Aprendizaje por refuerzo: Fundamentos teóricos del algoritmo AlphaZero e implementación

Reinforcement learning: Theoretical foundations of the AlphaZero algorithm and implementation

Autor
Alberto Maurel Serrano

Tutores
Miguel Palomino Tarjuelo
José Alberto Verdejo López

Trabajo de Fin de Grado del Doble Grado en Ingeniería Informática - Matemáticas

Facultad de Informática, Universidad Complutense de Madrid

Curso 2020 - 2021

Resumen

En 2016, el equipo de DeepMind sorprendió al mundo creando una inteligencia artificial capaz de jugar al go a un nivel superior al de los humanos y ganando a uno de los jugadores más laureados de la historia. Sin embargo, AlphaGo era un algoritmo complejo y requería de una gran potencia computacional.

Un año más tarde se publicó AlphaZero. La belleza de este algoritmo residía no solo en que requería menos potencia computacional y se podía aplicar a más juegos, sino en la elegancia con la que combinaba sus componentes para lograr un rendimiento por encima de cualquier otro algoritmo hasta el momento.

El objetivo de este trabajo es explicar el funcionamiento del algoritmo AlphaZero. Para ello se introducen primero las nociones teóricas básicas del aprendizaje por refuerzo y las redes neuronales y posteriormente los detalles particulares del algoritmo. Además, se implementa una versión reducida del mismo y se entrena para jugar al tres en raya y al Conecta 4, estudiándose los resultados obtenidos.

Palabras clave

- AlphaZero
- AlphaGo Zero
- DeepMind
- Aprendizaje por refuerzo
- Árboles de búsqueda de Monte Carlo
- Tres en raya
- Conecta 4

Abstract

In 2016, DeepMind's team surprised the world by crafting an artificial intelligence that was able to play Go at a superhuman level and win the second most laureate Go player in history. However, AlphaGo was a complex algorithm, that required huge computing power.

A year later AlphaZero was published. The beauty behind this algorithm relies not only on the smaller computing power required or that it can be applied to more board games but also on the way they skillfully put together its components to achieve a performance way better than other Go programs at that moment.

The objective of this work is to explain how AlphaZero works. First, we briefly introduce the theoretical basis of reinforcement learning and neural networks and later we explain the details of the algorithm. In addition, a slightly simplified version of the algorithm is implemented and trained to play Tic Tac Toe and Connect 4, and its performance is analyzed.

Keywords

- AlphaZero
- AlphaGo Zero
- DeepMind
- Reinforcement learning
- Monte Carlo Tree Search (MCTS)
- Tic Tac Toe
- Connect 4

Índice

Página

Agradecimientos

Muchas gracias a Miguel y a Alberto por sus incontables correcciones y sus acertados comentarios. Sin ellos habría sido imposible que el trabajo hubiera llegado a buen puerto.

Muchas gracias también a Marta y a mi familia por todo su apoyo durante todos estos años, y especialmente por ofrecerse a jugar contra el algoritmo aunque perdiesen una y otra vez.

Por último, muchas gracias a mi ordenador por realizar simulaciones hasta caer rendido.

1. Introducción

1.1. Motivación

En el año 1996 el ordenador Deep Blue (supercomputador diseñado por IBM específicamente para jugar al ajedrez) se enfrentó por primera vez a Kasparov, y aunque perdió el encuentro 2 a 4, fue capaz de ganar una partida. Un año más tarde, en 1997, se celebró otro encuentro entre ambos en el que Deep Blue sí fue capaz de vencer. Esto supuso un hito porque era la primera vez que un ordenador derrotaba al campeón del mundo de ajedrez, un juego de considerable dificultad.

Con el juego del go había sucedido algo distinto. Pese a tener solo un tipo de fichas, tanto la cantidad de posiciones como la cantidad de movimientos posibles en cada jugada hacían que el espacio de exploración del go (número de líneas de juego que se pueden seguir a partir de un estado dado) fuese inmensamente superior al del ajedrez. Por ello, hasta la aparición del algoritmo AlphaGo en 2016 los programas de go eran incapaces de ganar a jugadores profesionales.

El equipo de DeepMind logró, uniendo los árboles de búsqueda de Monte Carlo utilizados por todos los programas de go en el momento con una innovadora aproximación empleando redes neuronales y aprendizaje por refuerzo, crear un programa que fuese capaz de jugar al go al máximo nivel. Su primer encuentro con un jugador profesional fue contra Fan Hui, 3 veces campeón de go en Europa, en octubre de 2015. AlphaGo lo ganó 5-0, siendo el primer encuentro en el que una inteligencia artificial ganaba a un jugador profesional en igualdad de condiciones. No obstante, el verdadero desafío llegó en marzo de 2016, cuando AlphaGo se enfrentó a Lee Sedol, uno de los mejores jugadores del mundo en el momento, y el segundo de la historia con más títulos internacionales. En este encuentro AlphaGo venció 4-1, cediendo una sola partida tras un *movimiento divino* de Sedol.

El encuentro generó una increíble expectación a nivel mundial y supuso un importante punto de inflexión, puesto que no se esperaba que las máquinas superasen a los humanos en el go tan pronto. Desde entonces, han aparecido varias versiones del algoritmo, mejorando notablemente no solo su rendimiento sino también simplificándolo y haciéndolo más general, incluyendo a AlphaZero.

1.2. Objetivos y plan de trabajo

Los objetivos de este trabajo son dos. El primero de ellos es estudiar el algoritmo AlphaZero desde un punto de vista teórico. Para ello se comienza en el capítulo 2 definiendo el marco teórico del aprendizaje por refuerzo y brevemente las redes neuronales, dos de los pilares sobre los que se asienta el algoritmo. El capítulo 3 será la sección principal del trabajo, en la que abordaremos los detalles teóricos de AlphaZero, extraídos del artículo en el que se define el algoritmo. En este capítulo también se hablará brevemente de las diferencias de AlphaZero respecto al resto de algoritmos pertenecientes a su misma familia.

El segundo objetivo del trabajo es observar el comportamiento del algoritmo desde el punto de vista práctico. Para ello, se implementa una versión ligeramente simplificada de AlphaZero y se analizan los detalles más importantes de la implementación en el capítulo 4. Por último, en el capítulo 5 se entrena esa implementación para jugar al tres en raya y al Conecta 4 y se analizan los resultados obtenidos.

Introduction

Motivation

In 1996, Deep Blue (a supercomputer designed by IBM specifically to play chess) faced Kasparov for the first time. Although it lost 2 to 4, Deep Blue was able to win one game. One year after, in 1997, another match was played between them and then Deep Blue was able to achieve the victory. This victory was a landmark in computer science research because it was the first time a computer was able to defeat a chess world champion, a game with sizeable complexity.

However, the game of Go was substantially different. Although in Go there is just one type of piece, the number of different boards and possible movements made Go's game tree (number of games that can be played from a certain state) much larger than chess' one. Due to that, until the appearance of AlphaGo in 2016 computer programs were unable to win professional players.

