyo  $s$ -ésimo componente es  $V_s = V_\pi(s)$  y  $R$  es la matriz columna de recompensas cuyo  $s$ -ésimo componente es  $R_s = \mathbb{E}[r(s, \pi(s))]$ .

Veamos en el siguiente teorema que para un modelo de procesos de decisión de Markov finito, el sistema de ecuaciones lineales admite solución única.

**| Teorema 2.9.** Para un modelo de procesos de decisión de Markov finito, las ecuaciones de Bellman admite solución única dada por:

$$V_0 = (I - \gamma P)^{-1}R$$

**Demostración.** Las ecuaciones de Bellman pueden ser reescritas como  $(I - \gamma P)V = R$ , por lo que para probar el teorema tan solo es necesario ver que la matriz  $I - \gamma P$  es invertible. Veamos cómo hacerlo:

Nótese que debido a sus propiedades estocásticas, la siguiente expresión se cumple,

$$\|P\|_\infty = \max_s \sum_{s'} |P_{ss'}| = \max_s \sum_{s'} \mathbb{P}[s'|s, \pi(s)] = 1$$

Esto implica que  $\|\gamma P\|_\infty = \gamma < 1$ . Los autovalores de  $\gamma P$  son por lo tanto todos menores a 1 y, por lo tanto,  $(I - \gamma P)$  es invertible

### 2.3.1 Modelado con Procesos de Decisión de Markov

Como ejemplo de representación, en esta sección vamos a usar el famoso juego del Tetris, donde daremos una visión intuitiva del conjunto de estados, que consta del conjunto de todas las posibles configuraciones del tablero que describe todas las posiciones ocupadas del tablero, el próximo bloque, etc. Además, daremos la representación del estado inicial, que consta del tablero vacío junto al primer bloque a colocar.

Para que un estado sea válido deben verificarse algunas condiciones, por ejemplo, que ninguna pieza esté situada *flotando* sin estar apoyada en otras. Un ejemplo de estado sería el mostrado en la figura 2.10.

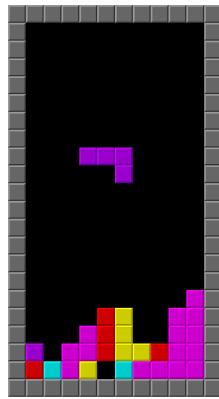


Figura 2.10: Ejemplo de estado en Tetris

El conjunto de acciones válidas vendrá dado por todos los movimientos válidos que se pueden hacer en el estado actual. Por ejemplo, para el estado inicial, sería el conjunto de todos los movimientos posibles en los que el bloque se situaría en la primera fila para cada una de sus posibles rotaciones. Un ejemplo de acciones sería el mostrado en la figura 2.11.

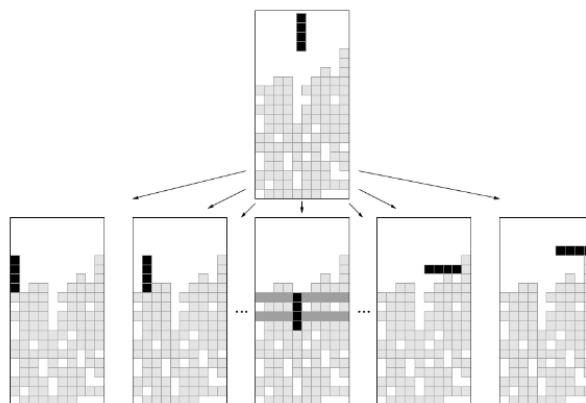


Figura 2.11: Ejemplo de conjunto de acciones en Tetris

La recompensa es la función que guía el objetivo del aprendizaje por refuerzo, tiene que describir qué es lo que queremos conseguir, en el caso de Tetris, depende de si buscamos sobrevivir mucho tiempo o si es un sistema de puntuación por líneas completadas, entre otras. En la tabla 2.1 podemos consultar los diferentes puntos en el Tetris original. En este caso, la recompensa viene dada por la puntuación explícita que proporciona el juego, pero en otros casos la recompensa adecuada deberá ser explorada para guiar adecuadamente el entrenamiento.

Nivel	Puntos por 1 línea	Puntos por 2 líneas	Puntos por 3 líneas	Puntos por 4 líneas
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
9	400	1000	3000	12000
n	$40 \times (n + 1)$	$100 \times (n + 1)$	$300 \times (n + 1)$	$1200 \times (n + 1)$

Cuadro 2.1: Puntos por completar líneas en diferentes niveles en Tetris original

En un ejemplo posterior mostraremos una representación completamente formalizada para un juego de dos jugadores que será completamente implementado y resuelto.

# 3 | Análisis General de AlphaZero

La idea general del algoritmo AlphaZero es tener una red neuronal capaz de guiar la búsqueda a través del árbol, produciendo valores que sirvan de guía y explorando las posibilidades a través de los nodos. Para ello usa una combinación de las estimaciones del valor y del interés de exploración para influenciar la búsqueda hacia nodos más prometedores [44].

El objetivo de este capítulo es describir el proceso detrás de AlphaZero haciendo uso de los elementos definidos anteriormente.

## 3.1 Introducción

Una de las principales diferencias entre AlphaZero y su predecesor, AlphaGo, es que la primera, además de poder jugar al go, shogi y ajedrez, no necesita ningún conocimiento adicional excepto las reglas del juego, demostrando así que un algoritmo de aprendizaje por refuerzo generalizado puede conseguir un desempeño superior a los humanos *tabula rasa*.

En vez de una función de evaluación creada manualmente y heurísticas que guíen los movimientos, AlphaZero utiliza un red neuronal profunda,  $(p, v) = f_\theta(s)$ , con parámetros  $\theta$ . Esta red neuronal toma la posición del tablero,  $s$ , como entrada y devuelve un vector de probabilidades de movimientos,  $p$ , con componentes  $p_a = Pr(a|s)$  para cada acción  $a$ , y un valor escalar  $v$  estimando el resultado esperado,  $z$ , desde la posición  $s$ ,  $v \approx \mathbb{E}[z|s]$ . AlphaZero aprende estas probabilidades de movimiento y estima valores exclusivamente mediante partidas que juega él solo, y que posteriormente usa para guiar su búsqueda.

AlphaZero usa MCTS como método de búsqueda en el árbol de jugadas. Cada búsqueda consiste en una serie de partidas simuladas que juega él solo recorriendo el árbol desde su nodo raíz a su nodo hoja. En cada simulación, se seleccionan movimientos para ambos jugadores mediante MCTS,  $a_t \sim \pi_t$ . Al final de la partida, la posición terminal,  $s_t$ , es fijada según las reglas del juego computado,  $z$ : -1 si es una derrota, 0 si es un empate, y +1 si es una victoria.

Los parámetros  $\theta$  son actualizados con el objetivo de minimizar el error entre el resultado predicho,  $v_t$ , y el resultado de la partida,  $z$ , y en maximizar el parecido entre el vector de tácticas,

$p_t$ , y las probabilidades de búsqueda,  $\pi_t$ . Los parámetros  $\theta$  son ajustados mediante descenso de gradiente con una función de pérdida que suma el error cuadrático medio y pérdidas de entropía cruzada respectivamente,

$$(p, v) = f_\theta(s),$$

$$l = (z - v)^2 - \pi^\top \log(p) + c\|\theta\|^2$$

donde  $c$  es un parámetro controlando la  $L_2$ -regularización de los pesos. Los parámetros actualizados son usados en las partidas posteriores que juega solo, en las que estima y optimiza el resultado esperado tomando en cuenta los resultados obtenidos.

AlphaZero no modifica los datos y la posición del tablero durante el MCTS, además, siempre mantiene una sola red neuronal que es actualizada continuamente, en vez de esperar a que se termine una iteración.

Las partidas que juega solo son generadas usando los últimos parámetros para la red neuronal, omitiendo el paso de evaluación y de selección del mejor jugador. En todas las partidas se vuelven a usar los hiperparámetros sin ninguna mejora específica, la única excepción a esto es el uso de ruido previo en la red de tácticas para asegurar la exploración.

### 3.1.1 Estructura General

En esta sección trataremos de explicar cómo funciona el algoritmo AlphaZero en rasgos generales, sin entrar en cada uno de sus elementos. Para ello, usaremos las siguientes imágenes de [20] y analizaremos los procesos que se ejecutan en el algoritmo.

Lo primero será definir el estado del juego (ver figura 3.1). que depende del juego para el que se pretenda entrenar la red. Consiste en una pila de matrices, cada una encargada de representar "algo", por ejemplo, una matriz podría encargarse de señalar las posiciones de un jugador, otra de señalar las posiciones del oponente, otra de señalar las posiciones finales de los movimientos válidos de una ficha en concreto, etc.

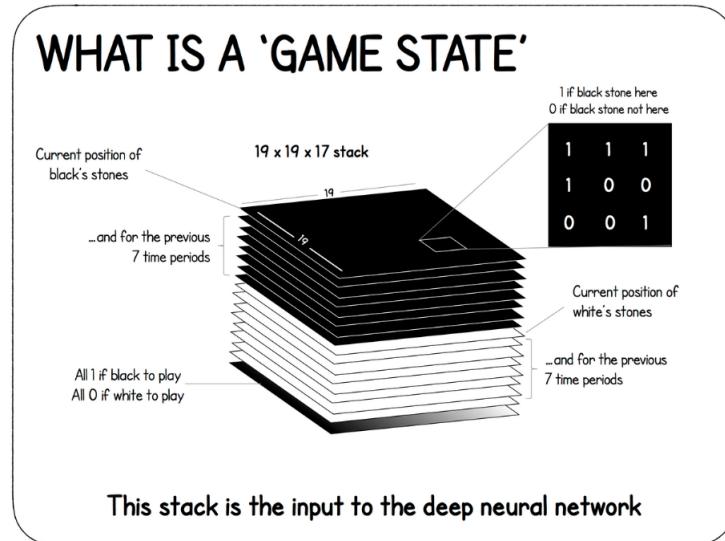


Figura 3.1: Representación visual de estado de juego de AlphaZero

Después, es importante definir la arquitectura de AlphaZero: una red con una capa convolucional inicial y 40 capas residuales consecutivas. La entrada será el estado del juego definido anteriormente, y la salida será una red de tácticas y de valores (ver figura 3.2).

La cabeza de valores de la red, que será descrita en profundidad en este capítulo, aplica al output la función *tanh* como función de activación, que distribuye los valores entre [-1,1]. Se encargará de evaluar los estados.

La cabeza de tácticas de la red, que también será descrita en profundidad en este capítulo, termina en una capa completamente conectada y se encargará de distribuir las probabilidades de los movimientos a realizar.

Posteriormente, la red será entrenada mediante la simulación de partidas comparando las predicciones hechas y los resultados obtenidos en las simulaciones.

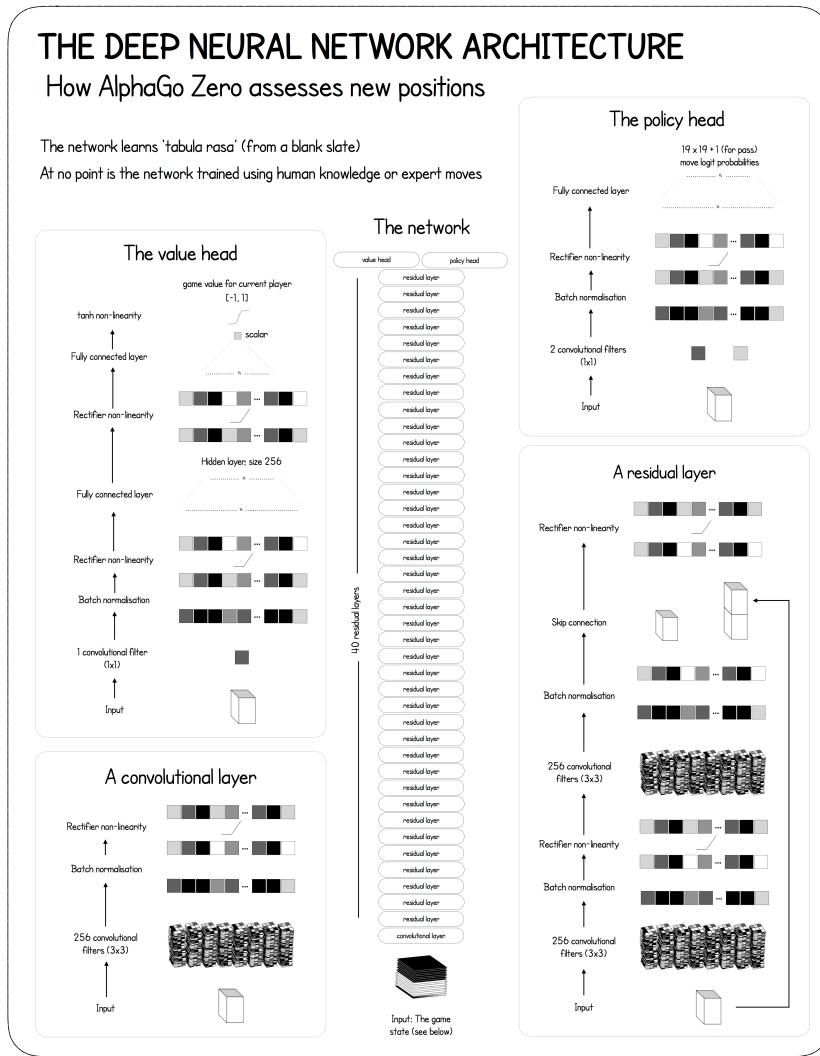


Figura 3.2: Representación visual de arquitectura de AlphaZero

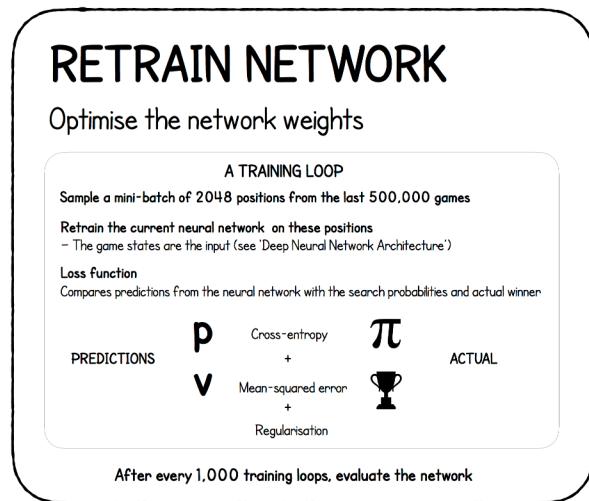


Figura 3.3: Representación visual de bucle de entrenamiento AlphaZero

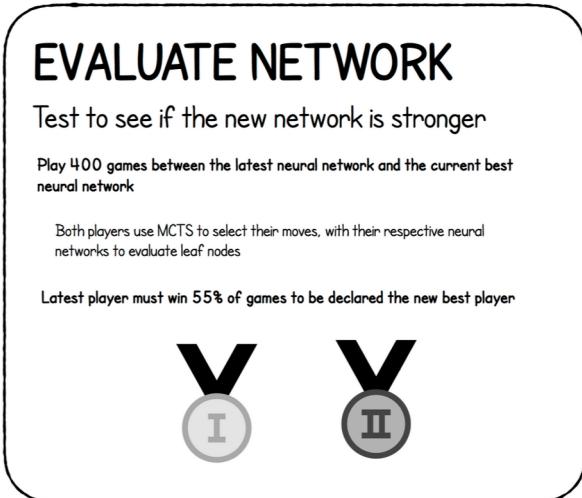


Figura 3.4: Representación visual de proceso de evaluación de AlphaZero

Mediante el uso de MCTS para la decisión de movimientos se realizarán evaluaciones periódicas para asegurar la mejora del algoritmo.

MCTS (ver figura 3.5) almacena para cada movimiento 4 variables: el número de veces que ha sido tomada cierta acción desde cierto estado, el valor total del próximo estado, el valor medio del próximo estado y la probabilidad previa a elegir la acción.

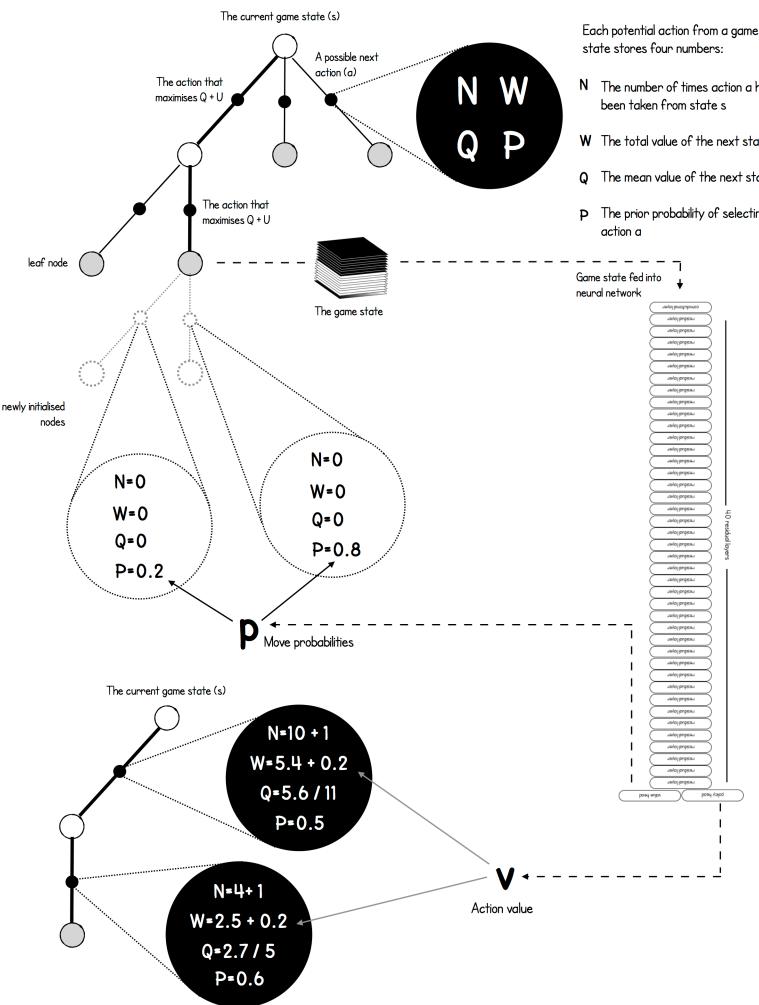
El proceso de elección de los movimientos viene dado por dos procesos, el primero en el que se realizan simulaciones de las acciones que maximicen la suma del valor medio del próximo estado más una función que devuelve un valor mayor según lo poco que ha sido explorado una acción respecto a las demás y cuan alta es la probabilidad de acción. Inicialmente, la función tendrá más peso para fomentar la exploración.

Posteriormente, el algoritmo continuará hasta que un nodo hoja sea alcanzado, entonces se pasará el estado de juego como input y devolverá mediante la red de valor y tácticas el valor estimado y las probabilidades de las acciones. Tras esto, se actualizan los nodos recorridos ajustando el número de veces que se ha tomado, el valor total, y el valor medio, y se fijan las probabilidades de las acciones al nodo hoja.

El segundo proceso, en el que se selecciona el movimiento como tal, se puede hacer de 2 formas: determinísticamente para juegos competitivos, en el que se toma la acción con mayor número de visitas (se interpreta que es el mejor), y estocásticamente, en el que se toma la acción desde el presente estado de la distribución  $\pi \sim N^{1/\tau}$ .

# MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move



## ...then select a move

After 1,600 simulations, the move can either be chosen:

Deterministically (for competitive play)

Choose the action from the current state with greatest  $N$

Stochastically (for exploratory play)

Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where  $\tau$  is a temperature parameter, controlling exploration

First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

### 1. Choose the action that maximises...

$$Q + U$$

The mean value of the next state

A function of  $P$  and  $N$  that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation,  $U$  dominates (more exploration), but later,  $Q$  is more important (less exploration)

### 2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

$p$ : Move probabilities

$V$ : Value of the state (for the current player)

The move probabilities  $p$  are attached to the new feasible actions from the leaf node

### 3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$$N \rightarrow N + 1$$

$$W \rightarrow W + v$$

$$Q = W / N$$

## Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

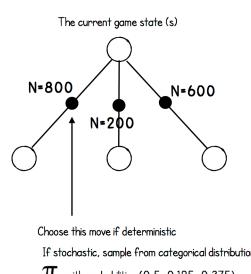


Figura 3.5: Representación visual del uso de MCTS en AlphaZero

## 3.2 Representación del Juego

Como se mencionó anteriormente, la representación del tablero [37] se realiza mediante una red neuronal convolucional, que normalmente se usa para reconocer patrones en imágenes que permiten reconocer objetos, clases o categorías.

En este caso el input de la red neuronal convolucional es un paquete de imágenes de tamaño  $N \times N \times (MT + L)$  que representan un estado usando una concatenación de  $T$  conjuntos de  $M$  planos de tamaño  $N \times N$ .

Cada conjunto de planos representa la posición del tablero en tiempo dado  $t - T + 1, \dots, t$ , y es fijado como 0 para tiempos menores a 1. El tablero está orientado de la misma forma que la perspectiva del jugador actual.

Los  $M$  planos de atributos están compuestos por planos de atributos binarios que indican la presencia de las fichas del jugador, con un plano para cada tipo de ficha, y un segundo conjunto de planos indicando la presencia de las fichas del oponente. En shogi hay planos adicionales indicando el número de prisioneros capturados de cada tipo. Además hay  $L$  planos adicionales que denotan el color del jugador, la cuenta de movimientos totales, y el estado de las reglas especiales, como los enroques cortos y largos, la cuenta de movimientos repetidos (3 movimientos repetidos es automáticamente un empate en el ajedrez, 4 en shogi), y el número de movimientos sin progreso (50 movimientos sin progreso es automáticamente un empate en el ajedrez).

Cada movimiento en el ajedrez puede ser descrito en dos partes, la primera que describe la selección de la ficha a mover, y la segunda, que describe el movimiento a realizar entre todos los movimientos válidos. Representamos la táctica  $\pi(a|s)$  por un conjunto de planos de tamaño  $8 \times 8 \times 73$  que codifican la distribución de probabilidad sobre 4672 movimientos posibles. Cada una de las  $8 \times 8$  posiciones identifica el tablero del que tomar una ficha. Por ejemplo, los primeros 56 planos codifican los posibles movimientos de la reina para cada ficha, el número de posiciones que avanzará, de 1 a 7, por las 8 direcciones relativas  $\{N, NE, E, SE, S, SW, W, NW\}$ . Los 8 planos siguientes codifican los movimientos posibles del caballo, y los últimos 9 planos codifican las posibles coronaciones de los peones o capturas en dos posibles diagonales a caballos, alfiles o torres.

Las tácticas en shogi son representadas por un conjunto de planos de tamaño  $9 \times 9 \times 139$  codificando similarmente una distribución de probabilidad sobre 11259 movimientos posibles. Los primeros 64 planos codifican movimientos de reina y los siguientes 2 movimientos codifican movimientos de caballo. Unos 64+2 planos adicionales codifican los movimientos con promoción de reina y los movimientos de promoción de caballos respectivamente. Los últimos 7 planos codifican una ficha capturada devuelta al tablero en esa posición específica.

Las tácticas en Go vienen dadas por el mismo procedimiento que en AlphaGo Zero, usando una distribución plana sobre  $19 \times 19 + 1$  movimientos representando posibles colocaciones de fichas

y el movimiento de paso.

Este último procedimiento también fue intentado para el ajedrez y el shogi, pero el resultado era prácticamente idéntico y el aprendizaje resultó ser levemente más lento.

Los movimientos ilegales son evitados fijando sus probabilidades a cero y re-normalizando las probabilidades para los movimientos restantes.

### 3.3 Redes de Tácticas y Valores

En la página 27 de [14] se describe el bloque de la red neuronal convolucional, que aplica los siguientes módulos:

1. Una convolución de 256 filtros de tamaño de núcleo  $3 \times 3$  con tamaño de paso 1.
2. Normalización por lotes.
3. ReLU.

Cada bloque residual aplica los siguientes módulos secuencialmente a su entrada:

1. Una convolución de 256 filtros de tamaño de núcleo  $3 \times 3$  con tamaño de paso 1.
2. Normalización por lotes.
3. ReLU.
4. Una convolución de 256 filtros de tamaño de núcleo  $3 \times 3$  con paso 1.
5. Normalización por lotes.
6. Una conexión omitida que añade el input al bloque.
7. ReLU.

El output de la torre residual es tratada posteriormente como dos módulos separados para computar la red de tácticas y la de valores separadamente.

La red de tácticas aplica los siguientes módulos:

1. Una convolución de 2 filtros de tamaño de núcleo  $1 \times 1$  con tamaño de paso 1.
2. Normalización por lotes.
3. ReLU.
4. Una capa completamente conectada que devuelve un vector de tamaño  $19^2 + 1 = 362$  que corresponde a las probabilidades de logits para todas las intersecciones y el movimiento de paso.

La red de valores aplica los siguientes módulos:

1. Una convolución de 2 filtros de tamaño de núcleo  $1 \times 1$  con tamaño de paso 1.
2. Normalización por lotes.
3. ReLU.
4. Una capa completamente conectada a una capa oculta de tamaño 256.
5. ReLU.
6. Una capa completamente conectada a un escalar.
7. Una función tanh que devuelve un escalar en el rango [-1,1].

La profundidad general de la red, en la red de 20 o 40 bloques, es de 39 o 79 capas parametrizadas respectivamente por la torre residual más 2 capas adicionales para la red de tácticas y 3 para la de valores.

En [38] se prueban distintas arquitecturas entre las que se encuentran

1. Redes de 12 bloques convolucionales separadas para la de tácticas y la de valores.
2. Una sola red de 12 bloques convolucionales tanto para tácticas como para valores.
3. Redes separadas de 20 bloques convolucionales.
4. Una sola red de 20 bloques convolucionales tanto para tácticas como para valores.
5. Una sola red de 40 bloques convolucionales tanto para tácticas como para valores.

Sin embargo en [37] AlphaZero gana 60 partidas en un encuentro de 100 contra la red única de 20 bloques de AlphaGo, pero no está explicitamente enunciada la arquitectura que usa AlphaZero, probablemente usa la red de 40 bloques, que es la que mejor rendimiento mostró, o la red de 20 bloques única para hacer una comparación justa.

## 3.4 MCTS en AlphaZero

El uso de MCTS es exactamente igual al descrito en capítulos anteriores pero con ciertas particularidades, entre ellas se encuentran las siguientes:

1. Existen dos cabezas de red, la de tácticas y la de valores que son mejoradas mediante partidas jugadas solas y que permiten la evaluación de nodos para mejorar el tiempo de procesamiento.
2. Durante el MCTS en AlphaZero no se realizan simulaciones, simplemente se evalúan los nodos mediante una red neuronal residual de 19 capas.

$$V(S_L) = F_0(S_L)$$

### 3.5 Rendición Temprana

En los distintos papers de AlphaZero y AlphaGo se mencionan métodos de rendición temprana durante el aprendizaje para optimizar el proceso, pero no son muy descriptivos.

El único explicado es el siguiente: se define una variable real,  $v_{resign}$ , seleccionada automáticamente, de forma que la relación de falsos positivos (partidas que habrían sido ganadas si AlphaZero no se hubiese rendido) sea menor que un 5 %. Para medir los falsos positivos se desactiva la opción de rendición temprana en un 10 % de los juegos y se finalizan completamente.

### 3.6 Configuración

Durante el aprendizaje de AlphaZero, cada MCTS usó 800 simulaciones. El número de partidas, posiciones, y tiempo de procesamiento varió ampliamente según el juego debido a los tamaños de los tableros y la duración de las partidas. La tasa de aprendizaje fue de 0.2 para cada juego, y fue reducida 3 veces (a 0.02, 0.002 y 0.0002, respectivamente) durante el transcurso del aprendizaje.

Los movimientos fueron seleccionados en proporción al contador de visitas de cada nodo. Se añadió ruido de Dirichlet ( $Dir(\alpha)$ ) a las probabilidades previas en el nodo raíz, esto fue ajustado en proporción inversa al número aproximado de movimientos legales en una posición típica, a un valor  $\alpha = \{0.03, 0.15, 0.03\}$  para el ajedrez, shogi y go respectivamente.

	Ajedrez	Shogi	Go
<b>Mini-batches</b>	700 mil	700 mil	700 mil
<b>Tiempo de aprendizaje</b>	9h	12h	34h
<b>Partidas</b>	44 millones	24 millones	21 millones
<b>Tiempo de procesado</b>	40 ms	80 ms	200 ms

Cuadro 3.1: Datos de aprendizaje para ajedrez, shogi y go en 800 simulaciones

### 3.7 Evaluación

Para evaluar el desempeño en ajedrez, se usó Stockfish versión 8 como programa base, usando 64 hilos de CPU y un tamaño de hash de 1GB.

Para evaluar el funcionamiento en shogi, se usó Elmo versión WCSC27 junto a YaneuraOu 2017 Early KPPT 4.73 64AVX2 con 64 hilos de CPU y un tamaño de hash de 1GB con la opción USI de EnteringKingRule en NoEnteringKing.

Se evaluó la fuerza relativa de AlphaZero midiendo el ELO de cada jugador. Se estimó la probabilidad del jugador  $a$  de vencer al jugador  $b$  mediante una función logística:

$$p(a \text{ vence a } b) = \frac{1}{1 + \exp(c_{elo}(e(b) - e(a)))},$$

y se estimaron los evaluadores  $e(\cdot)$  con una regresión logística Bayesiana, computada por el programa BayesElo usando la constante estándar  $c_{elo} = \frac{1}{400}$ .

Los ELOs fueron computados mediante los resultados de los torneos de un movimiento por segundo entre iteraciones de AlphaZero durante su aprendizaje, y también un jugador base: cualquiera entre Stockfish, Elmo o AlphaGo Lee, respectivamente.



# 4 | Implementación

En esta sección vamos a describir la implementación de un juego llamado "Dots and boxes" en Python con pequeñas modificaciones que permitirán evitar tácticas ganadoras, el trabajo original hecho por Johannes Scherer puede encontrarse en el siguiente link: [josch14](#).

Aquí describiremos las funciones y clases más relevantes y que más se relacionan con los conceptos mencionados durante el proyecto, las implementaciones adicionales al trabajo inicial también serán añadidas en la memoria.

Empecemos definiendo las reglas del juego para saber de qué se trata.

## 4.1 Reglas del Juego

*Dots and boxes* es un juego que empieza con  $N \times N$  puntos describiendo una cuadrícula de  $(N - 1) \times (N - 1)$  cuadrados, todos ellos sin conectar. En cada turno, un jugador dibuja una línea entre dos puntos adyacentes (vertical, u horizontalmente) y pasa al turno del otro jugador, salvo si la línea dibujada cierra un cuadrado, en cuyo caso el jugador repetirá turno y sumará un punto. El ganador será el que consiga más puntos, parándose el juego en caso de superar la mitad.

Este juego tiene una estrategia ganadora que rápidamente podría hacer que nuestro programa repitiese la misma estrategia una y otra vez, una breve explicación puede consultarse en [How to always win at Dots and Boxes - Numberphile](#). Por esto implementaremos una cuadrícula de tamaño  $(N - 1) \times (N - 1)$  que le dará un valor aleatorio a cada cuadrado (cada jugador lo sabrá desde el inicio), para así evitar partidas repetitivas. De esta forma una partida podría decidirse en la toma de una sola casilla.

### 4.1.1 Implementación

Implementaremos una clase llamada **DotsAndBoxesGame** que se encargará de todo lo relacionado con las reglas y la jugabilidad, de esta destacaremos los atributos de la clase, y algunos métodos relevantes.

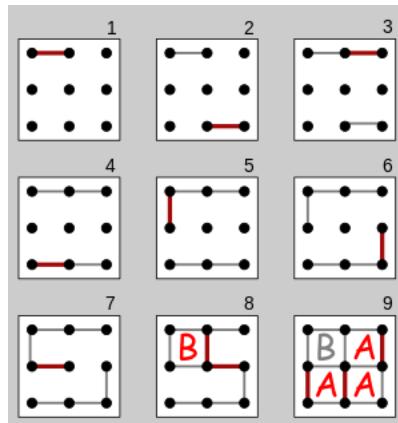


Figura 4.1: Representación gráfica de partida de Dots and Boxes

Los atributos de la clase serán los siguientes:

- Un entero que describirá el número de cuadrados que hay por fila o columna, al que llamaremos SIZE.
- Un booleano que señale de qué jugador es el turno, al que llamaremos current\_player.
- Un entero que sirva de contador para el resultado al que llamaremos result.
- Un entero que describa el número de líneas totales al que llamaremos N\_LINES.
- Un vector de tamaño el número de líneas totales, en el que cada componente valdrá -1,0,1 según si la línea ha sido tomada por un jugador u otro, o si no ha sido tomada aún, al que llamaremos l.
- Un entero que describa el número de cajas totales, al que llamaremos N\_BOXES.
- Un vector de tamaño el número de cajas totales, en el que cada componente valdrá -1,0,1 según si la caja ha sido tomada por un jugador u otro, o si no ha sido tomada aún, al que llamaremos b.
- Un vector aleatorio de tamaño el número de cajas totales, en el que cada componente marcará cuánto vale cada caja, al que llamaremos r, esto nos permitirá diferenciarnos del juego original y sortear el problema de estrategias ganadoras en partidas con tablero pequeños.

Los únicos atributos que han de ser definidos al inicializarlo son el tamaño y el jugador que empieza.

Definiremos un método que se ocupará de dibujar las líneas:

```
def draw_line(self, line: int):
    assert self.l[line] == 0, "Línea ya dibujada"
    self.l[line] = self.current_player
```

Otro método para cambiar de jugador en cada final de turno:

```
def switch_current_player(self):
    self.current_player *= -1
```

Definiremos un método que se encargará de los movimientos usando el método de draw\_line, y comprobará si una caja ha sido capturada o no para decidir el jugador que empezará el próximo turno:

```
def execute_move(self, line: int):

    self.draw_line(line)

    box_captured = False
    for box in self.get_boxes_of_line(line):
        lines = self.get_lines_of_box(box)

        if len([self.l[line] for line in lines if self.l[line] != 0]) == 4:
            self.capture_box(
                row=box[0],
                col=box[1]
            )
            box_captured = True

    if not box_captured:
        self.switch_current_player()
    else:
        self.check_finished()
```

Necesitaremos un método auxiliar que nos devuelva las cajas en contacto con una línea:

```
def get_boxes_of_line(self, line: int) -> List[Tuple[int, int]]:

    if line < int(self.N_LINES / 2):
        #Lineas horizontales
        i = line // self.SIZE
        j = line % self.SIZE

        if i == 0:
            return [(i, j)]
```

```

        elif i == self.SIZE:
            return [(i - 1, j)]
        else:
            return [(i - 1, j), (i, j)]

    else:
        #Linea vertical
        line = line - int(self.N_LINES / 2)
        j = line // self.SIZE
        i = line % self.SIZE

        if j == 0:
            return [(i, j)]
        elif j == self.SIZE:
            return [(i, j - 1)]
        else:
            return [(i, j - 1), (i, j)]

```

Y un equivalente, que devuelva las líneas en contacto con una caja:

```

def get_lines_of_box(self, box: Tuple[int, int]) -> List[int]:
    i = box[0]
    j = box[1]

    #Lineas horizontales
    line_top = i * self.SIZE + j
    line_bottom = (i + 1) * self.SIZE + j

    #Lineas verticales
    line_left = int(self.N_LINES / 2) + j * self.SIZE + i
    line_right = int(self.N_LINES / 2) + (j + 1) * self.SIZE + i

    return [line_top, line_bottom, line_left, line_right]

```

Finalmente, crearemos un método con una lista de los movimientos válidos. Aquí usaremos el atributo l, que se encarga de almacenar las líneas capturadas o libres, y en este caso buscamos las que están libres:

```

def get_valid_moves(self) -> List[int]:
    return np.where(self.l == 0)[0].tolist()

```

Hay más métodos que se usan para mejorar el funcionamiento y disminuir la potencia computacional necesaria aprovechando las simetrías del juego, entre otras, pero no son imprescindibles.