DeepMind's team was able to build an algorithm that played Go at a superhuman level by putting together the Monte Carlo Trees used by all the contemporary programs, a neural network and reinforcement learning. Its first match against a professional player was against Fan Hui, 3 times Go European champion, in October 2015. AlphaGo won the match 5-0, and that was the very first time an artificial intelligence won a match against a professional player without any handicap. However, the real challenge was held in March 2016, when AlphaGo played against Lee Sedol, one of the best players in the world at that moment and the second player with more international titles in Go's history. AlphaGo won the match 4 to 1 losing just one game after a *divine movement* by Sedol.

This match generated an incredible expectation worldwide and was an important landmark in AI research because experts did not expect that computers would be able to defeat professional players in Go so early. Since then, several versions of the algorithm have appeared, improving significantly not only its performance but also making it more simple and more general, including AlphaZero.

Objectives and work plan

This dissertation has two main objectives. The first one is to study the AlphaZero algorithm from a theoretical point of view. Firstly, in Chapter 2 we define the theoretical framework of reinforcement learning and briefly the neural networks, two fundamental pillars the algorithm relies on. Chapter 3 will be the crucial part of the dissertation. On it, we will discuss the theory behind AlphaZero, based on the paper where the algorithm was proposed. In this chapter, we will also talk briefly about the differences between AlphaZero and the other variants of the algorithm.

The second objective is to observe how the algorithm behaves when implemented. To achieve this, a slightly simplified version of the algorithm is implemented. The crucial details of this implementation will be discussed in Chapter 4. Lastly, in Chapter 5 we teach the algorithm to play Tic Tac Toe and Connect 4, and we analyze the results achieved.

2. Fundamentos teóricos

2.1. Aprendizaje por refuerzo

2.1.1. Marco teórico

El aprendizaje por refuerzo es una de las tres grandes ramas que existen en el aprendizaje automático. Por un lado tenemos el aprendizaje supervisado, en el que a partir de un conjunto de datos etiquetados el sistema trata de entender por qué cada ejemplo pertenece a esa clase y predecir las etiquetas de nuevos ejemplos sin etiquetar. En el aprendizaje no supervisado, se proporcionan ejemplos sin etiquetar y se busca una estructura subyacente que permita separarlos en grupos.

En el aprendizaje por refuerzo la aproximación es completamente diferente. Vamos a tener un *agente* que va a interactuar con un *entorno* por medio de *acciones*. En cada momento, el entorno va a estar en un *estado*, el cual se va a ver modificado por las acciones realizadas. Además, cada acción va a proporcionar al agente una *recompensa* y su objetivo es maximizar la recompensa total obtenida. Para explicar la teoría subyacente al aprendizaje por refuerzo se utilizará el capítulo 17 de *Foundations of Machine Learning* [11] y el capítulo 3 de *Reinforcement Learning: An Introduction* [23].

Una primera diferencia del aprendizaje por refuerzo respecto a las dos otras aproximaciones es que, en estas, nosotros proporcionamos los ejemplos. Sin embargo, en el aprendizaje por refuerzo es el propio agente el encargado de generarlos por medio de interacciones con el entorno. Esto provoca que el proceso de obtención de datos de entrenamiento se intercale con el propio entrenamiento.

Por otro lado, el hecho de que el propio agente sea el que recabe los datos genera el conflicto conocido como explotación contra exploración. Por un lado, el sistema ya ha explorado una cierta parte del entorno y en base al conocimiento obtenido tiene una "mejor estrategia", aquella que le reporta una mayor recompensa. Sin embargo, puede existir una parte del entorno aún desconocida y que al explorarla encontremos una estrategia mejor. El agente tiene que equilibrar el explotar la mejor estrategia hasta el momento con explorar en busca de otras estrategias mejores.

Para formalizar el marco en el que se desarrolla la teoría se utilizan los procesos de decisión de Markov (MDP por sus siglas en inglés). Tenemos los siguientes elementos:

- Un conjunto de estados $\mathcal{S}$, que contiene todos los posibles estados en los que se puede encontrar el entorno.
- Un estado inicial s_0, la situación inicial del entorno. Será la situación de la que partirá el agente.
- Un conjunto de acciones $\mathcal{A}$, con las distintas acciones que podemos realizar.
- Un conjunto de recompensas $\mathcal{R}$ que podemos recibir por las acciones realizadas.
- Para cada par de estado s y acción a, una distribución de probabilidades sobre los estados de destino. Es decir, una función $\delta : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ tal que:

$$\delta(s'|s, a) = \text{probabilidad de ir a } s' \text{ desde } s \text{ ejecutando la acción } a$$

Al ser una probabilidad, además verificará que:

$$\sum_{s' \in \mathcal{S}} \delta(s'|s, a) = 1$$

- Para cada par de estado s y acción a, una distribución de probabilidades sobre las recompensas. Es decir, una función $r : \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ tal que:

  $r(r'|s, a) =$ probabilidad de obtener una recompensa r' desde s ejecutando la acción a

  Y de nuevo se verifica:

$$\sum_{r' \in \mathcal{R}} r(r'|s, a) = 1$$

El proceso de decisión se denomina de Markov porque las funciones δ y r verifican la propiedad de Markov. Esta propiedad la poseen aquellas distribuciones de probabilidad que dependen únicamente del estado actual, pero no de estados anteriores. De esta forma, el estado ha de encapsular toda la información para identificar unívocamente el momento en el que nos encontramos.

Otro inciso importante es que se ha definido un marco muy general, en el que a pesar de que se realice dos veces la misma acción en el mismo estado se puede llegar a un estado diferente u obtener una recompensa distinta. Esto se debe a que tanto la función δ como la función r devuelven una distribución de probabilidad. Para el desarrollo que vamos a realizar no se necesita tanta generalidad, puesto que desde un estado tras ejecutar una acción se irá siempre al mismo estado y nos proporcionará la misma recompensa. Por ello, se puede sobrecargar la notación para simplificar las funciones δ y r, dado que desde el estado s efectuando la acción a se pasará siempre al estado s' y se obtendrá recompensa r':

$$\delta(s, a) = s'$$
$$r(s, a) = r'$$

Por otro lado, vamos a considerar un modelo discreto. Esto significa que las decisiones se van a tomar en una serie de instantes temporales $\{0, \ldots, T\}$, y es una elección natural dado que nosotros vamos a trabajar con juegos de tablero en los que hay que tomar las decisiones en ciertos puntos de la simulación (y mientras el jugador está pensando no se modifica el entorno). El valor T se conoce como horizonte, y puede ser finito o infinito.

Todo lo anterior hace referencia al entorno, es decir, a cómo se va a comportar el medio con el que interactúa el agente. Para modelizar el comportamiento del agente empleamos la *política*. Una política es una función $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, siendo $\Delta(\mathcal{A})$ el conjunto de distribuciones de probabilidad sobre $\mathcal{A}$. Esto significa que para cada estado s tendremos una distribución de probabilidades $\pi(s)$ que nos indicará cuál es la probabilidad con la que el agente realizará cada acción en dicho estado. Una política es determinista cuando para todo s hay una única a tal que $\pi(s)(a) = 1$ (siempre realizamos la misma acción desde cada estado).