## 4.2 Modelos

La implementación de este juego es exactamente como la de AlphaZero pero con una diferencia, al ser Dots And Boxes un juego tan simple, puede ser representado por un vector, como hemos hecho. Por ello, en nuestro caso el uso de una red neuronal convolucional no tiene sentido y en su lugar se ha optado por una simple red de capas completamente conectadas, con una inicialización aleatoria.

Entre los modelos implementados se encuentran uno que describe la red residual, otro que describe la red feed forward que describimos anteriormente (la capa completamente conectada) y una red neuronal.

### 4.2.1 Implementación

Para la implementación de los modelos podemos encontrar webs que describen la creación de las redes de valores y tácticas y guían el proceso de entrenamiento, [33]. Aquí implementaremos las siguientes clases; **AZNeuralNetwork**, **AZFeedForward** y **AZDualRes**:

La implementación de la red neuronal AlphaZero  $f(s) = (p, v)$ , combinando el uso de una red de tácticas y otra de valores,

- **p** (vector de tácticas): vector de probabilidades de movimiento  $p = P(a|s)$ ,
- **v** (valor escalar): probabilidad del jugador de ganar desde la posición  $s$ .

```
class AZNeuralNetwork(ABC, nn.Module):

    def __init__(self, game_size: int, inference_device: torch.device):
        super(AZNeuralNetwork, self).__init__()

        self.inference_device = inference_device
        self.game_size = game_size
```

Definiremos el método **forward**, que avanza a través de la red neuronal. Posteriormente definiremos un método más complejo, este; sin embargo, solo será usado durante el aprendizaje de la

red neuronal. Durante la etapa de auto-juego usando MCTS y la evaluación del modelo, **p\_v** será usado, que usa a su vez este modelo.

Los parámetros usados serán los siguientes:

- **x** : torch.Tensor | estado de juego codificado, supuestamente en su forma canónica

y devolverá:

- **p, v** : [torch.Tensor, torch.Tensor] | vector de tácticas **p** (potencialmente conteniendo valores >0 para movimientos no válidos), valor **v**

```
def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    pass
```

Aquí definiremos el método **p\_v** mencionado anteriormente, este avanza una vez a través de la red neuronal. A diferencia del método **forward()**, este método se asegura de que **p** tenga probabilidad 0 en los movimiento no válidos (asegurando a su vez la distribución de probabilidad en **p**).

Parámetros:

- **l** : np.ndarray | vector de líneas, supuestamente en su forma canónica
- **b** : np.ndarray | matriz de cajas, supuestamente en su forma canónica

y devolverá:

- **p, v** : [np.ndarray, float] | vector de tácticas **p** (conteniendo valores  $\geq 0$  solo para movimientos validos), valor **v**

En **p\_v** calcularemos los movimientos válidos asegurándonos de que hay alguno, tras ello, **x** codifica **l** y **b**, lo convertimos en tensor y luego le añadimos una dimensión extra.

Con **torch.no\_grad()** nos aseguramos de que no se calcularán los gradientes y, por lo tanto, no se actualizarán los coeficientes mediante retropropagación. Tras esto, avanza en la red y almacena **p** y **v**.

Aquí  $p$  contendrá valores  $p > 0$  para movimientos no válidos y, por ello, calculamos una lista de ceros para sustituir por 1 solo las posiciones que corresponden a movimientos válidos, de esta forma, al multiplicar esta lista por  $p$  obtenemos nuestro  $p$  con valores nulos para movimientos no válidos.

Finalmente, comprobamos que el modelo devolvió al menos un movimiento válido y nos encargamos de distribuir la probabilidad equivalentemente entre todos los movimientos válidos para después normalizarlos y poder devolver  $p$  y  $v$ .

```
def p_v(self, l: np.ndarray, b: np.ndarray) -> Tuple[np.ndarray, float]:
    valid_moves = np.where(l == 0)[0].tolist()
    assert len(valid_moves) > 0, "No hay movimientos válidos, el modelo" +
        "no debería ser llamado en este caso"

    #El modelo requiere
    x = self.encode(l, b) # ... funcion para codificar
    x = torch.from_numpy(x).to(self.inference_device) # ... tensor
    x = x.unsqueeze(0) # ... lote, por la normalización por lotes

    # cpu solo necesario cuando gpu es usada
    with torch.no_grad():
        p, v = self.forward(x)
    p = p.squeeze().detach().cpu().numpy()
    v = v.detach().cpu().item()

    # p posiblemente contiene p > 0 para movimientos no válidos -> borrar
    valid = np.zeros(l.squeeze().shape, dtype=np.float32)
    valid[valid_moves] = 1

    p_valid = np.multiply(p, valid)
    if np.sum(p_valid) == 0:
        logging.warning(f"El modelo no devolvió una probabilidad > 0" +
            "para ningún movimiento válido:\n" +
            f"(p,v) = {(p, v)} con movimientos válidos {valid_moves}.")
        # Distribuye la probabilidad equivalentemente para todos
        #los movimientos válidos
    p_valid = np.multiply([1] * l.shape[0], valid)

    #Normaliza
    p = p_valid / np.sum(p_valid)
```

```
    return p, v
```

Como la posición de un juego en Dots and Boxes es representada por un vector (y no es representable por una imagen), el uso de capas convolucionales no tiene sentido. Por ello, el modelo hace uso de simples capas completamente conectadas, que son inicializadas con coeficientes aleatorios.

Mostraremos tan solo la clase de **AZFeedForward**, el resto son equivalentes pero haciendo uso de las capas que hemos mencionado en capítulos anteriores.

Primero definimos el número de líneas y cajas que habrá y tomamos los parámetros del modelo.

Posteriormente definiremos las capas ocultas, para ello conectaremos mediante **nn.Sequential**, que nos permite crear una pequeña secuencia de capas, con un **nn.linear** inicialmente (que aplica una transformación lineal y suele ser usada en redes feed forward, a diferencia de **nn.Conv2d**, que aplica una convolución 2D y suele ser usada en redes convolucionales).

Los parámetros de entrada del primer **nn.linear** serán el número de líneas y cajas, a partir de él serán el número de salidas e la capa anterior, por lo que, obviamente, el parámetro de salida será determinado por la componente i-ésima de **hidden\_layers**.

Tras esto se normaliza por lotes y se aplica una *ReLU*, creando así una lista de capas completamente conectadas con normalización por lotes y función de activación *ReLU* para después agruparlas con **nn.ModuleList**.

Finalmente definimos las capas de salida, **p** y **v**, que toman la última capa oculta; la primera tiene un output de tamaño **n\_lines** que pasa por un softmax de dimensión 1, y la segunda tiene un output de tamaño 1 que pasa por una tangente hiperbólica. Tras esto simplemente se inicializan los coeficientes con una función que para cada capa y **nn.Linear**, usa una inicialización Xavier uniforme y un bias.

```
class AZFeedForward(AZNeuralNetwork):

    def __init__(
        self,
        game_size: int,
        inference_device: torch.device,
        model_parameters: dict):

        super(AZFeedForward, self).__init__(game_size, inference_device)

        n_lines = 2 * self.game_size * (self.game_size + 1)
        n_boxes = 2 * self.game_size
```

```

#Haremos uso de la información de parámetros de config
hidden_layers = model_parameters["hidden_layers"]

#Capas ocultas
self.fully_connected_layers = []
for i in range(len(hidden_layers)):
    fc_layer = nn.Sequential(
        nn.Linear(
            in_features=(n_lines + n_boxes if i == 0 else
            hidden_layers[i - 1]),
            out_features=hidden_layers[i]
        ),
        nn.BatchNorm1d(hidden_layers[i]),
        nn.ReLU(),
    )
    self.fully_connected_layers.append(fc_layer)

self.fully_connected_layers = nn.ModuleList(self.fully_connected_layers)

#Capas de salida
self.p_head = nn.Sequential(
    nn.Linear(hidden_layers[-1], n_lines),
    nn.Softmax(dim=1),
)

self.v_head = nn.Sequential(
    nn.Linear(hidden_layers[-1], 1),
    nn.Tanh(),
)

#Inicializa coeficientes
self.weight_init()

def weight_init(self):
    """Inicializa capas completamente conectadas y ambos outputs"""
    for layer in self.fully_connected_layers + [self.p_head, self.v_head]:
        for linear in [m for m in layer if isinstance(m, nn.Linear)]:
            nn.init.xavier_normal_(linear.weight) #Inicialización de coef
            linear.bias.data.fill_(0.01) #Inicialización bias

```

Por último aquí definiremos la clase **AZDualRes** con los distintos bloques y métodos, pueden consultarse en josch14. De esta solo mencionaremos que se construye de manera equivalente a la anterior, y los bloques convoluciones y residuales, que sí que pueden ser de interés, además de la función encode.

Esta función genera 4 imágenes, todas vacías, la primera contiene las líneas horizontales y verticales, las horizontales se asignan con las filas pares y columnas impares, y las verticales con las filas impares y columnas pares, todo sin tener en cuenta por quién han sido capturadas.

La segunda imagen contiene las cajas capturadas por el jugador 1 en las posiciones con filas y columnas impares (en las que  $b = 1$ ).

La tercera hace lo mismo pero con el jugador 2 (en las que  $b = -1$ ), aunque sigue siendo indicado por 1 en vez de -1.

La cuarta tan solo rellena esquinas, que no son ni cajas ni líneas, por lo que no son relevantes.

Y finalmente se apilan en una sola matriz de 4 canales.

Para nuestra implementación, como hemos sustituido los valores de b para que no siempre todas las cajas valgan 1, la función encode podría variar, se pueden hacer de diversas formas, la primera, creando una imagen más en las que las posiciones que indican las distintas cajas (columnas y filas impares), valiesen el número seleccionado aleatoriamente para esa caja (vector r) en el rango 0 a 255 para una imagen en escala de grises, o podría hacerse incluso en esas 4 imágenes. Sería fijar el punto medio entre 0 y 255 y fijarlo como caja no capturada, y normalizar todos los valores siendo el -1 en r fijado como 0 y el 1 como 255.

De hecho, sería interesante hacerlo de tal forma que la equivalencia no fuera lineal, si no exponencial, por ejemplo, ya que realmente una caja de 0.01 de valor no serviría prácticamente para nada (el algoritmo al iterar sería capaz de "aprender" a buscar valores en los extremos así que tal vez no serviría para nada o podría acelerar el proceso).

```
def encode(l: np.ndarray, b: np.ndarray) -> np.ndarray:
    """Codifica líneas y cajas en imágenes"""

    game_size = DotsAndBoxesGame.n_lines_to_size(l.size)
    img_size = 2 * game_size + 1

    img_l = np.zeros((img_size, img_size), dtype=np.float32)
    img_b_player = np.zeros((img_size, img_size), dtype=np.float32)
    img_b_opponent = np.zeros((img_size, img_size), dtype=np.float32)
    img_background = np.zeros((img_size, img_size), dtype=np.float32)

    # 1) La imagen contiene información sobre las líneas dibujadas
```

```

# para la predicción de la red de tácticas
h, v = DotsAndBoxesGame.l_to_h_v(l)
#Horizontales: filas pares, columnas impares
#Verticales: filas impares, columnas pares
img_l[::2, 1::2] = h
img_l[1::2, ::2] = v
img_l[img_l == -1.0] = 1.0

# 2) Imagen indicando las cajas capturadas por el jugador (para
#predicción del valor)
img_b_player[1::2, 1::2] = b
img_b_player[img_b_player == -1.0] = 0.0

# 3) Imagen indicando cajas capturadas por el oponente (para
#predicción del valor)
img_b_opponent[1::2, 1::2] = b
img_b_opponent[img_b_opponent == 1.0] = 0.0
img_b_opponent[img_b_opponent == -1.0] = 1.0

# 4) Imagen indicando pixeles no importantes
img_background[0::2, 0::2] = np.ones((game_size+1, game_size+1),
dtype=np.float32)

feature_planes = np.stack([img_l, img_b_player, img_b_opponent,
img_background], axis=0)
return feature_planes

```

Son de especial interés los siguientes bloques ya que son los que definen la red convolucional residual que estamos implementando.

Lo primero en lo que nos podemos fijar es que el bloque convolucional es implementado en solitario, mientras que el residual está implementado dentro de un **nn.ModuleList** uniendo varios bloques residuales, esto es debido a que el bloque residual se construye en serie, dándole profundidad a la red, de ahí el bucle for, aunque normalmente no suelen ser más que 2 o 3.

También podemos ver la clase de **ConvBlock** y la de **ResBlock**, en la que para el primero vemos las capas de convolución 2D, la normalización por lotes 2D y una capa final *ReLU* junto con su método forward que simplemente atraviesa las capas anteriormente mencionadas en ese orden, y también podemos ver la clase de **ResBlock**, para las que hay una capa convolucional 2D, una normalización por lotes 2D y una *ReLU* y se repite, como si fuesen dos bloques convolucionales seguidos, aunque podemos ver en el método forward que al hacer la segunda *ReLU* se le suma el valor inicial sin pasar por el primer bloque convolucional.

```

#Bloque convolucional
self.conv_block = ConvBlock(
    out_channels=channels,
    kernel_size=conv_kernel_size,
    stride=stride,
    padding=padding
)

#Bloque residual
self.residual_blocks = nn.ModuleList(
    [ResBlock(
        n_channels=channels,
        kernel_size=res_kernel_size,
        stride=stride,
        padding=padding
    ) for _ in range(residual_blocks)]
)

```

```

class ConvBlock(nn.Module):

    def __init__(self, out_channels, kernel_size, stride, padding):
        super(ConvBlock, self).__init__()

        self.conv = nn.Conv2d(4, out_channels, kernel_size, stride, padding)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.bn(self.conv(x)))

    return x

```

```

class ResBlock(nn.Module):

    def __init__(self, n_channels, kernel_size, stride, padding):
        super(ResBlock, self).__init__()

        self.conv1 = nn.Conv2d(n_channels, n_channels, kernel_size, stride, padding)
        self.bn1 = nn.BatchNorm2d(n_channels)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(n_channels, n_channels, kernel_size, stride, padding)

```

```

    self.bn2 = nn.BatchNorm2d(n_channels)
    self.relu2 = nn.ReLU()

def forward(self, x):
    x_in = x
    x = self.relu1(self.bn1(self.conv1(x)))
    x = self.bn2(self.conv2(x))
    x += x_in
    x = self.relu2(x)

    return x

```

## 4.3 MCTS

El método de búsqueda en árboles Montecarlo es exactamente el descrito en capítulos anteriores. Una pequeña guía de su implementación puede ser consultada en [43] y con más profundidad en el capítulo 2.3 de [11] y [21], en la que se implementarán dos de los procesos que posteriormente definiremos. De su implementación se encargarán la clase **MCTS** y la clase **AZNode**, de la que presentaremos los métodos más importantes, principalmente los encargados del movimiento en sí y no tanto los atributos:

La clase **AZNode** implementa la búsqueda por árbol de AlphaZero. Cada nodo corresponde a la posición **s**.

Atributos:

- **s** : DotsAndBoxesGame | posición correspondiente al nodo, incluye el vector de posición s
- **a** : int | movimiento ejecutado desde la posición del nodo padre, resultando en la posición s del nodo
- **children** : [AZNode] | nodos hijos
- **Q** : dict | valores de las acciones  $Q[a] = Q(s,a)$
- **N** : dict | contador de visitas  $N[a] = N(s,a)$
- **P** : np.ndarray | valores de acción  $P(s,a)$  como los devueltos por la red neuronal

```

class AZNode:

    def __init__(self, parent, s: DotsAndBoxesGame, a: int):

```

```

if parent is not None: #Solo el nodo raíz no tiene padre
    assert isinstance(parent, AZNode)
    for child in parent.children: #El nodo no debería existir aún
        #entre los hijos del padre
        assert child.s != s and child.a != a

#todo nodo excepto el raíz de un árbol de búsqueda completa de MCTS
#ha de tener un movimiento correspondiente
assert (parent is None and a is None) or
    (parent is not None and isinstance(a, int))

#Crea enlace con nodo padre
if parent is not None:
    parent.children.append(self)

#Usa parámetros del constructor
self.a = a
self.s = s

self.children = []
self.Q = {}
self.N = {}
self.P = None

def get_child_by_move(self, a: int):
    for child in self.children:
        if child.a == a:
            return child

```

En la clase MCTS el método Montecarlo es ejecutado, guiado por la red neuronal, en cada posición s durante el auto-juego para determinar el siguiente movimiento.

Atributos:

- **model** : AZNeuralNetwork | red neuronal para evaluar las posiciones del tablero
- **root** : AZNode | nodo desde el que el MCTS es ejecutado (posición del input s)
- **n\_simulations** : int | simulaciones para cada MCTS (para determinar el próximo movimiento)
- **c\_puct** : float | constante determinando el nivel de exploración
- **dirichlet\_eps** : float | peso del ruido de Dirichlet en nodo raíz de una simulación
- **dirichlet\_alpha** : float | parámetro de distribución para el ruido de Dirichlet

*play* será el método definido que proveerá la funcionalidad del MCTS.