Todo esto nos lleva a la siguiente construcción, que se encuentra descrita en la figura 1:

1. El entorno comienza en el estado s_0.

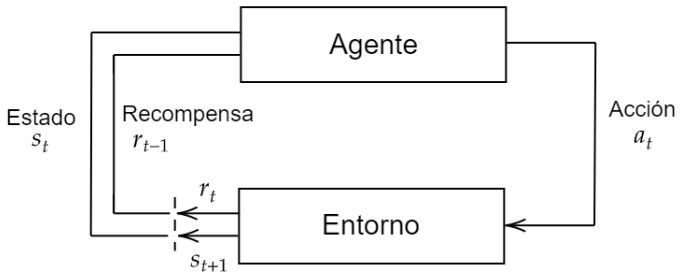


Figura 1: Funcionamiento de un proceso de decisión de Markov. Basado en la figura 3.1 de [23].

2. El agente selecciona la acción a realizar en base a la distribución de probabilidad $\pi(s_0)$.
3. El agente realiza la acción a_0, recibe por ella la recompensa r_0 y pasa al estado s_1.
4. Repetimos el procedimiento hasta el instante T, generando la trayectoria:

$$s_0 \;\; a_0 \;\; r_0 \;\; s_1 \;\; a_1 \;\; r_1 \;\; s_2 \;\; \cdots$$

Al introducir el concepto de política hemos hablado de la política que sigue el agente. Sin embargo, como está aprendiendo al mismo tiempo que interactúa con el entorno, es habitual que esta política sea dinámica, es decir, se modifique con el paso del tiempo con el objetivo de obtener la mayor recompensa posible. Las anteriores eran políticas estacionarias, puesto que no dependían del instante temporal. Cuando se emplea una política no estacionaria pasa a estar parametrizada por una t, indicando el instante actual:

$$\pi_t : \mathcal{S} \to \Delta(\mathcal{A})$$

Una cuestión interesante es por qué utilizamos distribuciones de probabilidad para modelizar el comportamiento de un agente si cuando este se enfrente al problema real solamente va a poder escoger una de las acciones. Esto se emplea para poder ir modificando la política poco a poco y explorar el espacio de soluciones de forma más gradual. Si en el instante de tiempo t la política π_t indica que en el estado s siempre hay que realizar una acción y la política π_{t+1} indica que en ese mismo estado siempre se realiza otra, la modificación de la política es muy brusca. Pese a que en unos pocos ejemplos hayamos visto que esta acción es mejor, puede que si tenemos en cuenta todas las posibilidades esta acción no lo sea, y en el fondo hayamos empeorado la política.

Si por el contrario trabajamos con distribuciones de probabilidad los cambios que podemos hacer son mucho más suaves, ya que podemos hacer una acción ligeramente más probable que antes (en detrimento de otras). Evidentemente existe el mismo riesgo, que hagamos más probable una acción que conduzca a una recompensa menor. Sin embargo, puede que tras un empeoramiento de la política muy grande no seamos capaces de recuperarnos. Como ya dijimos anteriormente, en el aprendizaje por refuerzo es el propio agente el encargado de recabar los datos de entrenamiento. Si la política actual se centra en una región del espacio, es posible que no seamos capaces de salir

de esa región en la exploración. De ahí que sea muy importante no dar pasos excesivamente grandes y querer hacer tan solo ligeras modificaciones en cada paso[1].

Dependiendo de si el horizonte que utilizamos es finito o infinito, la recompensa total obtenida en una trayectoria se computa de diferente forma:

- En caso de tener un horizonte finito, la recompensa total será la suma de las recompensas obtenidas:

$$\sum_{t=0}^{T} r_t$$

  Partimos del estado s_0 y en cada paso realizamos la acción indicada por la política para el estado s_i[2], pasando al estado s_{i+1} y acumulando la recompensa.

- En el caso de tener un horizonte infinito, la suma de las recompensas podría ser infinita. Por ello se utiliza un descuento, un parámetro $\gamma \in [0, 1)$ que garantiza que la suma es finita cuando r_t está acotado:

$$\sum_{t=0}^{+\infty} \gamma^t \, r_t$$

  El descuento otorga además mayor importancia a las recompensas obtenidas más cerca del estado actual. Es algo razonable incluso en el caso de tener un horizonte finito, puesto que habitualmente las acciones más próximas a una recompensa son aquellas que han contribuido en mayor medida a ella. Supongamos que estamos en una partida de ajedrez donde recibimos una unidad de recompensa cada vez que ganamos. Las últimas jugadas son las responsables de la victoria. Para estos estados, la recompensa se sumará prácticamente íntegra porque el descuento estará elevado a una potencia pequeña. Sin embargo, las primeras jugadas de la partida, pese a que han contribuido a la victoria probablemente no lo han hecho tanto como las últimas. Para esos estados, la recompensa final tendrá menos peso porque el descuento estará elevado a una potencia mayor.

Una vez introducidos los conceptos de política y recompensa, tiene sentido introducir el concepto de *valor* de una política π a partir de un estado s, que es la recompensa total esperada si partimos desde s y en cada paso aplicamos la política π. La diferencia respecto al párrafo anterior es que antes suponíamos que teníamos una única trayectoria generada por nuestra política. Por ejemplo, que para cada estado realizábamos la acción a_t a la que la política otorgaba mayor probabilidad, es decir, que maximizaba $\pi(s_t)(a_t)$. Ahora vamos a pensar en todas las posibles trayectorias que se pueden tomar a partir de nuestra política y cuál es la probabilidad de cada una de ellas, y vamos a ponderar la recompensa obtenida en cada una de ellas por su probabilidad para obtener una media. De nuevo, distinguimos dependiendo del horizonte temporal:

[1] De hecho, en el aprendizaje por refuerzo existe el concepto de región de confianza (*trust region* en inglés). Es la región en la que podemos optimizar la política sin miedo a "optimizar demasiado", con el propósito de evitar estas actualizaciones destructivas.

[2] Como ya hemos dicho anteriormente, la política es realmente una distribución de probabilidad. Aquí nos estamos refiriendo a la acción escogida en base a dicha distribución de probabilidad.

- Horizonte finito:

$$V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{T} r(s_t, \pi(s_t)) \middle| s_0 = s \right]$$

- Horizonte infinito:

$$V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{T} \gamma^t \, r(s_t, \pi(s_t)) \middle| s_0 = s \right]$$

Utilizamos la esperanza matemática para calcular la recompensa esperada con el fin de contabilizar la recompensa de todas las trayectorias al mismo tiempo. Una forma más intuitiva de formularlo es en términos de trayectorias. Como ya introdujimos anteriormente, una trayectoria es una secuencia de estados y acciones que llevará aparejada una serie de recompensas. Dado que las funciones δ y r son deterministas, con conocer tanto s_0 como las acciones realizadas podemos describir una trayectoria.

Para escribir la función V en términos de trayectorias necesitamos conocer la probabilidad de cada trayectoria, puesto que de esto dependerá su "contribución" a la recompensa esperada total. Si denotamos con $P(\tau|\pi)$ la probabilidad de la trayectoria τ con la política π y estamos trabajando con un horizonte finito, entonces podemos calcular esta probabilidad como el producto de las probabilidades de las acciones en la trayectoria, es decir:

$$P(\tau|\pi) = \prod_{\tau = a_0, a_1, \cdots, a_T} \pi(s_i)(a_i)$$

Por otro lado, la recompensa de una trayectoria es la suma de las recompensas obtenidas a lo largo de la trayectoria:

$$R(\tau) = \sum_{\tau = a_0, a_1, \cdots, a_T} r(s_i, a_i)$$

Con esto podemos definir la función V como:

$$V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{T} r(s_t, \pi(s_t)) \middle| s_0 = s \right] = \sum_{\tau} P(\tau|\pi) R(\tau)$$

siendo τ todas aquellas trayectorias que comienzan en s_0. Esta definición nos da una mejor idea de qué es lo que queremos para la política: aumentar la probabilidad de aquellas trayectorias que nos aportan una recompensa mayor y decrementar la de aquellas con recompensas menores.