Parámetros:

- **temp** : int | parámetro que describe el control de temperatura

Devuelve:

- **probs** : [float] | probabilidades de movimiento  $\pi(a) \sim N(s,a)^{(1/temp)}$

```
def play(self, temp: int) -> [float]:  
  
    s = self.root.s # posición s del nodo raíz  
    valid_moves = self.root.s.get_valid_moves()  
  
    #Realiza simulaciones del MCTS  
    for i in range(self.n_simulations):  
        #Añade ruido de Dirichlet solo a los movimientos válidos, añadido después  
        #de las probabilidades previas del nodo raíz  
        dirichlet_noise = np.zeros((s.N_LINES,), dtype=np.float32)  
        dirichlet_noise[valid_moves] = np.random.dirichlet([self.dirichlet_alpha] *  
            len(valid_moves))  
        self.search(self.root, is_root=True, dirichlet_noise=dirichlet_noise)  
  
    #Solo movimientos válidos pueden ser visitados  
    assert set(list(self.root.N.keys())).issubset(set(s.get_valid_moves()))  
  
    #El vector de probabilidad devuelto debe tener un valor para cada línea  
    #Cuando la línea está dibujada la probabilidad debería ser de 0  
    counts = [self.root.N[a] if a in self.root.N else 0 for a in range(s.N_LINES)]  
  
    if temp == 0:  
        #Selecciona el movimiento con el máximo número de visitas para  
        #realizar la mejor jugada posible  
        probs = [0] * len(counts)  
        probs[np.array(counts).argmax()] = 1  
        return probs  
  
    #  $\pi(a) \sim N(s,a)^{(1/temp)}$  mientras asegura una distribución de probabilidad  
    probs = [n ** (1. / temp) for n in counts]
```

```

probs = [p / float(sum(probs)) for p in probs]
return probs

```

*search* será el método definido para ejecutar una simulación por el MCTS de manera recursiva

Parámetros:

- **node** : AZNode | nodo que corresponde a la posición actual s del MCTS
- **is\_root** : bool | si el nodo actual es el nodo raíz de la búsqueda o no
- **dirichlet\_noise** : bool | ruido de Dirichlet que es aplicado solo a las probabilidades previas del nodo raíz

Devuelve:

- **v** : float | probabilidad del jugador actual de ganar desde la posición s

```

def search(self, node: AZNode, is_root: bool = False,
          dirichlet_noise: np.ndarray = None) -> float:

    if not node.s.is_running():
        #El juego termina antes de llegar a un nodo no visitado
        #Devuelve el resultado actual v para el jugador actual
        #En caso de que haya un ganador, lo sería el jugador actual
        #ya que si hubiese sido por toma de caja, el jugador actual no cambia
        result = node.s.result

        if node.s.current_player == result:
            v = 1
        elif result == 0:
            v = 0
        else:
            v = -1
        return v

    #Se alcanzó una hoja (nodo no visitado aún)
    if node.P is None:
        v = self.evaluate(node)
        return v

    #El nodo fue visitado anteriormente, continua atravesando el árbol

```

```

a = self.select(node, is_root, dirichlet_noise)

if a not in node.N:
    #Aplicar el movimiento seleccionado significa realizar un m
    #movimiento hacia una hoja (nodo sin visitar aun)
    child = self.expand(node, a)
else:
    child = node.get_child_by_move(a)

#continua atravesando, llamando al método recursivamente
v_child = self.search(child)

#Recibimos una puntuación v del nodo hijo, bien por alcanzar una
#hoja (v en [-1,1] calculado por la red neuronal) o bien por
#terminar el juego (v en {-1,0,1})
v = v_child if node.s.current_player == child.s.current_player else -v_child

#método backup antes de devolver v
self.backup(node, a, v)

return v

```

El método *select* selecciona el movimiento con máximo valor de acción  $Q$ , más un límite superior de confianza que depende de la probabilidad previa almacenada  $P$  y del contador de visitas

Parámetros:

- **node** : AZNode | nodo no-hoja correspondiente a la posición actual del MCTS s
- **is\_root** : bool | si el nodo actual es el nodo raíz de la búsqueda o no
- **dirichlet\_noise** : bool | ruido de Dirichlet que es aplicado solo a las probabilidades previas del nodo raíz

Devuelve:

- **a\_max** : int | movimiento a para el que  $Q(s,a) + U(s,a)$  es máximo

```

def select(self, node: AZNode, is_root: bool, dirichlet_noise: np.ndarray) -> int:

    assert len(node.s.get_valid_moves()) > 0

```

```

maximum = float('-inf')
a_max = -1

N_sum = sum(node.N.values())
N_sqrt = math.sqrt(N_sum)

P = node.P if not is_root else \
    (1 - self.dirichlet_eps) * node.P + self.dirichlet_eps * dirichlet_noise
assert abs(np.sum(P) - 1) < 1e-6, \
f"Es raiz: {is_root}, suma de P: {np.sum(node.P)}, suma de P tras" + \
    "añadir el ruido de Dirichlet: {np.sum(P)}"

for a in node.s.get_valid_moves():
    #cada movimiento se corresponde con un nodo hijo que puede
    #o no haber sido ya visitado

    p = P[a]
    if a in node.N:
        q = node.Q[a]
        n = node.N[a]
    else:
        q = 0
        n = 0

    #Límite superior de confianza  $U(s, a) \sim P(s, a) / (1 + N(s, a))$ 
    u = self.c_puct * p * N_sqrt / (1 + n)

    #Maximiza el valor de acción  $Q(s, a) + \text{límite superior de confianza } U(s, a)$ 
    if q + u > maximum:
        maximum = q + u
        a_max = a

return a_max

```

El método *expand*, para el nodo de entrada, crea el hijo del nodo que es alcanzado al ejecutar el movimiento a.

Parámetros:

- **node** : AZNode | nodo que corresponde a la posición actual del MCTS s
- **a** : int | movimiento con el que la hoja es alcanzada desde el nodo actual

Devuelve:

- **leaf** : AZNode | el nodo hijo/hoja creado

```
def expand(self, node: AZNode, a: int):

    s = copy.deepcopy(node.s)
    s.execute_move(a)
    leaf = AZNode(
        parent=node,
        a=a,
        s=s
    )
    return leaf
```

El método *evaluate* evalúa la posición asociada del nodo hoja por la red neuronal y almacena el vector P de valores. Primero obtiene el estado canónico de la situación del juego, invariante a trasposiciones desde la perspectiva del jugador actual. Tras ello aplica una rotación o reflejo aleatorio (utiliza  $i, j = 0, \dots, 7$  para referirse a todas estas transformaciones). Tras ello, evalúa y ajusta el vector de probabilidades aplicando la transformación de manera inversa.

Parámetros:

- **leaf** : AZNode | nodo hoja que corresponde a la posición actual del MCTS

Devuelve:

- **v** : float | probabilidad del jugador actual de ganar en la posición s

```
def evaluate(self, leaf: AZNode) -> float:

    #Reglas son invariantes a las trasposiciones, representa el tablero
    #desde la perspectiva del jugador actual
    canonical_lines = leaf.s.get_canonical_lines()
    canonical_boxes = leaf.s.get_canonical_boxes()

    #Evaluación de la red neuronal es llevada a cabo con reflejos y rotaciones
    #que son seleccionados uniformemente
    i = randint(0, 7)
```

```

j = i
if i == 1:
    j = 3
elif i == 3:
    j = 1

lines = DotsAndBoxesGame.get_rotations_and_reflections_lines(
canonical_lines)[i]
boxes = DotsAndBoxesGame.get_rotations_and_reflections_boxes(
canonical_boxes)[i]

#Toma la predicción y aplica la misma rotación revertida que fue aplicada
#al vector de lineas antes de ser enviado a la red neuronal
p, v = self.model.p_v(lines, boxes)
p = DotsAndBoxesGame.get_rotations_and_reflections_lines(p)[j]

leaf.P = p
return v

```

El método *backup* actualiza el valor de acción Q para calcular la media de todas las evaluaciones v en el sub-árbol tras ese nodo, y el contador de visitas.

Parámetros:

- **node** : AZNode | nodo que corresponde a la posición actual del MCTS s
- **a** : int | movimiento que fue seleccionado y ejecutado dentro de la actual búsqueda
- **v** : float | puntuación resultante de este nodo para esta simulación actual

```

def backup(self, node: AZNode, a: int, v: float):

    if a not in node.N:
        # hoja : el nodo fue visitado por primera vez
        node.Q[a] = v
        node.N[a] = 1

    else:
        n = node.N[a]
        q = node.Q[a]
        node.Q[a] = (n * q + v) / (n + 1) # Q = media de v
        node.N[a] += 1

```

```

@staticmethod
def determine_move(model: AZNeuralNetwork, s: DotsAndBoxesGame,
mcts_parameters: dict) -> int:

    mcts = MCTS(model, s, mcts_parameters)
    probs = mcts.play(
        temp=0 #Selecciona el movimiento con máximo conteo de visitas,
#para llevar a cabo la mejor jugada posible
    )
    move = np.array(probs).argmax()

    valid_moves = np.where(s.l == 0)[0].tolist()
    assert move in valid_moves, f"El movimiento {move} no es un movimiento " +
"válido de {valid_moves}"

    return move

```

Finalmente están los archivos y clases definiendo los jugadores, según si juega la máquina con AlphaZero, con el método implementado de Alpha-Beta, o contra una persona. Para definir los comandos para mostrar la situación del juego, la toma de decisiones..., para entrenar nuestros modelos, permitiendo iniciar, reanudar, parar el proceso, entre otros. Un evaluador de los modelos..., todos pueden ser consultados en el proyecto original junto a métodos no tan relacionados con el objetivo de nuestro estudio pero que facilitarán la comprensión del funcionamiento del algoritmo.

## 4.4 Algoritmo Original: Implementación y Ajustes

Para implementar las casillas con valores aleatorios ha sido necesario crear un atributo adicional en la clase de **DotsAndBoxesGame**, este atributo es **r** una matriz del tamaño de **b**, la variable encargada de apuntar qué casillas han sido tomadas, pero con componentes aleatorias entre 0 y 1 que marcan cuánto vale cada casilla.

```

self.r = np.random.random((size,size))
self.r_total=sum(sum(self.r))

```

También definiremos una variable llamada **boxes\_to\_win** que será la mitad de la suma de todas estas componentes, al superarla sabremos que ya el jugador ha ganado e implementaremos el contador. Finalmente si no quedan líneas que dibujar se termina y se marca como empate,

aunque, al ser variables aleatorias, es extremadamente improbable que ambos terminen con justo la mitad de la puntuación total.

Este método se ejecutará al terminar cada turno.

```
def check_finished(self):
    assert self.result is None, "La partida ya terminó"

    # El jugador ha superado la mitad de
    # la puntuación total para ganar
    boxes_to_win = self.r_total/2
    self.con_neg=0
    self.con_pos=0
    for row in range(0,self.SIZE):
        for col in range(0,self.SIZE):
            if (self.b[row][col]>0):
                self.con_pos+=self.b[row][col]
            elif (self.b[row][col]<0):
                self.con_neg-=self.b[row][col]
    if self.con_pos> boxes_to_win:
        self.result = 1
    elif self.con_neg>boxes_to_win:
        self.result= -1

    if np.count_nonzero(self.l==0)==0:
        self.result=0
```

También se han modificado funciones auxiliares a la función **execute\_move** que permite hacer el conteo con el atributo **r** que hemos implementado. En esta nos aseguramos de que la caja no ha sido tomada aún y multiplicamos el valor de la caja correspondiente en **r** por el valor del jugador actual. Tras esto, lo guardamos en **b** como  $r_i$  si lo tomamos nosotros o  $-r_i$  si lo toma el oponente.

```
def capture_box(self, row: int, col: int):
    assert self.b[row][col] == 0, "Caja ya capturada"
    self.b[row][col] = self.current_player*self.r[row][col]
```

También es necesario cambiar la interfaz del juego, para permitir que el jugador vea cuánto vale cada caja, y también modificar el contador, el texto y la apariencia. De esto solo mostraremos el apartado del print de la tabla, modificado en la clase **DotsAndBoxesPrinter**:

```

def board_string(self) -> str:
    if self.SIZE > 6:
        sys.exit("ERROR: Para asegurar una calidad suficiente el" +
                 "tamaño del tablero mostrado está limitado a 6.\n")

    #Itera a través de las cajas desde arriba a abajo, izquierda a derecha
    string = ""
    for i in range(self.SIZE):

        # 1) Línea superior
        for j in range(self.SIZE):
            string += self.str_horizontal_line(
                line=self.get_lines_of_box((i, j))[0],
                last_column=(j == self.SIZE - 1)
            )
        string += "\n"

        # 2) Línea derecha e izquierda
        for repeat in range(3):
            for j in range(self.SIZE):
                string += self.str_vertical_line(
                    left_line=self.get_lines_of_box((i, j))[2],
                    print_line_number=(repeat == 1)
                )

        # Última línea vertical en una fila
        right_line = self.get_lines_of_box((i, self.SIZE - 1))[3]
        value = self.l[right_line]
        if value != 0:
            string += colored("|", value_to_color(value))
        else:
            if repeat == 1:
                string += f"{right_line}"
        string += "\n"

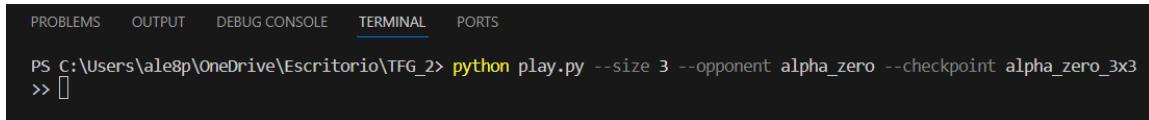
    # 3) Printea líneas inferiores para la última fila de cajas
    if i == self.SIZE - 1:
        for j in range(self.SIZE):
            string += self.str_horizontal_line(
                line=self.get_lines_of_box((i, j))[1],
                last_column=(j == self.SIZE - 1)
            )

```

```
# 4) Añadir r tras mostrar el estado del juego
string += "\n"
string += "\n"
string += "Lista de coeficientes de cajas: \n"
string += "\n"
string += str(self.r)+"\n"
string += "\n\n Necesario para ganar:"
string += str(self.r_total/2)
return string
```

## 4.5 Consola

Para iniciar el juego tan solo es necesario insertar el comando siguiente 4.2



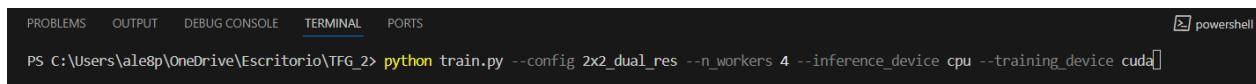
The screenshot shows a terminal window with the following text:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ale8p\OneDrive\Escritorio\TFG_2> python play.py --size 3 --opponent alpha_zero --checkpoint alpha_zero_3x3
>> []
```

Figura 4.2: Línea de código para iniciar el juego

Tenemos que asegurarnos de estar en la carpeta donde se encuentra nuestro proyecto. La variable *size* puede ser sustituida por 2,3 o 4 según el tamaño del tablero en el que queramos jugar, la variable *opponent* puede ser sustituida por el jugador al que nos queramos enfrentar, están disponibles alpha\_zero, person, random y alpha\_beta, en la variable *checkpoint* podemos indicar el checkpoint del modelo que queramos usar, asegurándonos de que es válido para la partida que queremos jugar, y en el caso de que seleccionemos el jugador alpha\_beta, es posible añadir una variable *d* con la profundidad de la poda deseada.

Si queremos entrenar nuestro propio modelo desde cero será necesario un archivo config, que incluye el proyecto original, y la siguiente línea de código 4.3



The screenshot shows a terminal window with the following text:

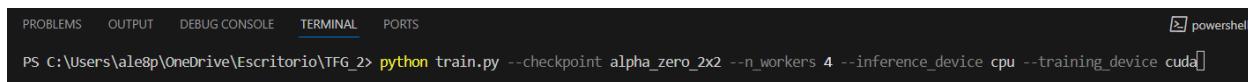
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ale8p\OneDrive\Escritorio\TFG_2> python train.py --config 2x2_dual_res --n_workers 4 --inference_device cpu --training_device cuda[]
```

Figura 4.3: Línea de código para entrenar modelo desde cero

De nuevo, tenemos que asegurarnos de estar en la carpeta donde se encuentra nuestro proyecto. La variable *config* puede ser sustituida por el archivo de configuración que queramos, tenemos uno para cada tamaño de tablero en el que queramos entrenar nuestro modelo, la variable *n\_workers* indica el número de "threads" o hilos que usaremos durante el proceso de aprendizaje.

La variable *inference\_device* indica el dispositivo que llevará a cabo el proceso de inferencia en nuestro modelo, que es un componente software de un sistema inteligente que aplicará reglas lógicas al conocimiento base para deducir nueva información, las opciones son *cuda* y *cpu*. Y finalmente, la variable *training\_device* que indica el dispositivo encargado de entrenar el modelo, para el que, de nuevo, las opciones son *cuda* y *cpu*.

Por último, si queremos reanudar el entrenamiento de un modelo desde un checkpoint, las opciones son similares a las anteriores, con la única diferencia siendo el uso de *checkpoint* en vez de *config*, e indicando el modelo que queremos continuar entrenando 4.4.

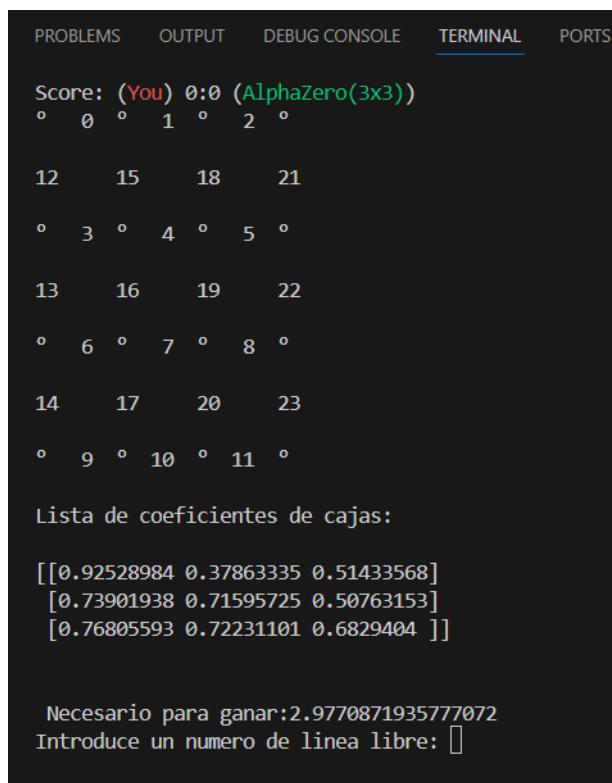


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ale8p\OneDrive\Escritorio\TFG_2> python train.py --checkpoint alpha_zero_2x2 --n_workers 4 --inference_device cpu --training_device cuda
```

Figura 4.4: Línea de código para entrenar modelo desde checkpoint

Por último se muestra una captura de pantalla del juego, para el que tenemos la lista de coeficientes de cajas y la puntuación necesaria para ganar.

Para jugar tan solo es necesario marcar el número que corresponde a la línea que queremos dibujar, los turnos avanzan automáticamente y la partida terminará cuando se alcance la puntuación necesaria para ganar 4.5.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Score: (You) 0:0 (AlphaZero(3x3))
  °  0  °  1  °  2  °

  12    15    18    21

  °  3  °  4  °  5  °

  13    16    19    22

  °  6  °  7  °  8  °

  14    17    20    23

  °  9  °  10  °  11  °

Lista de coeficientes de cajas:

[[0.92528984 0.37863335 0.51433568]
 [0.73901938 0.71595725 0.50763153]
 [0.76805593 0.72231101 0.6829404 ]]

Necesario para ganar: 2.9770871935777072
Introduce un numero de linea libre: 
```

Figura 4.5: Captura de pantalla con interfaz del juego



# 5 | Resultados

## 5.1 Enfrentamiento entre Algoritmos

Las siguientes gráficas mostrarán los resultados obtenidos por AlphaZero para 20, 25 y 25 iteraciones para partidas de 2x2, 3x3 y 4x4 de Dots and Boxes, respectivamente contra el algoritmo AlfaBeta de profundidad 1,2 y 3.

En cada gráfica se representará la pérdida de táctica, la pérdida de valor y la pérdida total durante el entrenamiento del modelo de AlphaZero, la gráfica izquierda mostrará la pérdida calculada cada actualización de lote durante el entrenamiento del modelo, y la de la derecha mostrará la evaluación de la pérdida que se halla al calcular de nuevo la pérdida de todos los lotes después de la actualización del modelo.

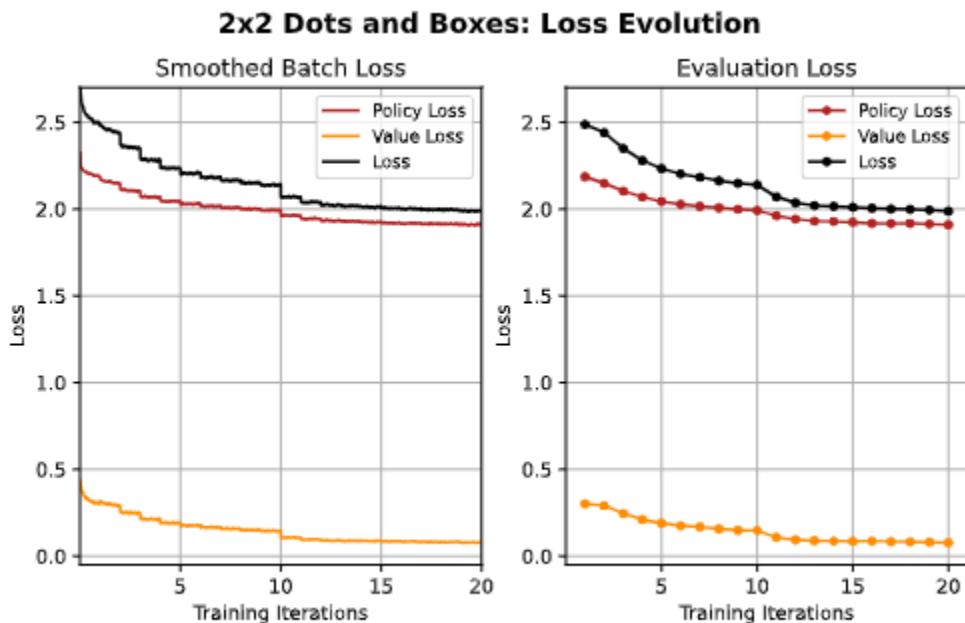


Figura 5.1: Gráfica de evolución de la pérdida en Dots and Boxes 2x2

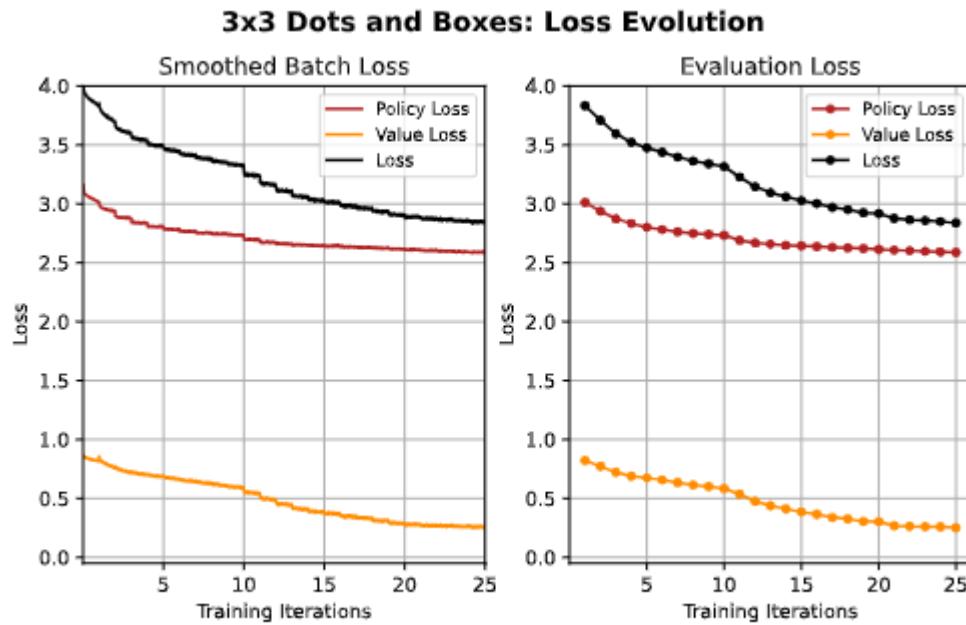


Figura 5.2: Gráfica de evolución de la pérdida en Dots and Boxes 3x3

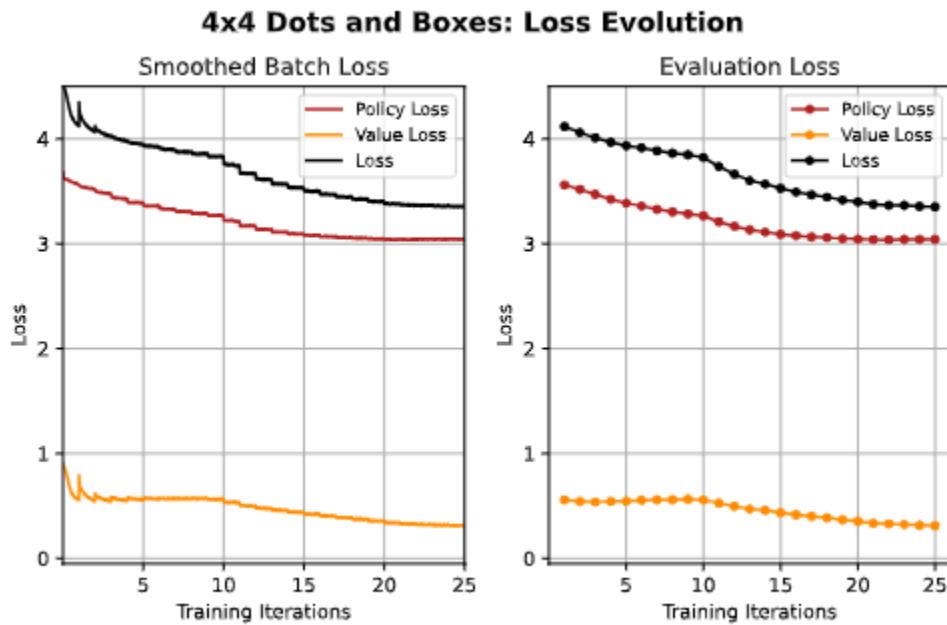


Figura 5.3: Gráfica de evolución de la pérdida en Dots and Boxes 4x4

Podemos observar que a partir de la décima iteración la pérdida total disminuye drásticamente, esto es debido a que, al usarse las últimas 10 iteraciones del entrenamiento, en la undécima ya se

dejan de usar las partidas iniciales, que son las que más empeoran su rendimiento.

También es importante mencionar que es posible reducir aún más la pérdida con más iteraciones, pero sería necesario más tiempo o más poder computacional. Tras cada actualización de la red neuronal, se evalúa la capacidad de AlphaZero jugando un total de 100 partidas, de las cuales en 50 tiene el primer turno, contra otros algoritmos implementados.

Al ser par el número total de cajas en  $2\times 2$  y  $4\times 4$ , la partida puede terminar en empate. En concreto, en la partida  $2\times 2$ , a partir de la segunda iteración, AlphaZero es capaz de ganar casi todas las partidas en las que empieza, aunque no es hasta la sexta iteración en la que la actuación de AlphaZero se estabiliza cuando no lo hace.

Las partidas son muy cortas en el tablero  $2\times 2$ , por lo que AlphaZero nunca supera el 50 % de victorias contra, por ejemplo, el jugador que emplea Alfa-Beta de profundidad 3. En el tablero de  $4\times 4$ , sin embargo, la actuación no para de mejorar en ninguna iteración, incluso en la final, lo que nos indica que 25 iteraciones no son suficientes para acercarnos a las capacidades reales en ese tablero de AlphaZero.

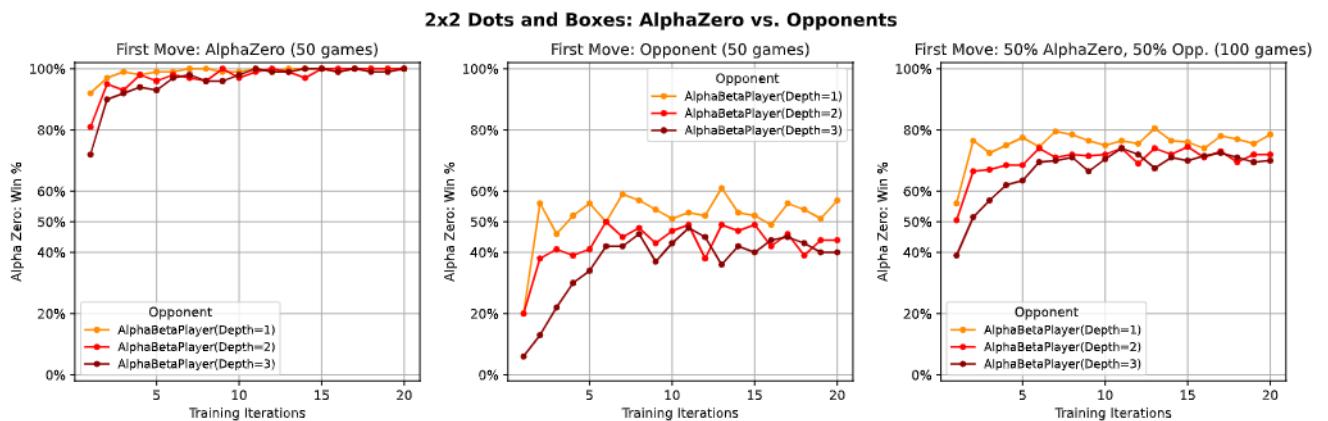


Figura 5.4: Porcentaje de victorias de AlphaZero en  $2\times 2$

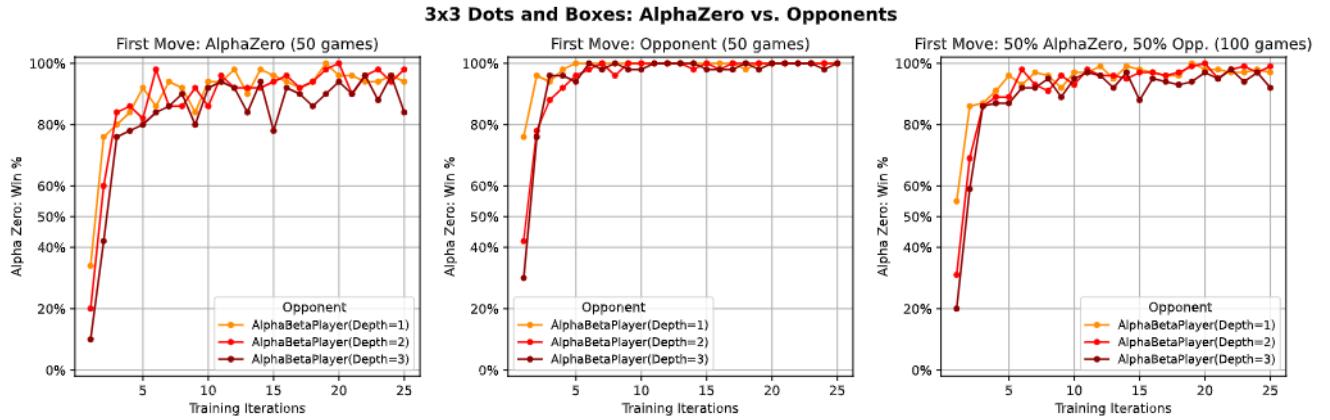


Figura 5.5: Porcentaje de victorias de AlphaZero en 3x3

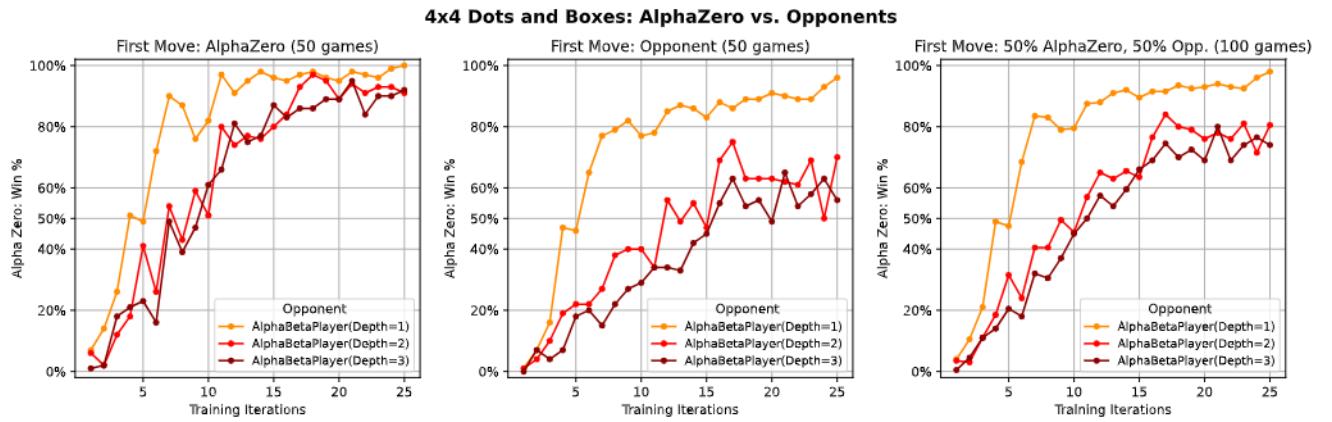


Figura 5.6: Porcentaje de victorias de AlphaZero en 4x4

El resumen de los resultados viene dado por las siguientes tablas,

Cuadro 5.1: AlphaZero vs. Oponentes: 2x2 Dots and Boxes (Tras 20 iteraciones)

Oponente	Victorias / Derrotas			% victorias de AlphaZero		
	Empezando	Segundo	Total	Empezando	Segundo	Total
Jugador Aleatorio	250 / 0 / 0	184 / 65 / 2	434 / 64 / 2	100.00	89.40	93.20
AlfaBeta(prof=1)	250 / 0 / 0	34 / 209 / 7	284 / 209 / 7	100.00	55.40	77.70
AlfaBeta(prof=2)	250 / 0 / 0	3 / 204 / 43	253 / 204 / 43	100.00	42.00	71.00
AlfaBeta(prof=3)	248 / 2 / 0	6 / 187 / 57	254 / 189 / 57	99.60	39.80	69.70

Cuadro 5.2: AlphaZero vs. Oponentes: 3×3 Dots and Boxes (Tras 25 iteraciones)

Oponente	Victorias / Derrotas			% victorias de AlphaZero		
	Empezando	Segundo	Total	Empezando	Segundo	Total
Jugador Aleatorio	250 / 0 / 0	250 / 0 / 0	500 / 0 / 0	100.00	100.00	100.00
AlfaBeta(prof=1)	240 / 0 / 10	250 / 0 / 0	490 / 0 / 10	96.00	100.00	98.00
AlfaBeta(prof=2)	235 / 0 / 15	250 / 0 / 0	485 / 0 / 15	94.00	100.00	97.00
AlfaBeta(prof=3)	227 / 0 / 23	247 / 0 / 1	476 / 0 / 24	90.80	99.60	95.20

Cuadro 5.3: AlphaZero vs. Oponentes: 4×4 Dots and Boxes (Tras 25 iteraciones)

Oponente	Victorias / Derrotas			% victorias de AlphaZero		
	Empezando	Segundo	Total	Empezando	Segundo	Total
Jugador Aleatorio	250 / 0 / 0	250 / 0 / 0	500 / 0 / 0	100.00	100.00	100.00
AlfaBeta(prof=1)	242 / 7 / 1	221 / 21 / 8	463 / 28 / 9	98.20	92.60	95.40
AlfaBeta(prof=2)	231 / 15 / 4	149 / 34 / 67	380 / 49 / 71	95.40	66.40	80.90
AlfaBeta(prof=3)	225 / 14 / 11	142 / 32 / 76	367 / 46 / 87	92.80	63.20	78.00

Es evidente que el algoritmo tiene resultados visiblemente inferiores cuando es el segundo en comenzar tanto en 2×2 como en 4×4, esto puede indicar que la posición de comienzo dota de una gran ventaja al jugador que la posea. De hecho, es posible ver que AlphaZero es capaz de compensar esta ventaja al ser superior al rival de manera más frecuente cuanto más grande es el tablero, llegando a un porcentaje de victoria en el tablero 2×2 del 39.80 % cuando no tiene el turno inicial y de 63.20 % en el tablero 4×4.