2.1.2. Algoritmos de aprendizaje por refuerzo

Dentro del aprendizaje por refuerzo encontramos diferentes familias de algoritmos en base a las características del problema que resuelven y cómo lo aproximan. Una clasificación interesante es

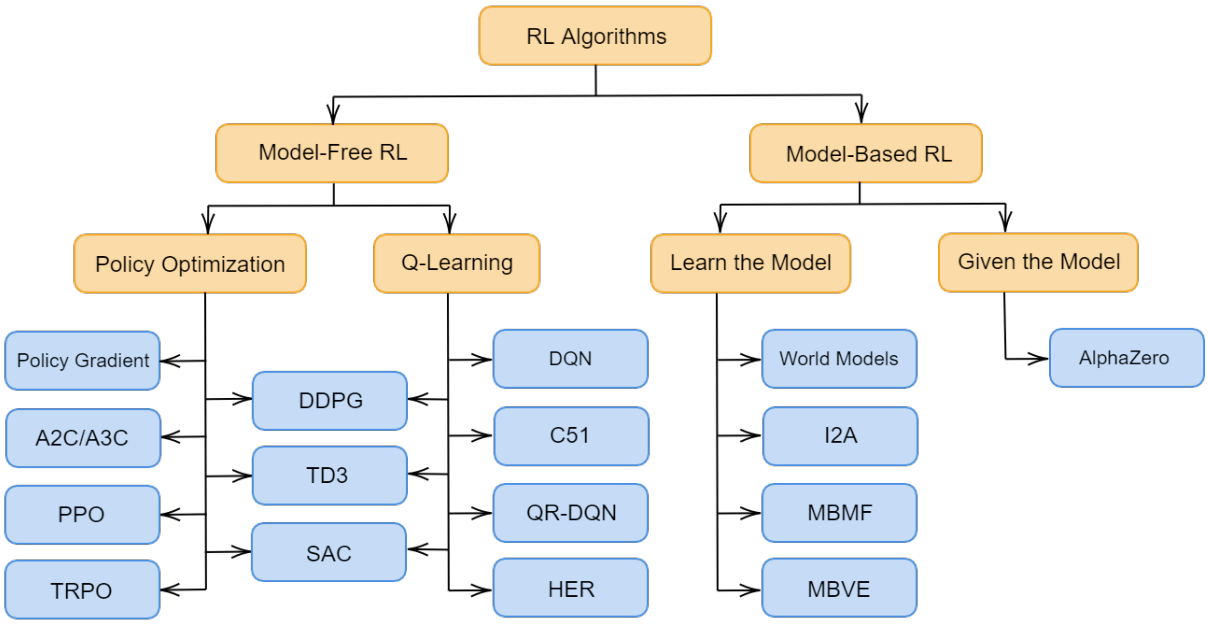


Figura 2: Taxonomía de los algoritmos de aprendizaje por refuerzo. Diagrama sacado de [12].

la que hacen en Spinning Up[3] [12], y que se puede ver en la figura 2. Como se dice en la propia fuente, es "*una clasificación no exhaustiva pero sí útil de los algoritmos de aprendizaje por refuerzo actuales*".

En la primera división diferencian entre algoritmos basados en el modelo (*model-based*) o no (*model-free*). El paradigma subyacente a estas dos vertientes es totalmente distinto. En los algoritmos basados en el modelo del entorno, una vez tengamos un buen modelo del entorno (es decir, tengamos información sobre las funciones de transición δ y r) vamos a emplearlo para planificar sobre él. Un ejemplo de ello es el algoritmo que vamos a estudiar, AlphaZero. En él sabemos exactamente cómo funciona el modelo y explotamos esta información estudiando qué movimiento es mejor probando múltiples variantes sobre nuestro modelo y quedándonos con la que conduzca a mejores resultados. Los algoritmos no basados en el modelo por el contrario no se preocupan en absoluto de construir un modelo del entorno. Se dedican a probar las acciones sobre el entorno y en base a las recompensas obtenidas ajustan la política y siguen realizando pruebas para mejorar la política. Un ejemplo de ellos es el algoritmo PPO[4], que sigue exactamente este patrón.

Dentro de los algoritmos basados en el modelo, se distinguen aquellos que tienen que aprender el modelo y aquellos que ya tienen el modelo. AlphaZero se encuentra en esta segunda división: nosotros le proporcionamos un modelo completo y exacto del entorno (que además ha de verificar otras condiciones que veremos más adelante) y AlphaZero trabaja sobre él.

[3]Esta es una página web con unos interesantísimos recursos de aprendizaje por refuerzo (especialmente aprendizaje por refuerzo profundo). Es propiedad de OpenAI, una de las principales entidades del sector.

[4]Este otro algoritmo será estudiado en mi TFG de Matemáticas: *Aprendizaje por refuerzo: Fundamentos matemáticos del algoritmo PPO e implementación.*

Los que tienen que aprender el modelo sin embargo son más versátiles. En juegos como el ajedrez o el go es sencillo codificar las reglas que rigen el juego. Sin embargo, en otros juegos, como pueden ser los de Atari[5], no es sencillo indicar cómo se modifica la imagen píxel por píxel al realizar cada acción. Un ejemplo de estos algoritmos es MuZero [16], una evolución de AlphaZero. En él, la parte de planificación es similar a la de AlphaZero, pero además permite aprender el entorno mediante la estimación de la función de recompensa. Sin embargo, en este trabajo estudiaremos AlphaZero porque ya poseemos un modelo exacto del juego con el que vamos a trabajar y en esta situación MuZero no representa una mejora.

2.1.3. Ejemplo de problema de aprendizaje por refuerzo

Para situar al lector vamos a enunciar un problema clásico de aprendizaje por refuerzo y a definirlo en términos de un proceso de decisión de Markov. El ejemplo 3.4 de [23] es el problema de equilibrado de una barra. Se tiene un carro con una barra en su parte superior. Esta barra se puede mover hacia la izquierda o hacia la derecha, y el objetivo es mantener la barra vertical durante el máximo tiempo posible, evitando que se caiga por el movimiento producido por el carro. En el OpenAI Gym aparece implementado bajo el nombre de CartPole. En la figura 3 se ilustra el problema.

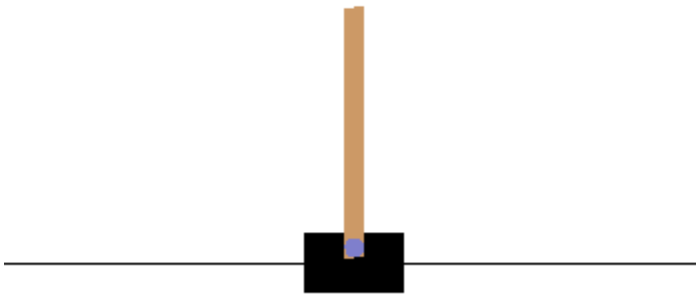


Figura 3: Fotograma sacado de la simulación de CartPole. El objetivo es mantener la barra sin que se caiga durante 500 pasos, obteniendo +1 de recompensa cada instante que no se termine la simulación. La simulación se termina antes de 500 pasos si la barra se inclina más de 15° respecto a la vertical o si el carro se sale de la pantalla.

En particular, en la implementación de OpenAI Gym el estado se describe con una tupla de cuatro elementos, indicando la posición y la velocidad del carro, y el ángulo y la velocidad de la barra. En el estado inicial el carro está aproximadamente en el centro, y la barra tiene un ligero ángulo y velocidad (diferente entre dos simulaciones distintas para no aprender una estrategia adaptada a un estado de inicio en particular). Las acciones posibles son mover el carro a la izquierda

[5]OpenAI Gym es un conjunto de entornos sobre los que se prueba habitualmente el funcionamiento de los algoritmos de aprendizaje por refuerzo [5]. Entre ellos se encuentran varios juegos de Atari y son comúnmente utilizados en las publicaciones, como [16, 17, 18], para comparar distintos algoritmos.

o a la derecha y se otorga +1 de recompensa por cada paso que mantengamos la barra en pie, hasta un máximo de 500 pasos. La limitación de pasos es un constructo para evitar que las ejecuciones sean infinitas, pero podemos extender de forma natural este problema a un número infinito de pasos. Por lo tanto:

- $\mathcal{S} = \{\text{tuplas de 4 valores permitidas}\} \cup \{\text{estado terminal}\}$.
- $s_0 =$ estado generado aleatoriamente, pero con el carro centrado y la barra casi vertical.
- $\mathcal{A} = \{izquierda, derecha\}$.
- $\mathcal{R} = \{1\}$.
- $\delta(s, a) =$ estado s' al que llegamos tras aplicar la acción a, calculado por la simulación.
- $r(s, a) = 1$.

La función de valor actual dependerá de la política actual. Si por ejemplo con la política actual π_t tenemos desde el estado actual s tres posibles trayectorias con probabilidad 0,6, 0,25 y 0,15, y mantienen la barra equilibrada durante 12, 15 y 14 pasos respectivamente, entonces

$$V_{\pi_t}(s) = 0{,}6 \cdot 12 + 0{,}25 \cdot 15 + 0{,}15 \cdot 14 = 13{,}05$$

Nótese la dependencia intrínseca entre política y valor. Este será uno de los puntos claves del aprendizaje por refuerzo que nos acompañarán a lo largo de todo el desarrollo de AlphaZero y tendrá suma importancia en su funcionamiento.

2.2. Redes neuronales

En esta sección se introducirán brevemente las ideas tras las redes neuronales. Para ello se utilizarán los libros *Neural Networks and Deep Learning: A Textbook* [1] y *Deep Learning* [9]. Los detalles teóricos concretos de la red neuronal utilizada en AlphaZero se comentan en el apéndice A.

Las redes neuronales reciben este nombre por estar inspiradas en el cerebro humano. Este está compuesto de neuronas, que reciben impulsos eléctricos de neuronas cercanas a través de las dendritas. Cuando la cantidad de estímulo supera un umbral, la neurona envía una señal eléctrica por su axón, comunicándose así con las neuronas a las que está conectada.

El objetivo de las redes neuronales es aprender a asociar cada dato (un vector de números) con una etiqueta (que puede ser la clase a la que pertenece el dato, un valor numérico asociado, etc). Para ello, nosotros le proporcionaremos múltiples ejemplos en los que aparece un dato y su etiqueta asociada, y la meta es que tras aprender la red neuronal sea capaz de predecir correctamente la etiqueta de datos que no ha visto previamente, utilizando el conocimiento adquirido.

En la red neuronal, la función de neurona la desempeña el perceptrón. Este va a recibir un dato $X = \{x_0, x_1, \ldots, x_n\}$ y va a producir una salida $\hat{y}$ con la predicción. Esta salida se calcula mediante el producto escalar XW, siendo $W = \{w_0, w_1, \ldots, w_n\}$ los pesos otorgados a cada dato de entrada. El objetivo del entrenamiento es aprender estos pesos, de forma que se seleccionen los

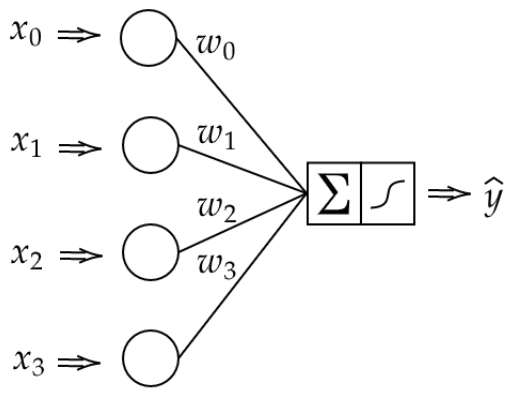


Figura 4: Esquema de un perceptrón para un dato con 4 componentes. El perceptrón toma el dato X y los pesos W y calcula $XW = \sum_{i=0,\dots,3} x_i w_i$. A este resultado le aplica una función no lineal para producir la salida $\hat{y}$. Figura basada en la figura 1.3(a) de [1].

pesos que logren mejores predicciones. Además, para poder aprender funciones más complejas tras el producto escalar se aplica una función no lineal. En la figura 4 se puede ver un esquema de un perceptrón.

Por ejemplo, supongamos que tenemos un juego entre dos jugadores cuyo estado viene definido por el vector $X = \{x_0, x_1, \dots, x_n\}$ y queremos saber a qué jugador favorece la posición (valor entre 1 si favorece al jugador que tiene que mover y -1 si favorece al oponente). Si utilizamos como función de activación la tangente hipebólica (que transforma $\mathbb{R}$ en $[-1, 1]$), entonces el valor predicho por nuestro perceptrón sería:

$$\hat{y} = \tanh \sum_{i=0}^{n} x_i w_i$$

Para entrenar nuestra red necesitaremos conocer el valor y que otorgamos a la posición X. Queremos que los valores y e $\hat{y}$ sean lo más similares posibles, para lo cual vamos a tratar de minimizar una función que refleje la diferencia entre estos dos valores. A esta función se la conoce como *función de pérdida*.

Para minimizarla vamos a utilizar una técnica conocida como descenso de gradiente. Esta técnica se basa en utilizar la dirección proporcionada por el signo de la derivada. Si suponemos que nuestra función de pérdida es f y nuestro dato X es unidimensional, entonces $f'(X) > 0$ si la función f es creciente en el punto actual y $f'(X) < 0$ si la función f es decreciente. Por lo tanto, para minimizar f simplemente tendremos que desplazarnos en la dirección contraria a la indicada por la derivada.