Sin embargo, en el tablero 3×3 los resultados sugieren lo contrario, que el jugador con la segunda posición parte con cierta ventaja. Ascendiendo casi al 100 % de victorias contra todos los algoritmos menos contra el alfabeto de profundidad 3, contra el que perdió tan solo una partida (estos resultados son tan dispares respecto al 4×4 en parte por la ausencia del empate).

Los resultados pueden ser consultados en [josch14](#) y se refieren al juego sin modificaciones, no deberían variar mucho respecto al cambio propuesto y se obtienen de la misma forma, tan solo sería necesario entrenar el modelo con nuestras implementaciones y enfrentarlo a los mismos oponentes pero, debido a la fuerza computacional necesaria para entrenar al modelo para que muestre resultados interesantes, he optado por mostrar los resultados del original.

Existe la posibilidad de entrenar el modelo desde un checkpoint, aunque, de nuevo, el tiempo empleado para terminar una epoch es extremadamente alto incluso para el tablero de 2×2.

## 5.2 Enfrentamiento contra Humanos

El análisis de resultados contra humanos muestra resultados aún más aplastantes que los mostrados contra los distintos algoritmos de poda alfa beta, siendo que tan solo se ha conseguido ganar una vez contra AlphaZero.

La victoria contra AlphaZero ha sido solo posible con el tablero  $2 \times 2$  y siendo el jugador humano el que empezaba, excluyendo las partidas en las que se ha usado la estrategia ganadora, para las que se ha podido forzar empates y victorias.

El motivo del gran porcentaje de victorias de AlphaZero contra los distintos algoritmos y humanos es la capacidad de actuar en consecuencia a movimientos más "profundos" de los que los oponentes pueden ver, permitiéndole al menos forzar empates cuando partía de una situación desfavorable y forzar victorias cuando lo hacía de una favorable, e incluso buscando victorias cuando el oponente cometía un error. Los algoritmos de poda, al igual que los humanos, son capaces de hacer uso de estrategias que conlleve sacrificar cajas para tomar luego una mayor cantidad, esto es especialmente evidente en el tablero  $4 \times 4$ , en el que, por regla general, se suele ir ganando hasta que llega un momento en el que AlphaZero te fuerza a tomar una línea para la que posteriormente es capaz de tomar todas las cajas necesarias y ganar en tan solo un turno.

Tras la implementación del valor de cajas aleatorias sí que ha sido posible ganarle con bastante frecuencia, sobre todo en las partidas en las que casi todo el peso de la partida caía en una caja en concreto. Esto es debido a que, aunque se ha entrenado el modelo un par de iteraciones con los nuevos atributos tras reanudar el aprendizaje sobre un checkpoint, no han sido suficientes para hacer una diferencia tras las múltiples iteraciones que han sido entrenadas previamente.

Podemos ver un ejemplo de la toma de decisiones en la figura 5.7

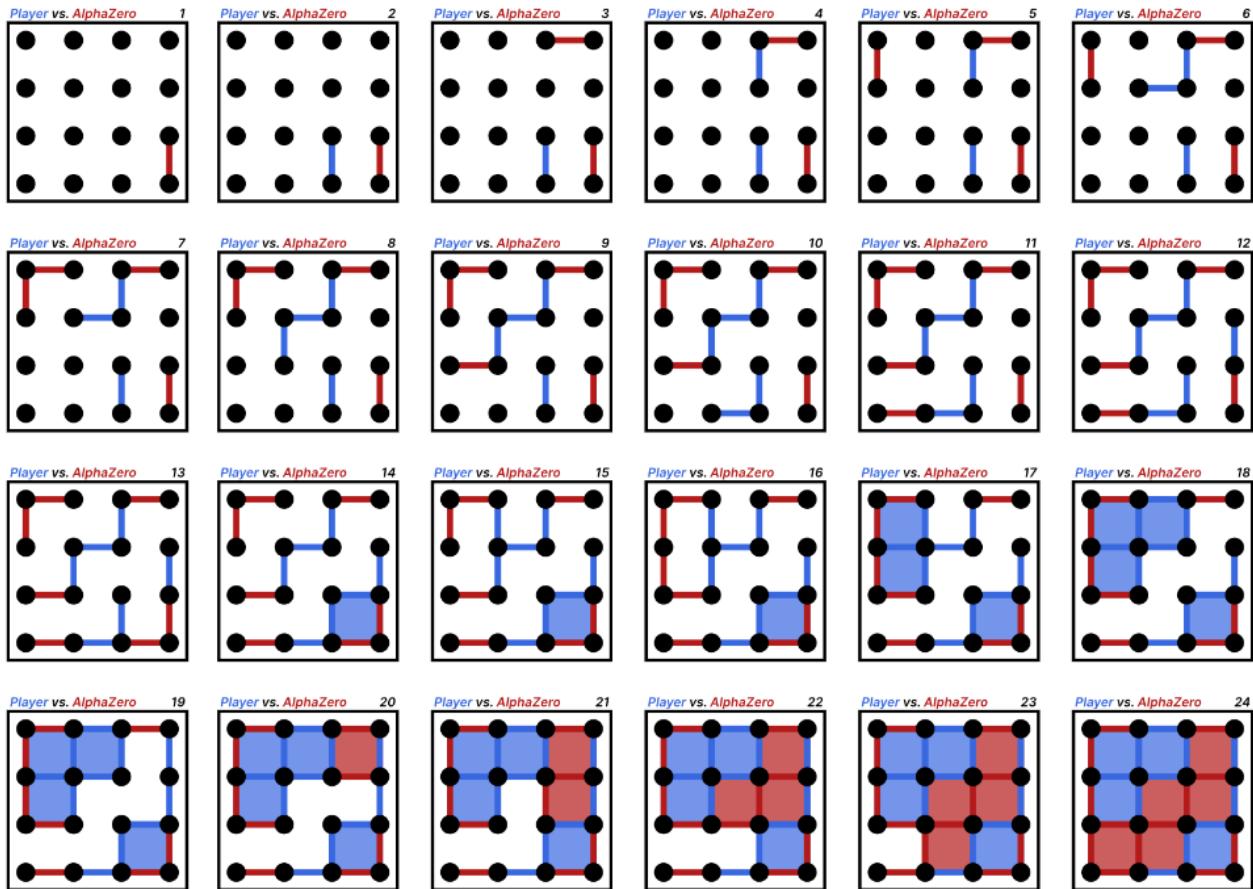


Figura 5.7: Partida de jugador vs AlphaZero en el tablero 3x3



# Conclusión (español)

A lo largo de la memoria hemos analizado los elementos que componen AlphaZero y hemos tratado de describirlos desde una perspectiva formal, probando resultados que tratan de contestar el cómo y el por qué funciona. Durante la implementación hemos podido observar que los resultados son lo suficientemente positivos como para vencer a cualquier jugador casual de Dots and Boxes, además de permitirnos aplicar los conceptos aprendidos sobre las distintas redes neuronales descritas, inicializaciones, métodos de búsqueda, funciones de activación, entre otros...

Es de especial importancia entender por qué las redes neuronales funcionan, el teorema de aproximación universal nos permite entender qué es realmente lo que hace una red neuronal e intenta acotar ese espacio de hipótesis con todas las funciones descriptibles por la red. Otros teoremas como el de No free lunch, nos permite también conocer resultados sobre algoritmos de optimización y nos abre a reflexionar sobre la comparación entre algoritmos, si de verdad existe un algoritmo "mejor" que otro. También es importante conocer el contexto en el que surgió AlphaZero, y un repaso de los algoritmos previos y de la situación histórica nos permite dilucidar sobre las distintas decisiones que se tomaron durante su implementación.

Los resultados muestran una mejora evidente respecto a los métodos usados previos a la llegada de AlphaZero con tiempos de procesamiento mínimos incluso cuando su uso es en sistemas con baja capacidad computacional. Uno de los principales focos de interés es que, aunque para el entrenamiento de las redes el proceso es muy costoso, a la hora de su uso es asequible por multitud de dispositivos, a diferencia de métodos como el minimax con la poda alfa beta que a medida que aumenta la profundidad con la que se usan, más recursos requieren, creciendo a un ritmo enorme.

Una de las principales limitaciones ha sido la capacidad computacional, existen plataformas como LightningAI que permiten el uso de GPUs de manera remota permitiendo un entrenamiento con dispositivos computacionalmente potentes y con duraciones altas. Otra posible limitación podría considerarse el uso de modelos entrenados con pocas iteraciones a la hora de usarlos como referencia en el capítulo de resultados, tanto el modelo entrenado para  $3\times 3$  como el entrenado para  $4\times 4$  muestran indicios de posibles mejoras si se siguen entrenando durante más iteraciones y, de la misma forma, los algoritmos usados para enfrentarse a AlphaZero, al ser altamente costosos, han podido ser usados solo con profundidades de valores inferiores a 4.

Entre las mejoras más evidentes podríamos considerar el aumento de tiempo de entrenamiento

en nuestros modelos, ya que el entrenamiento a partir del checkpoint original ha sido de tan solo 2 iteraciones, y la implementación de tableros más grandes, aunque de nuevo dependeríamos del uso de plataformas o dispositivos que proporcionasen capacidades de procesamiento superiores. Otra mejora a considerar sería el aumento de la muestra de partidas contra jugadores, para el que en este proyecto tan solo se han considerado 6 jugadores de perfiles diferenciados.

Todo algoritmo puede mejorarse y, en nuestro caso, se podría hacer uso de inicializaciones distintas a la de Xavier, como la inicialización He en redes donde se usa ReLU, también es posible mejorar el sistema de rendición temprana, permitiendo jugar más partidas al identificar derrotas seguras más rápido, puede afinarse la configuración, o incluso mejorar la arquitectura pero, como puede consultarse en los anexos, en el mundo de las redes neuronales es extremadamente difícil conseguir acotaciones o estimaciones que puedan guiar nuestras arquitecturas por lo que, a priori, es difícil saber cuánto margen de mejora hay para la arquitectura propuesta.

AlphaZero representa un gran avance en el campo de la inteligencia artificial, es por eso que a lo largo de esta memoria hemos podido ver desde su capacidad para aprender y dominar juegos sin intervención humana hasta su rendimiento sobre-humano para superar a programas hasta entonces invictos en ajedrez, shogi y Go, entre otros.

La implementación de AlphaZero y su análisis en este proyecto nos ha permitido observar no solo la complejidad y los desafíos que supone, sino también puede servir como inicio en el estudio de las distintas arquitecturas y modelos basados en redes neuronales, evidenciando las amplias oportunidades que las tecnologías involucradas en el algoritmo ofrecen, además de permitir su aplicación en dispositivos que no están preparados exclusivamente para procesos altamente costosos. Además, nos ha permitido repasar conceptos básicos teniendo la posterior implementación como hilo conductor, guiándonos a través de estos conceptos y viendo como cada uno "superá" o busca una forma de mejorar a su predecesor en un contexto determinado.

Resumiendo, este proyecto no solo ha permitido una inmersión en el mundo de la inteligencia artificial avanzada, sino que también ha contribuido a una comprensión más profunda de cómo abordar un problema tan complejo usando elementos ya conocidos dentro del área del machine learning. Las implicaciones de este trabajo van más allá de los juegos que hemos podido ver, sugiere aplicaciones potenciales en diversas áreas y abre un abanico infinito de posibilidades. La verdadera importancia de AlphaZero no son sus aplicaciones a día de hoy, sino las fronteras que abre para futuras investigaciones y desarrollos tecnológicos.

# Conclusion (english)

Throughout this document, we have analyzed the elements that compose AlphaZero and have attempted to describe them from a formal perspective, testing results that try to answer how and why it works. During the implementation, we observed that the results are sufficiently positive to defeat any casual player of Dots and Boxes, and allowed us to apply the concepts learned about the various neural networks described, initializations, search methods, activation functions, among others...

It is specially important to understand why neural networks work, the universal approximation theorem helps us understand what a neural network actually does and attempts to bound the hypothesis space with all the functions describable by the network. Other theorems, such as the No Free Lunch theorem, also allows us to understand results about optimization algorithms and prompts us to think about the comparison between algorithms, whether there truly is a "better" algorithm than another. It is also important to know the context in which AlphaZero emerged, and a review of previous algorithms and the historical situation allows us to shed light on the various decisions made during its implementation.

The results show a clear improvement over the methods used before the arrival of AlphaZero with minimal processing times, even when used in systems with low computational capacity. One of the main points of interest is that, although the training of the networks is very costly, their use is affordable for multitude of devices, unlike methods such as minimax with alpha-beta pruning which, as the depth of use increases, requires more resources, growing at an enormous rate.

One of the main limitations has been computational capacity. Platforms like LightningAI allow the use of GPUs remotely, enabling training with computationally powerful devices for extended durations. Another possible limitation could be the use of models trained with few iterations as a reference in the results chapter, both the model trained for 3x3 and the one trained for 4x4 show signs of potential improvements if they continued to be trained for more iterations. Similarly, the algorithms used to compete against AlphaZero, being highly costly, could only be used with depths of values lower than 4.

Among the most evident improvements, we could consider increasing the training time of our models, as the training from the original checkpoint has only been for 2 iterations, and the implementation of larger boards, although again we would depend on the use of platforms or

devices that provide superior processing capabilities. Another improvement to consider would be increasing the sample size of games against players, for which only 6 players of different profiles were considered in this project.

Any algorithm can be improved and, in our case, we could use initializations other than Xavier, such as He initialization in networks where ReLU is used. It is also possible to improve the early stopping system, allowing more games to be played by identifying certain defeats faster, fine-tuning the configuration, or even improving the architecture. However, as can be consulted in the appendices, in the world of neural networks, it is extremely difficult to achieve constraints or estimates that can guide our architectures. Therefore, it is initially difficult to know how much room for improvement there is for the proposed architecture.

AlphaZero represents a significant breakthrough in the field of artificial intelligence, which is why throughout this thesis we have been able to explore in depth the concepts, algorithms, and architecture that underpin it. We have observed its ability to learn and master games without human intervention and its superhuman performance in surpassing previously undefeated programs in chess, shogi, and Go.

The implementation of AlphaZero and its analysis in this project have allowed us to observe not only the complexity and challenges it entails but also to serve as a starting point for studying various architectures and models based on neural networks, demonstrating the wide-ranging opportunities that the technologies involved in the algorithm offer.

In summary, this project has not only facilitated an immersion into the world of advanced artificial intelligence but also contributed to a deeper understanding of how to tackle such a complex problem using already known elements within the field of machine learning. The implications of this work go beyond the games we have explored; it suggests potential applications in various areas and opens up an infinite array of possibilities. The true importance of AlphaZero lies not in its applications today, but in the boundaries it opens for future research and technological developments.

# Bibliografía

- [1] ANONYMOUS. How to choose the number of hidden layers and nodes in a feedforward neural network? <https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw>.
- [2] ANONYMOUS. Purpose of dirichlet noise in the alphazero paper. <https://stats.stackexchange.com/questions/322831/purpose-of-dirichlet-noise-in-the-alphazero-paper>.
- [3] ANONYMOUS. Universal approximation theorem for equivariant maps by group cnns. <https://openreview.net/pdf?id=7TBP8k7TLFA>.
- [4] BAHETI, P. Activation functions in neural networks [12 types use cases]. <https://www.v7labs.com/blog/neural-networks-activation-functions>.
- [5] BARKER, J., AND KORF, R. Solving dots-and-boxes. *Proceedings of the AAAI Conference on Artificial Intelligence* 26, 1 (Sep. 2021), 414–419.
- [6] BERZAL, F. Backpropagation. <https://elvex.ugr.es/decsai/computational-intelligence/slides/N2>
- [7] BOGOMOLNY, A. Minimax principle. <https://www.cut-the-knot.org/arithmetic/MinMax.shtml>.
- [8] CAPARRINI, F. S. Mcts. <https://www.cs.us.es/ fsancho/Blog/posts/MCTS.md.html>.
- [9] CAPARRINI, F. S. Minimax. <https://www.cs.us.es/ fsancho/Blog/posts/Minimax.md.html>.
- [10] CAPARRINI, F. S. Redes neuronales. [https://www.cs.us.es/ fsancho/Blog/posts/Redes\\_Neuronales/](https://www.cs.us.es/ fsancho/Blog/posts/Redes_Neuronales/).
- [11] CARLSSON, F., AND ÖHMAN, J. Alphazero to alphahero. <https://www.diva-portal.org/smash/get/diva2:1350740/FULLTEXT01.pdf>.
- [12] CHESSPROGRAMMING. Stockfish vs alphazero match. <https://www.chessprogramming.org/AlphaZero>.