Por ejemplo, para nuestro problema de predecir el valor de una posición en un tablero, podemos utilizar como función de pérdida la diferencia al cuadrado, para penalizar más a los errores mayores:

$$L_W = (y - \hat{y})^2 = (y - \tanh(XW))^2$$

y en este caso la actualización de los pesos de nuestra red sería

$$W = W - \alpha \nabla L_W(X)$$

donde trabajamos con la noción de gradiente porque cada dato de entrada suele estar compuesto de varias componentes. Además aquí estamos aprendiendo de un solo ejemplo. Para el entrenamiento necesitaremos múltiples ejemplos con los que enseñar a nuestra red. Durante el entrenamiento, el conjunto de entrenamiento se divide en varios subconjuntos, llamados *batches*. Cada uno de estos subconjuntos se introduce en la red y se obtienen las etiquetas predichas. La actualización se realiza de la misma forma que habíamos visto para un único dato, con la salvedad de que la función de pérdida se escoge para tener en cuenta varios datos. Por ejemplo, utilizando

$$L_W = \sum_{(X,y) \in D} (y - \hat{y})^2$$

y siendo D el conjunto de datos de entrenamiento.

El valor α se denomina ratio de aprendizaje. Ratios mayores harán que la red aprenda más "rápido", es decir, tenga más en cuenta en la actualización los nuevos ejemplos, mientras que ratios más pequeños harán que los nuevos ejemplos introducidos influyan menos. Si pensamos en términos de la derivada, esta nos proporciona una dirección de mejora local si α es lo suficientemente pequeño, por lo que en la práctica nuestro objetivo es encontrar un ratio lo suficientemente grande para que la red aprenda rápido pero al mismo tiempo lo suficientemente pequeño para garantizar la mejora.

Este proceso de entrenamiento se realiza para todos los subconjuntos en los que hemos dividido los datos (es decir, para cada *batch* se obtienen las predicciones y se actualiza solo tras procesar todos los datos de cada *batch*), y se denomina época (*epoch* en inglés) a cada iteración de entrenamiento. Lo normal es realizar varias épocas, las suficientes para que nuestro algoritmo incorpore el conocimiento proporcionado por los ejemplos pero no demasiadas para que no se produzca sobreaprendizaje (*overfitting* en inglés). Podría ocurrir que nuestro ejemplo, en vez de fijarse en las características de nuestro dato y tratar de relacionarlo con la etiqueta simplemente aprenda para cada dato la etiqueta que le corresponde. Esto evidentemente minimiza el error cometido, pero cuando presentamos a la red un nuevo ejemplo desconocido, esta no es capaz de clasificarlo porque realmente no está encontrando las características que indican la pertenencia de una muestra a una clase en particular.

Al igual que en el cerebro una única neurona es poco útil, una red neuronal con un único perceptrón no tiene mucho poder de clasificación. La idea es utilizar múltiples perceptrones para aumentar las relaciones que son capaces de identificar. Para ello, colocamos los perceptrones en capas, de forma que las neuronas de una capa van a estar conectadas con las de la capa anterior y siguiente. La estructura de una red neuronal multicapa se puede ver en la figura 5.

Para entrenar las redes neuronales multicapa el proceso es más complejo que para un único perceptrón. Esto se debe a que la pérdida de una capa es una función de la pérdida en capas anteriores. Por lo tanto, al computar el gradiente tendremos que utilizar la regla de la cadena, complicando los cálculos. El proceso de entrenamiento en redes multicapa se denomina retropropagación (*backpropagation* en inglés) y básicamente consta de dos pasos:

- Fase de avance: se introducen los ejemplos de entrenamiento en la red y utilizando los pesos

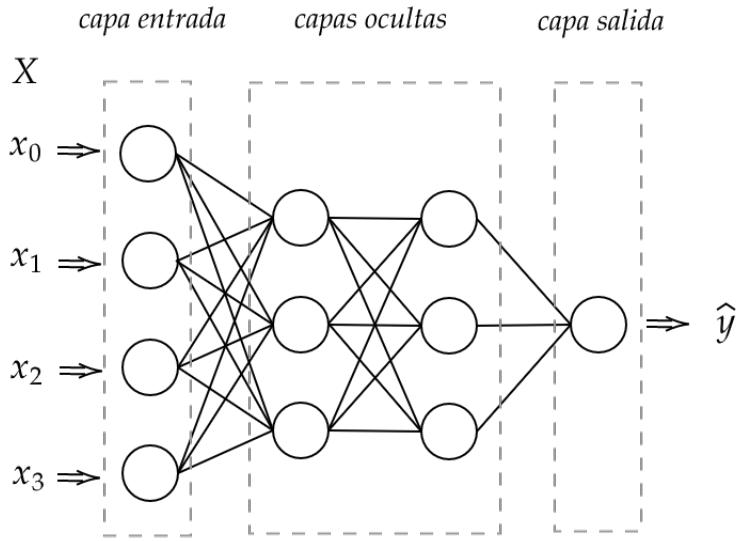


Figura 5: Ejemplo de red neuronal multicapa. La red comienza con la capa de entrada, con una neurona por cada dato de la entrada (en este caso, nuestro dato X tiene 4 componentes). Tras ello vienen varias capas ocultas (2 en este caso) con un número de neuronas escogidas por nosotros (en este caso 3 cada una, pero podría ser un número diferente, incluido un número diferente en cada capa). Por último tenemos la capa de salida, con tantas neuronas como datos queramos calcular (en este caso solo calculamos un valor, por lo que tenemos una única neurona). Las conexiones con líneas simples son las conexiones entre capas, y serán las conexiones para las que queremos aprender los pesos. Esto significa que entre la capa de entrada y la primera capa oculta aprenderemos $4 \times 3 = 12$ pesos, entre las capas ocultas $3 \times 3 = 9$ pesos y entre la segunda capa oculta y la capa de salida $3 \times 1 = 3$ pesos, para un total de 24 pesos en toda la red. Figura basada en la figura 1.11(c) de [1].

actuales se calcula la etiqueta final predicha y los valores en todas las neuronas intermedias.

- Fase de propagación: desde la última capa se calcula el gradiente de la función de pérdida (utilizando la regla de la cadena) y se actualizan los pesos conforme al gradiente.

Por último, para este trabajo son importantes dos tipos de redes neuronales en particular: las redes convolucionales y las residuales. Las redes convolucionales están especialmente concebidas para trabajar con datos con más de una dimensión que poseen una cierta estructura, como es el caso de una imagen. Si nosotros utilizamos una imagen como entrada de las redes neuronales que acabamos de ver (representada como una matriz con el valor de un píxel en cada posición), tendremos que "aplanar" la imagen. Concatenaremos cada una de las filas de la matriz para crear un vector unidimensional, que será el que pasemos a la red neuronal para que realice la predicción.

Sin embargo, con este proceso estaremos perdiendo la estructura de la imagen. En una imagen los píxeles adyacentes están relacionados. Al aplanar la imagen preservamos la adyacencia en horizontal, pero no en vertical ni en diagonal. Es cierto que la red podría aprender esta noción de adyacencia en el vector, a pesar de que estén separados los valores, pero es mucho más sencillo que aprenda a reconocer estos patrones adyacentes si le proporcionamos los datos con la estructura original. Las redes convolucionales aprovechan esta idea utilizando operaciones sobre áreas de píxeles.

Por otro lado, las redes residuales permiten incrementar la profundidad de la red mediante el uso de conexiones adicionales entre capas. Ambas redes se describen con más detalle en el apéndice A.

2.3. Reglas del go

Pese a no ser necesario saber jugar al go para entender el funcionamiento del algoritmo, sí que es interesante conocer ciertos aspectos del juego para entender la complejidad del mismo y ciertos detalles del algoritmo. Por ello se resumen brevemente aquí sus reglas.