- [13] DAUBECHIES, I. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.
- [14] DAVID SILVER, JULIAN SCHRITTWIESER, E. A. Mastering the game of go without human knowledge. [https://discovery.ucl.ac.uk/id/eprint/10045895/1/agz\\_unformatted\\_nature.pdf](https://discovery.ucl.ac.uk/id/eprint/10045895/1/agz_unformatted_nature.pdf).
- [15] DAVID SILVER, THOMAS HUBERT, E. A. Alphazero: Shedding new light on chess, shogi, and go. <https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/>.
- [16] DE ÁLGEBRA UNIVERSIDAD DE SEVILLA, D. Introducción a la teoría de grupos. <https://asignatura.us.es/algbas/groups/>.
- [17] DEEPLARNING.AI. Initialization ai notes. <https://www.deeplearning.ai/ai-notes/initialization/index.html>.
- [18] ENSMINGER, N. Is chess the drosophila of artificial intelligence? a social history of an algorithm. *Social Studies of Science* 42, 1 (2012), 5–30. PMID: 22530382.
- [19] FLORES, W. G. Perceptrón multicapa y algoritmo backpropagation. <https://www.tamps.cinvestav.mx/wgomez/material/RP/MLP.pdf>.
- [20] FOSTER, D. Alphago zero explained in one diagram. <https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0>.
- [21] FU, K. Reinforcement learning in alphazero. [https://tjmachinelearning.com/lectures/1819/guest/alphazero/RL\\_in\\_AlphaZero.pdf](https://tjmachinelearning.com/lectures/1819/guest/alphazero/RL_in_AlphaZero.pdf).
- [22] GEEKSFORGEEKS. Residual networks (resnet) – deep learning. <https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/>.
- [23] GLOROT, X., AND BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.
- [24] GOLDBLUM, M., FINZI, M., ROWAN, K., AND WILSON, A. G. The no free lunch theorem, kolmogorov complexity, and the role of inductive biases in machine learning, 2024.
- [25] HALL, M. Game theory and von neumann's minimax theorem. <https://spark.bethel.edu/cgi/viewcontent.cgi?article=1002&context=honors-works>.
- [26] HE, K., ZHANG, X., REN, S., AND SUN, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.
- [27] IBM. What are convolutional neural networks. <https://www.ibm.com/topics/convolutional-neural-networks>.
- [28] INT8.IO. Monte carlo tree search – beginners guide. <https://int8.io/monte-carlo-tree-search-beginners-guide/>.

- [29] JAVATPOINT. Alpha-beta pruning. <https://www.javatpoint.com/ai-alpha-beta-pruning>.
- [30] KLEIN, D. Neural networks for chess the magic of deep and reinforcement learning revealed. <https://arxiv.org/ftp/arxiv/papers/2209/2209.01506.pdf>.
- [31] KLUSOWSKI, J. M., AND BARRON, A. R. Approximation by combinations of relu and squared relu ridge functions with  $\ell^1$  and  $\ell^0$  controls, 2018.
- [32] MAHARAJ, S., POLSON, N., AND TURK, A. Chess AI: competing paradigms for machine intelligence. *CoRR abs/2109.11602* (2021).
- [33] MICHELANGELO. Alphazero for dummies! [https://medium.com/@\\_michelangelo\\_/alphazero-for-dummies-5bcc713fc9c6](https://medium.com/@_michelangelo_/alphazero-for-dummies-5bcc713fc9c6).
- [34] MOHRI, M., ROSTAMIZADEH, A., AND TALWALKAR, A. *Foundations of Machine Learning*, 2 ed. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2018.
- [35] ROTMAN, J. *An Introduction to the Theory of Groups*. Graduate Texts in Mathematics. Springer New York, 2012.
- [36] SHAI SHALEV-SHWARTZ, S. B.-D. *Understanding Machine Learning: From Theory to Algorithms*. 2014.
- [37] SILVER, D., AND ET AL., T. H. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR abs/1712.01815* (2017).
- [38] SILVER, D., AND HUANG, E. A. Mastering the game of go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489.
- [39] STATHAKIS, D. How many hidden layers and nodes? [http://dstath.users.uth.gr/papers/IJRS2009\\_Stathakis.pdf](http://dstath.users.uth.gr/papers/IJRS2009_Stathakis.pdf).
- [40] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning An Introduction*, second ed. 2018.
- [41] SZYMANSKI, L., McCANE, B., AND ALBERT, M. H. The effect of the choice of neural network depth and breadth on the size of its hypothesis space. *CoRR abs/1806.02460* (2018).
- [42] TOWARDSDATASCIENCE. Comprehensive guide to convolutional neural networks. <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [43] TOWARDSDATASCIENCE. General mcts alphazero implementation. <https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867>.
- [44] VARTY, J. A step-by-step look at alpha zero and monte carlo tree search. <https://joshvarty.github.io/AlphaZero/>.

- [45] WIKIPEDIA. Propagación hacia atrás. <https://en.wikipedia.org/wiki/Backpropagation>.
- [46] ZHOU, D. Universality of deep convolutional neural networks. *CoRR abs/1805.10769* (2018).
- [47] ZHOU, D.-X. Deep distributed convolutional neural networks: Universality. *Analysis and Applications* 16 (03 2018).

## **Anexos**



## Ruido de Dirichlet

Para asegurar que se exploran adecuadamente los movimientos en AlphaZero, como comenta el usuario *monk* en [2], se usa el ruido de Dirichlet previamente al cálculo del nodo raíz, específicamente de esta forma:

$$P(s, a) = (1 - \epsilon)p_a + \epsilon\eta_a$$

donde  $\eta \sim (0.03)$  y  $\epsilon = 0.25$ , que nos asegura que todos los movimientos son intentados, pero que la búsqueda seguirá priorizando los movimientos buenos.

La distribución de Dirichlet tiene la siguiente interpretación en este contexto:

Cuando  $\alpha$  es el vector contador de los resultados observados de una distribución categórica desconocida con probabilidades de resultado  $\pi$ , entonces  $Dir(\alpha)(\pi)$  es la probabilidad de que  $Cat(\pi)$  sea la distribución real dado el contador de  $\alpha$ .

Ahora,  $P(s, a)$  estima la probabilidad de que un buen jugador realice la jugada  $a$  en la situación  $s$ , que son los parámetros de su distribución categórica, que son los que AlphaZero quiere aprender. De esta forma,  $Dir(\alpha)$  reflejaría una estimación razonable para  $\pi = P(s, a)$  si observáramos un buen jugador mover  $\alpha$  veces. Pero si  $\alpha_i = 0$ , entonces todo  $\pi \sim Dir(\alpha)$  tendría un  $\pi_i = 0$ , lo que perjudica la exploración. Añadiendo el ruido asumiremos que cada movimiento ha sido, al menos, observado un pequeño número de veces.

La distribución simétrica de Dirichlet es especialmente útil cuando una distribución de Dirichlet sobre componentes es requerida pero no tenemos información previa que permita favorecer una componente sobre otra, como se da en AlphaZero al explorar.

Como todos los elementos del vector de parámetros tienen el mismo valor, la distribución simétrica de Dirichlet puede ser parametrizada por un escalar,  $\alpha$ , y su función de densidad viene dada por:

$$f(x_1, \dots, x_k; \alpha) = \frac{\Gamma(\alpha K)}{\Gamma(\alpha)^K} \prod_{i=1}^K x_i^{\alpha-1}$$

## Juegos Perfecto

En *Teoría de Juegos*, un juego resuelto se dice de aquel cuyo resultado es predecible correctamente desde cualquier posición, asumiendo un juego perfecto por parte de ambos jugadores.

Denominamos a ese comportamiento de un jugador que conduce al mejor resultado posible, independiente de la respuesta del oponente, como juego perfecto. Este puede usarse en juegos de información perfecta, pero también en juegos de información no perfecta, en la que esta estrategia garantizará el resultado mínimo esperado más alto, independiente de la estrategia del oponente.

Ejemplos de esto podrían darse en el popular juego piedra, papel o tijeras, en el que la estrategia perfecta sería escoger aleatoriamente cada una de las opciones con la misma probabilidad, o en el 3 en raya, cuya estrategia ganadora viene parcialmente descrita en la figura 8.

En [5] podemos ver el proceso para hallar la estrategia ganadora en Dots and Boxes para tableros de 4 casillas de alto y 5 de ancho. Esta es una de las motivaciones detrás de la implementación de una matriz que modifique el valor de cada casilla, ya que no es difícil encontrar esta estrategia para tamaños bajos como en  $2 \times 2$ .

De esta forma podemos "ralentizar" la velocidad a la que nuestro algoritmo llega a esta estrategia, aunque, como hemos dicho anteriormente, es posible llegar a un juego perfecto tanto si se sabe el valor de dichas casillas (seguiría siendo un juego con información completa), como si no se sabe (se buscaría maximizar el resultado mínimo posible).

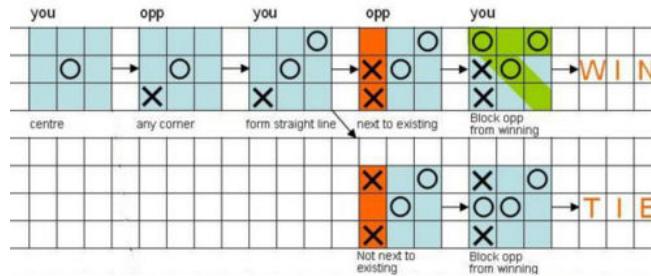


Figura 8: Ejemplo de estrategia ganadora en el 3 en raya

## Estimadores de Arquitecturas Óptimas

A lo largo de esta memoria hemos definido redes neuronales y hemos mencionado los elementos que las conforman, los nodos y las capas pero, una pregunta que puede surgir es ¿existe algún modo de encontrar la arquitectura perfecta?

Esta pregunta ha sido un tema de gran interés desde el inicio de las redes neuronales en la que se han hecho pequeños avances aunque, a día de hoy, lo más frecuente es estimar nuestras arquitecturas mediante prueba y error.

Existe una gran cantidad de fórmulas que permiten conseguir buenos resultados, aunque todas estas son aproximaciones, una por ejemplo es la mencionada por el usuario *hobs* en [1]:

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))}$$

donde  $N_h$  es el número de neuronas en la capa,  $N_s$  es el número de muestras en el conjunto de datos de entrenamiento,  $N_o$  es el número de neuronas output,  $N_i$  es el número de neuronas input y  $\alpha$  es un factor arbitrario entre 2 y 10.

Esta estimación se usa para tener un límite superior para prevenir el over-fitting en modelos de aprendizaje supervisado. Como podemos ver, son estimaciones extremadamente vagas ( $\alpha$  puede variar mucho las estimaciones) que tan solo sirven de guía para terminar usando prueba y error.

En [39] se mencionan, además de la prueba y error, métodos como la búsqueda heurística, la búsqueda exhaustiva, algoritmos constructivos con poda... y los compara, aunque en este anexo veremos una en concreto que es muy interesante.

Esta la podemos encontrar en [41] donde también podemos encontrar su prueba que veremos a continuación, para ello, veremos un par de definiciones primero.

Una red neuronal con una arquitectura específica no es más que un espacio de hipótesis, al que denominaremos  $\mathcal{H}$ . La arquitectura viene determinada por un conjunto de hiperparámetros, alguno de ellos, como el número de inputs  $U_0 = n$ , vienen dados por los atributos de los datos con los que la red tiene que trabajar, mientras que otros como el número de capas ocultas  $L$ , número de neuronas ocultas por capa  $U_l$  o función de activación  $\sigma$  son escogidas por el usuario.

En esta prueba nos referiremos a redes con tan solo un output, por lo que la función producida por nuestra red viene dada por:

$$h(x) = \sum_{j=1}^{U_L} w_j y_j^{[L]} + w_0$$

con  $w_j$  y  $w_0$  siendo respectivamente el peso y el bias de la única neurona output, veamos el output de la  $i$ -ésima neurona de la capa  $l$

$$y_i^{[l]} = \sigma \left( \sum_{j=1}^{U^{[l-1]}} w_{ij}^{[l]} y_j^{[l-1]} + w_{i0}^{[l]} \right)$$

donde  $\sigma$  es alguna función de activación,  $w_{ij}^{[l]}$  es el coeficiente o peso del  $j$ -ésimo input desde la capa  $l - 1$ ,  $w_{i0}^{[l]}$  es el bias,  $U_0 = n$  y con  $y_j^{[0]} = x_j$  siendo el  $j$ -ésimo atributo del input  $x \in \mathcal{R}^n$

El número total de parámetros entrenables (pesos y biases) en una red totalmente conectada con un único output es

$$W = \sum_{l=1}^L (U_{(l-1)} + 1) U_l + U_L$$

de nuevo,  $U_0 = n$ .

Una asignación concreta de valores a los coeficientes y biases se denominará como el estado de la red. El espacio de hipótesis  $\mathcal{H}$  dado por una red neuronal con una arquitectura específica es el conjunto de todas las funciones que esta arquitectura es capaz de devolver con todas las combinaciones de coeficientes y biases. Cuando sea necesario especificar la arquitectura, denotaremos el correspondiente espacio de hipótesis como  $\mathcal{H}_n - U_1 - \dots - U_L$ .

Para una red de  $W$  parámetros, donde cada uno puede tomar valores de un conjunto finito  $\mathcal{V}$  de cardinalidad  $V = |\mathcal{V}|$ , hay un total de  $V^W$  estados. Sin embargo, diferentes estados pueden dar lugar al mismo mapeo de funciones, y esas relaciones de equivalencia nos interesan para acotar nuestras estimaciones. Esto es consecuencia de que el orden de la suma de los coeficientes de las neuronas no influye en el resultado final, para reflejar esto haremos uso de la siguiente figura 9

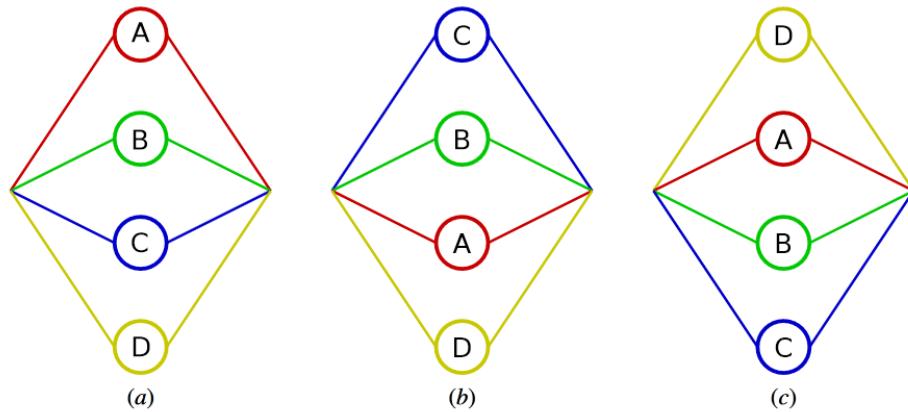


Figura 9: Tres permutaciones distintas de 4 neuronas en una capa oculta que no afectan al output

En la figura podemos ver que un cambio de estado como el de la figura **a** a la figura **b** es análogo a que la neurona **A** cambie su posición con la neurona **C**. La figura **c** simplemente cambia las neuronas de tal forma que la posición de **A** pasa a la de **B**, la de **B** a la de **C**...

Con este razonamiento es lógico pensar que para una capa de  $U_l$  neuronas y una elección particular de valores de conexiones, hay hasta  $U_l!$  permutaciones del orden de suma produciendo el mismo mapeo, sin importar el número de inputs y outputs de la capa. De hecho, podría haber incluso menos de  $U_l!$  permutaciones para algunas elecciones de valores de conexiones si los coeficientes de las neuronas se ajustan de tal forma que 2 o más permutaciones de neuronas producen el mismo estado, es el caso en el que todas tienen el mismo peso de entrada y salida, para el que todas las permutaciones son equivalentes.

A la hora de tener en cuenta la capacidad de mapeo de la combinación de múltiples capas también es necesario observar todas las combinaciones posibles de permutaciones de neuronas que puedan preservar el resultado en cada capa. De nuevo, usaremos una figura que nos permita deducir las combinaciones que preservan el output 10

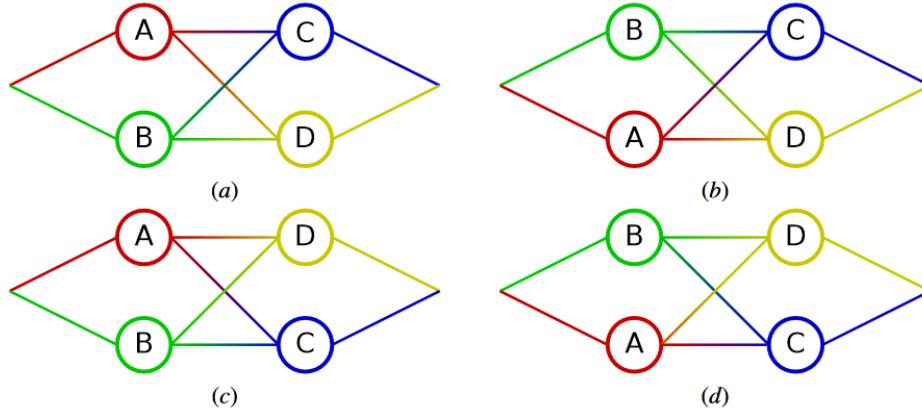


Figura 10: Cuatro permutaciones distintas de dos capas ocultas consecutivas, con dos neuronas cada una, que no afectan al output

La figura muestra todas las combinaciones posibles para dos capas ocultas consecutivas con dos neuronas cada una. Como cada una tiene dos neuronas, tenemos  $2!$  permutaciones equivalentes. En general, dependiendo de las elecciones de valores de parámetros en las conexiones, hay hasta  $V^W$  permutaciones de las neuronas que preservan el mapeo de funciones de la red.