El go se juega sobre un tablero con 19x19 intersecciones, inicialmente vacío. En cada turno el jugador correspondiente coloca una ficha en una intersección, comenzando las negras, aunque también se puede *pasar* el turno (y no poner ficha). Un conjunto de fichas de un color es capturado cuando está completamente rodeado por fichas del otro color. Además, no se puede realizar un movimiento que provoque que el tablero vuelva a una situación anterior. Al terminar la partida (ambos jugadores pasan, o el tablero está completamente lleno) se contabilizan los puntos. En primer lugar, se identifican cuáles son las *piezas vivas* y las *piezas muertas*. Las piezas muertas son aquellas que, en caso de continuar la partida, serán inevitablemente capturadas, mientas que las piezas vivas son las restantes. En cuanto al sistema de puntuación, hay dos variantes:

- Reglas chinas: cada jugador recibe un punto por cada ficha suya en el tablero y por cada intersección dentro de su territorio.
- Reglas japonesas: cada jugador recibe un punto por cada ficha contraria capturada y por cada intersección dentro de su territorio.

Además, dado que han comenzado las negras, a las blancas se les suele dar algún tipo de compensación en forma de puntos extra. Mediante esta regla además se evita que haya empates, asignando una cantidad no entera de puntos.

3. AlphaZero

A lo largo del tiempo, el equipo de DeepMind ha propuesto distintas formas de crear una inteligencia artificial para jugar, primero al go, y después a cualquier juego de mesa para dos jugadores del que conozcamos las reglas, el estado pueda ser visto por ambos jugadores durante toda la partida y no haya eventos que dependan del azar[6]. Podríamos considerarlos una familia de algoritmos, la cual ha evolucionado mediante distintas mejoras que han servido no solo para aumentar el rendimiento del algoritmo sino también para hacerlo más general y robusto. En la figura 6 se pueden ver las diferentes versiones.

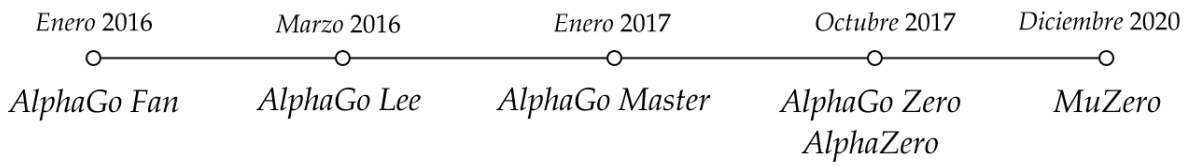


Figura 6: Cronología de las diferentes versiones del algoritmo, indicando su fecha de publicación[7].

El algoritmo en el que se centra este trabajo es AlphaZero. Sin embargo, la descripción principal de este algoritmo aparece en el artículo donde se presenta AlphaGo Zero: *Mastering the game of Go without human knowledge* [22]. AlphaGo Zero estaba centrado en jugar al go, pero sus ideas eran fácilmente extrapolables a otros juegos. Por ello los autores publicaron un nuevo artículo: *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play* [20], donde se indican las ligeras modificaciones que hay que hacer a AlphaGo Zero para que pueda enfrentarse a otros juegos. Es entonces cuando el algoritmo pasa a llamarse AlphaZero.

En esta sección se explica primero qué elementos componen AlphaZero y cuál es su función. Tras ello, se pasa a explicar en detalle cada uno de ellos: la red neuronal, el árbol de búsqueda de Monte Carlo y el aprendizaje por refuerzo, haciendo uso principalmente de los artículos anteriormente citados. Por último, se indican las modificaciones fundamentales entre las diferentes versiones del algoritmo expuestas en la figura 6, donde además de los artículos anteriormente citados se utilizan de forma puntual los de AlphaGo: *Mastering the game of Go with deep neural networks and tree search* [19] y MuZero: *Mastering Atari, Go, chess and shogi by planning with a learned model* [16].

3.1. Algoritmo

El algoritmo consta de tres elementos principales: aprendizaje por refuerzo, una red neuronal y árboles de búsqueda de Monte Carlo (de ahora en adelante MCTS por sus siglas en inglés). En esta sección vamos a introducir el papel general de estos tres componentes en el algoritmo y en

[6]En particular, los autores prueban el algoritmo en [20] sobre el ajedrez, shogi y go, juegos que claramente verifican estas premisas. Otros ejemplos serían el tres en raya y el Conecta 4, los cuales vamos a estudiar en este trabajo.

[7]Para aquellos algoritmos que han sido publicados, se utiliza como fecha la fecha de publicación del artículo: AlphaGo Fan [19], AlphaGo Zero [22] (AlphaZero fue publicado 2 meses después [20]) y MuZero [16]. Para los restantes (AlphaGo Lee y AlphaGo Master) se utiliza la fecha en la que se jugaron sus respectivos encuentros.

las secciones siguientes vamos a precisar el funcionamiento exacto de cada uno de ellos. La clave del algoritmo está en combinar de forma inteligente estos tres elementos para estimar de forma precisa el valor de una posición y la política a seguir para cada estado. De esta forma, se puede obtener para cada estado una jugada muy buena manteniendo el coste computacional dentro de unos límites razonables.

El primer elemento es la red neuronal, que va a recibir como entrada el estado actual s y nos va a devolver el valor actual v del estado, indicando la probabilidad de ganar, y un vector $\mathbf{p}$ indicando la probabilidad de seleccionar cada movimiento. Estas salidas vendrán determinadas por los pesos de la red, que denotaremos como θ. Por lo tanto, la red neuronal computará una función f_θ tal que:

$$f_\theta(s) = (\mathbf{p}, v)$$

Nuestro objetivo al entrenar esta red neuronal es encontrar unos parámetros θ de forma que f_θ nos proporcione una estimación acertada de cómo de buena es la posición actual y al mismo tiempo una buena política a seguir, dando una probabilidad mayor a aquellas acciones que conducen a mejores posiciones.

Aquí es donde entra en juego el segundo componente, el árbol de búsqueda de Monte Carlo. Desde el estado s actual vamos a simular varias partidas. En las simulaciones, aquellas líneas de juego más prometedoras serán exploradas más veces. Esto lo que nos permite es reducir el coste computacional, centrando nuestros recursos en aquellas variantes de la partida en las que tenemos mayores probabilidades de ganar. Tras realizar las simulaciones, ejecutaremos la acción a que nos reporte un mejor resultado. Para determinar cuáles son las mejores variantes utilizamos la red neuronal. Por un lado, las estimaciones del valor de cada estado nos permiten saber cuán bueno es el estado al que llegamos. Por otro, la distribución de probabilidades nos indica qué acciones deberíamos probar más y cuáles menos.

Por último, tenemos el aprendizaje por refuerzo. Tras jugar una partida, para cada acción ejecutada tendremos información sobre:

1. El estado en el que hemos realizado dicha acción.
2. La cantidad de veces que hemos explorado cada acción.
3. El resultado de la partida (si hemos ganado, empatado o perdido).