Si tomamos un conjunto finito de  $V$  valores, entonces hay  $V^W$  posibles estados en una red con una arquitectura con un total de  $W$  parámetros. Si cada estado de  $V^W$  fuera parte de una clase de equivalencia de al menos  $\prod_l U_l!$  estados produciendo el mismo mapeo de funciones, sería obvio que la red no podría mapear más de  $V^W / \prod_l U_l!$  funciones de mapeo únicas. La situación no es tan simple ya que hay estados con mismo valores y distintos parámetros que no tienen  $\prod_l U_l!$  permutaciones distinguibles. Sin embargo, usando resultados de [35], podemos establecer que el límite superior de mapeo de funciones únicas es  $\prod_l U_l!$ .

Una vez definidos estos conceptos, podemos enunciar el teorema y comenzar con su prueba.

**Teorema .1.** *El límite superior en el tamaño del espacio de hipótesis  $\mathcal{H}$  de una red completamente*

conectada con una función de activación arbitraria  $\sigma$  es  $\mathcal{O}(V^W / \prod_l U_l!)$ , donde  $L$  es el número de capas ocultas,  $U_l$  es el número de neuronas en la capa  $l$ , y  $W$  es el número total de parámetros, y los parámetros  $w_{ij} \in \mathcal{V}$ , donde  $|\mathcal{V}| = V$  es finito

**Demostración.** Sea  $X$  el conjunto de todos los posibles estados de una red neuronal de  $W$  parámetros. En nuestro caso,  $|X| = V^W$ . Para acotar el tamaño de  $\mathcal{H}$ , podemos crear una partición de  $X$  en clases de equivalencia de hipótesis idénticas y contar el número de clases. A continuación, recordaremos varias definiciones:

**Definición .1.** Un grupo [16] es un conjunto no vacío  $G$  dotado de una operación asociativa  $*$  conteniendo un elemento  $e$  tal que:

1.  $e * a = a = a * e$  para todo  $a \in G$
2. Para cada  $a \in G$ , hay un elemento  $b \in G$  tal que  $a * b = e = b * a$

Por la definición de grupo el conjunto de las biyecciones  $X \times X$  (o permutaciones) de los  $W$  parámetros que no afectan al mapeo de funciones resultante de la red es un grupo. La operación  $*$  es una permutación. De hecho, podemos aplicar una permutación a otra y obtener otra permutación de nuevo. La permutación identidad  $e$  es la permutación que mapea a cada elemento a sí mismo. Además, por lo mencionado anteriormente, el grupo  $G$  consiste de  $\prod_l U_l!$  permutaciones de parámetros isomorfas al producto de permutaciones del orden de  $U_l$  neuronas en cada capa oculta.

**Definición .2.** Un  $G$ -conjunto, [35] página 55, se denomina a  $X$  tal que si  $X$  es un conjunto y  $G$  es un grupo, cumple que existe una función  $\alpha : G \times X \mapsto X$  (llamada una acción), denotada por  $\alpha : (g, x) \mapsto gx$  tal que:

1.  $e * x = x$  para cada  $x \in X$  y
2.  $g(hx) = (gh)x$  para cada  $g, h \in G$  y  $x \in X$

$X$  es un  $G$ -conjunto ya que las permutaciones desde  $G$  reordenan el valor de los parámetros de la red creando otro estado en  $X$ . La acción es la reordenación de los valores de los parámetros indicada por la permutación  $g \in G$ . La condición 1. se satisface por la permutación identidad, que mapea cada estado de la red en sí mismo. La 2. se satisface ya que la aplicación entre permutaciones es asociativa.

**Definición .3.** Una  $G$ -órbita de  $x$ , [35] página 58, viene dada si  $X$  es un  $G$ -conjunto y  $x \in X$ , por:

$$\mathcal{O}(x) = \{gx : g \in G\} \subset X$$

Las  $G$ -órbitas que nos interesan son los subconjuntos de  $X$  creados por la aplicación de todas las permutaciones que cambian neuronas  $g \in G$  a todos los estados  $x \in X$ . Estos subconjuntos crean una partición de  $X$ , cada una conteniendo los estados que producen la misma hipótesis. ¿Cuántas  $G$ -órbitas hay en  $X$ ?

**Teorema .2.** *Lema de Burnside, [35] página 58, si  $X$  es un  $G$ -conjunto finito y  $N$  es el número de  $G$ -órbitas de  $X$ , entonces*

$$N = (1/|G|) \sum_{\tau \in G} F(\tau)$$

donde para  $\tau \in G$ ,  $F(\tau)$  es el número de  $x \in X$  fijado por  $\tau$

Hemos visto que cuando  $V$  es finito, el conjunto de los estados de la red  $X$  es un conjunto finito, y es un  $G$ -conjunto con las permutaciones de los parámetros de la red resultando en un cambio del orden de la suma de las neuronas output en las capas de la red, donde  $|G| = \prod_l U_l!$ .  $N$  es el número de  $G$ -órbitas en  $X$  creado por las acciones de las permutaciones de  $G$ , y, por lo tanto, es el número de mapeos de funciones únicas que una red puede producir. El último dato que necesitamos evaluar es  $F(\tau)$  para poder conseguir  $N$ .

En nuestro contexto  $F(\tau)$  especifica cuantos estados únicos una permutación  $\tau \in G$  de  $W$  elementos puede crear cuando todas las elecciones posibles  $w_{ij}$  para los  $W$  elementos son consideradas. La solución viene dada por el siguiente lema.

**Teorema .3.** *Sea  $\mathcal{V}$  un conjunto con  $|\mathcal{V}| = V$ , y sea  $G$  el subconjunto de todas las posibles permutaciones de  $W$  elementos. Si  $\tau \in G$ , entonces  $F(\tau) = V^{t(\tau)}$ , donde  $t(\tau)$  es el número de ciclos en la factorización completa de  $\tau$ .*

Cada permutación puede ser expresada como factor de ciclos disjuntos. Por ejemplo, una permutación escrita como  $(1,2)(3,4,5)(6)(7)$  denota el siguiente reordenamiento de 7 elementos en 4 ciclos:

- el elemento 2 se intercambia con el elemento 1;
- el elemento 3 va al lugar del 4, el 4 va al lugar del 5, y este último va al lugar del 3;
- el elemento 6 es fijo;
- el elemento 7 es fijo.

Por el anterior lema  $F(\tau) = V^{t(\tau)}$ , donde  $t(\tau)$  es el número de ciclos, y la suma del lema de Burnside será fijada por la permutación  $\tau \in G$  con el mayor número de ciclos. Para una permutación de  $W$  elementos, el mayor número posible de ciclos es  $t(\tau) = W$ , y viene dado por la identidad,  $\tau = e$ . Por lo tanto, conforme  $W$  aumenta, tenemos

$$N = \mathcal{O}(V^{t(e)} / |G|) = \mathcal{O}(V^W / \prod_l U_l!)$$

Dado que el conjunto  $X$  tiene  $N = \mathcal{O}(V^W / \prod_l U_l!)$   $G$ -órbitas con respecto a todas las combinaciones de permutaciones de neuronas en todas las redes neuronales, tenemos un límite superior en el número de funciones que una red neuronal con una arquitectura específica puede generar. Por lo tanto  $|\mathcal{H}| \leq \mathcal{O}(V^W / \prod_l U_l!).$

Esta cota puede reducirse dependiendo de la elección de la función de activación  $\sigma$  y un conjunto de parámetros  $\mathcal{V}$ . Otros métodos como la evaluación numérica pueden hacer uso de ciertas simetrías para mejorar la cota, pueden consultarse en [41] en el apartado 3.2

## Problemas de Gradiente e Inicialización Xavier

Los problemas de gradiente, tanto el de desvanecimiento como el de explosión, han sido enunciados varias veces durante la memoria pero, ¿a qué se deben?

Consideremos el ejemplo dado en [17], sea la siguiente red neuronal de 9 capas [11],

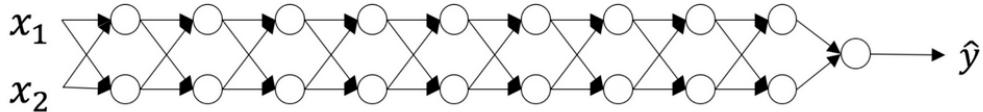


Figura 11: Representación gráfica de red neuronal de 9 capas

En cada iteración del bucle de optimización (avance, cálculo de coste, retropropagación, actualización) podemos observar que los gradientes retropropagados son amplificados o minimizados conforme avanzas desde la capa output a la capa input. Esto tiene sentido considerando el siguiente ejemplo;

Sean todas las funciones de activación lineales (función identidad). Entonces la activación del output viene dada por:

$$\hat{y} = a^{[L]} = W^{[L]}W^{[L-1]} \dots W^{[1]}x$$

con  $L = 10$  y  $W^{[1]}, W^{[L-1]}$  todas matrices de tamaño  $(2,2)$ , ya que todas las capas tiene 2 neuronas y reciben 2 inputs. Con esto en mente, si asumimos que  $W^{[L]} = \dots = W^{[L-1]} = W^{[1]} = W$  la predicción del output será  $\hat{y} = W^{[L]}W^{L-1}x$  donde  $W^{L-1}$  eleva la matriz  $W$  a  $L-1$ , mientras que  $W^{[L]}$  denota la L-ésima matriz. Veamos el resultado de inicializar valores muy pequeños, muy grandes y apropiados.

Si la inicialización toma valores muy grandes lleva a problemas de "explosión" de gradiente, consideremos el caso en el que cada coeficiente es inicializado de forma un poco mayor que la identidad, por ejemplo:

$$W^{[1]} = \dots = W^{[L-1]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

esto puede simplificarse a  $\hat{y} = W^{[L]}1.5^{L-1}x$ , y los valores de  $a^{[L]}$  crecen exponencialmente con  $L$ . Cuando estas activaciones son usadas en la retropropagación llevan a un problema de explosión de gradiente. Esto es, el gradiente del coste con respecto al parámetro es demasiado grande. Esto lleva al coste a oscilar alrededor de su valor mínimo.

Si la inicialización toma valores muy pequeños lleva a problemas de "desvanecimiento" de gradiente, consideremos el caso en el que cada coeficiente es inicializado de la siguiente forma:

$$W^{[1]} = \dots = W^{[L-1]} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

esto puede simplificarse a  $\hat{y} = W^{[L]} 0.5^{L-1} x$  y los valores de  $a^{[L]}$  decrecen exponencialmente con  $l$ . Cuando estas activaciones son usadas en la retropropagación llevan a un problema de desvanecimiento de gradiente. Esto es, el gradiente del coste con respecto al parámetro es demasiado pequeño. Esto lleva al coste a converger antes de alcanzar su valor mínimo.

La inicialización de Xavier [23] es un método diseñado para combatir los problemas de gradiente usado durante la implementación de AlphaZero y consiste en inicializar los coeficientes de tal forma que la varianza de los outputs de cada neurona es la misma que la varianza de sus inputs. Esto se realiza fijando los coeficientes iniciales a valores de una distribución con media 0 y una varianza específica que es determinada por el número de inputs y outputs de una neurona.

La inicialización recomendada de Xavier viene dado por la siguiente expresión para cada capa  $l$ :

$$W^l \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}})$$

$$b^{[l]} = 0$$

todos los coeficientes de la capa  $l$  son escogidos aleatoriamente de una distribución normal con media  $\mu = 0$  y varianza  $\sigma^2 = \frac{1}{n^{[l-1]}}$  con  $n^{[l-1]}$  el número de neuronas en la capa  $l - 1$ . Los biases son inicializados a cero.

Veamos cómo mantiene la varianza en cada capa

*Demostración.* Asumiremos que las activaciones de nuestras capas siguen una distribución normal alrededor del cero. Trabajaremos en la capa  $l$ -ésima, cuya propagación hacia delante es la siguiente:

$$a^{[l-1]} = g^{[l-1]}(z^{[l-1]})$$

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

y asumiremos que la función de activación es tanh. Por lo que la propagación hacia delante quedaría tal que así:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = \tanh(z^{[l]})$$

El objetivo es forzar una relación entre  $Var(a^{[l-1]})$  y  $Var(a^{[l]})$ . De esta forma sabremos cómo inicializar nuestros coeficientes para que  $Var(a^{[l-1]}) = Var(a^{[l]})$ .

Asumiremos que inicializamos nuestra red con valores apropiados y que el input está normalizado. En el comienzo del entrenamiento nos encontramos en el rango en el que  $\tanh$  se comporta linealmente, alrededor del 0. Los valores son lo suficientemente pequeños, por lo que  $\tanh z^{[l]} \approx z^{[l]}$ , por lo que:

$$Var(a^{[l]}) = Var(z^{[l]})$$

Además,  $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} = \text{vector}(z_1^{[l]}, \dots, z_{n^{[l]}}^{[l]})$ , donde  $z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]} + b_k^{[l]}$ . Por simplicidad asumiremos que  $b^{[l]} = 0$  (que terminará siendo verdad por la elección de inicialización que tomaremos). Dicho esto, teniendo en cuenta que  $Var(a^{[l-1]}) = Var(a^{[l]})$  tenemos:

$$Var(a^{[l]}) = Var(z^{[l]}) = Var(\sum_{j=1}^{n^{[l-1]}} w_{kj}^{[l]} a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]} a_j^{[l-1]})$$

Además, ya que  $Var(XY) = E[X]^2Var(Y) + Var(X)E[Y]^2 + Var(X)Var(Y)$ , si tomamos  $X = w_{kj}^{[l]}$  y  $Y = a_j^{[l-1]}$ , tenemos:

$$Var(w_{kj}^{[l]} a_j^{[l-1]}) = E[w_{kj}^{[l]}]^2 Var(a_j^{[l-1]}) + Var(w_{kj}^{[l]}) E[w_{kj}^{[l]}]^2 + Var(w_{kj}^{[l]}) Var(a_j^{[l-1]})$$

Pero, ya que los coeficientes fueron inicializados con media cero, y los inputs están normalizados, llegamos a:

$$Var(z_k^{[l]}) = \sum_{j=1}^{n^{[l-1]}} Var(w_{kj}^{[l]}) Var(a_j^{[l-1]}) = \sum_{j=1}^{n^{[l-1]}} Var(W^{[l]}) Var(a^{[l-1]}) = \\ n^{[l-1]} Var(W^{[l]}) Var(a^{[l-1]})$$

Esta igualdad sale de  $Var(w_{kj}^{[l]}) = Var(w_{11}^{[l]}) = \dots = Var(W^{[l]})$  y de  $Var(a_j^{[l-1]}) = Var(a_1^{[l-1]}) = \dots = Var(a^{[l-1]})$ , y con la misma idea llegamos a  $Var(z^{[l]}) = Var(z_k^{[l]})$

En resumen, llegamos a:

$$Var(a^{[l]}) = n^{[l-1]} Var(W^{[l]}) Var(a^{[l-1]})$$

Por lo que si queremos que la varianza no varíe en las capas ( $Var(a^{[l]}) = Var(a^{[l-1]})$ ), necesitamos que  $Var(W) = \frac{1}{n^{[l-1]}}$ .

En los pasos previos no hemos especificado en ningún momento una capa  $l$  específica. Por esto, esta expresión se cumple para cada una de ellas. Sea  $L$  la capa output de nuestra red. Usando esta expresión en cada capa, podemos vincular la varianza de la capa output a la varianza de la capa input:

$$\begin{aligned} Var(a^{[L]}) &= n^{[L-1]} Var(W^{[L]}) Var(a^{[L-1]}) \\ &= n^{[L-1]} Var(W^{[L]}) n^{[L-2]} Var(W^{[L-1]}) Var(a^{[L-2]}) \\ &\quad \dots \\ &= [\prod_{l=1}^L n^{[l-1]} Var(W^{[l]})] Var(x) \end{aligned}$$

Dependiendo de cómo inicialicemos los coeficientes, la relación entre la varianza de nuestro output e input variará drásticamente. Veamos los 3 posibles casos:

$$n^{[l-1]} Var(W^{[l]}) = \begin{cases} < 1 \rightarrow \text{Desvanecimiento de señal} \\ = 1 \rightarrow Var(a^{[L]}) = Var(x) \\ > 1 \rightarrow \text{Explosión de señal} \end{cases}$$

Por lo que, para evitar el desvanecimiento o explosión de la señal propagada hacia delante, debemos fijar  $n^{[l-1]} Var(W^{[l]}) = 1$  inicializando  $Var(W^{[l]}) = \frac{1}{n^{[l-1]}}$ .

Por esta justificación vemos el razonamiento en la propagación hacia delante. El mismo resultado puede ser calculado para gradientes en la retropropagación. Haciéndolo de la misma forma llegaremos a que, para evitar el problema de desvanecimiento y explosión de gradiente, hemos de fijar  $n^{[l]} Var(W^{[l]}) = 1$  inicializando  $Var(W^{[l]}) = \frac{1}{n^{[l]}}$ .

Uno de los principales beneficios de esta inicialización es que puede acelerar el proceso de aprendizaje. Al fijar los coeficientes iniciales a valores óptimos, puede ayudar a que la red neuronal converja más rápido, esto en particular es beneficioso en modelos de aprendizaje profundo, que son computacionalmente costosos. Otro beneficio es que al asegurar la misma varianza en los outputs de las neuronas que en los inputs, puede ayudar a que el gradiente se mantenga en rangos manejables, haciendo que la red sea más fácil de entrenar.

Su principal limitación es que asume que la función de activación es lineal, por lo que para funciones como ReLU se recomiendan otras inicializaciones como la inicialización He, de la que hay más información en [26].