Esta información nos sirve para entrenar la red neuronal, de forma que el valor asignado a la posición refleje la probabilidad de que el resultado de la partida sea favorable o no y que las probabilidades de las acciones sean lo más cercanas posibles a las que el MCTS ha considerado como buenas. El tener una mejor red neuronal hace que el MCTS se centre en zonas más prometedoras, y esto hace a su vez que la red neuronal proponga mejores políticas. Esta es la idea innovadora que ha permitido al algoritmo obtener unos resultados mucho mejores que otros programas anteriores. Además, en ningún momento se hace uso de características específicas del go o del ajedrez, como sí hacían los mejores programas existentes cuando se publicó AlphaZero.

3.2. Red neuronal

Si nosotros pudiésemos explorar todas las líneas de juego seríamos capaces de, para cada posición, calcular si son ganadoras o perdedoras (o conducen a un empate) suponiendo que jugamos de forma óptima. Habrá posiciones en las que hagamos lo que hagamos terminaremos perdiendo: estas son las posiciones perdedoras. Habrá otras posiciones desde las cuales si nosotros efectuamos una cierta acción, dejemos a nuestro rival en una posición perdedora: estas son las ganadoras. Por último, habrá posiciones en las que nuestro rival tiene una serie de movimientos que fuerzan un empate.

Calcular de forma exacta si una posición es ganadora o perdedora es posible en juegos con un espacio de exploración pequeño. Sin embargo es totalmente impracticable para juegos como el go, en el que tanto el factor de ramificación como la profundidad del juego son inmensas. En el artículo de AlphaGo (la primera versión del algoritmo, anterior a AlphaZero) [19] los autores proponen reducir el factor de ramificación y la profundidad del juego para convertirlo en un problema asumible.

En primer lugar, muchas de las líneas de juego no son interesantes. Algunos movimientos son muy malos y conducirán a una derrota inevitable contra un jugador óptimo. Por lo tanto, si consiguiésemos restringirnos solo a aquellas líneas de juego interesantes reduciríamos el factor de ramificación. Esto se consigue mediante la introducción de una política $\mathbf{p}$. Esta política reflejará lo "buena" que es una acción a en un cierto estado s asignándole una mayor probabilidad $\mathbf{p}(s)(a)$ que a otras acciones. En consecuencia, cuando simulemos partidas partiendo del estado s no probaremos todas las acciones de forma equiprobable, sino que estudiaremos mucho más aquellas que la política nos indica que son prometedoras.

En segundo lugar, saber si una secuencia de movimientos conduce a una victoria o a una derrota requiere ejecutar todos estos movimientos hasta llegar al final de la partida y computar el resultado. Sin embargo, reduciríamos la profundidad de juego si fuésemos capaces de predecir para una posición no terminal qué es lo que va a ocurrir. Concretamente, la función que querríamos conseguir es una que otorgase a una posición valor 1 cuando bajo juego óptimo gana el jugador actual, -1 cuando pierde y 0 cuando empata. Evidentemente, si queremos el valor exacto volveríamos al problema inicial, pero si queremos un valor aproximado podemos simplificar el problema. Por lo tanto, nuestro objetivo en AlphaZero es calcular una función que nos devuelva valores cercanos a 1 cuando esa posición sea muy favorable al jugador actual, cercanos a -1 cuando sea muy desfavorable y 0 cuando esté equilibrada. Además, cuanto más favorable sea la posición actual más cercano será el valor a 1 (y cuanto más desfavorable sea más cercano será el valor a -1).

Para el cálculo tanto de la política como del valor los autores proponen utilizar una red neuronal. Si estamos en un estado s y los parámetros de la red neuronal son θ, entonces nuestra red neuronal nos va a devolver una tupla:

$$f_\theta(s) = (\mathbf{p}, v)$$

En primer lugar, $\mathbf{p}$ es un vector con la política proporcionada por la red neuronal. Es una distribución de probabilidad discreta sobre las posibles acciones, de forma que la red nos dice cuáles cree que son las acciones más prometedoras para dicho estado. Por otro lado, el valor v estima lo favorable que es la posición s para el jugador actual, suponiendo que este sigue la política $\mathbf{p}$. Esta función v no es más que una estimación de la función $V_{\mathbf{p}}$ que definimos en la introducción teórica

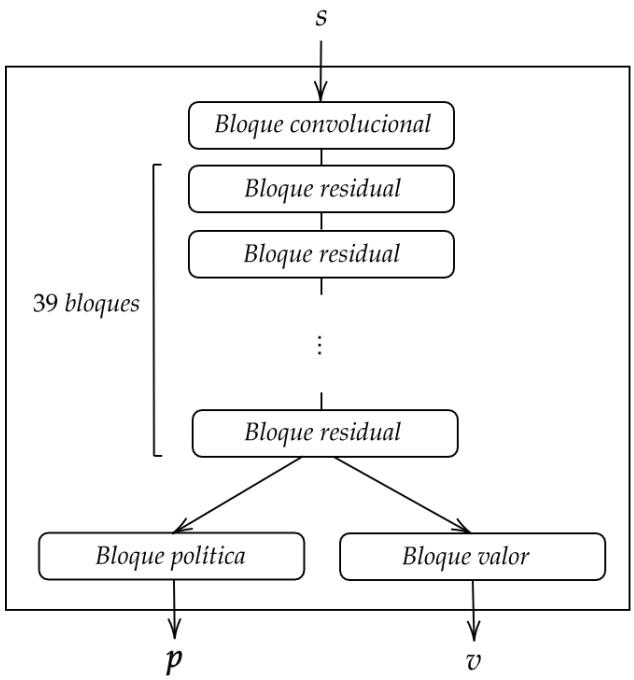


Figura 7: Estructura de la red neuronal utilizada en la versión final de AlphaGo Zero y AlphaZero. Durante el entrenamiento se utilizaban también instancias más pequeñas, con 19 bloques residuales.

al aprendizaje por refuerzo.

Mientras que en el algoritmo original AlphaGo había dos redes neuronales, encargándose una del cálculo del valor y la otra de la política, en AlphaGo Zero y AlphaZero se combinan ambas en una única red neuronal. Esto significa que las primeras capas son comunes y tras ellas tenemos unas pocas capas diferentes para generar cada salida.

La idea detrás de la red neuronal es que las capas vayan poco a poco capturando las características principales del estado. Tras aprender los parámetros, ciertos conjuntos de neuronas van a ser capaces de identificar ciertos patrones en el estado y en base a eso van a ser capaces de predecir si un estado es bueno o malo (valor) o qué acción sería más beneficiosa (política). Se supone además que estas características importantes de los estados van a ser fundamentales tanto en el cómputo del valor como de la política. Por ello se obliga a que los parámetros de varias capas sean iguales compartiendo dichas capas entre ambas redes. La arquitectura general de la red aparece descrita en la figura 7. La entrada de la red es un estado s. Este se pasa al primer bloque, un *bloque convolucional*. Su salida pasará por 19/39 *bloques residuales*[8]. La salida del último *bloque residual* está conectada por un lado al *bloque política*, encargado de calcular la política $\mathbf{p}$ para el estado actual, y por otro lado al *bloque valor*, encargado de calcular el valor v.

Un detalle importante es determinar qué es el estado s. Queremos que s contenga toda la

[8]Los autores entrenaron una pequeña red con 19 bloques durante 3 días para probar el funcionamiento del algoritmo y después la red definitiva con 39 bloques durante 40 días.

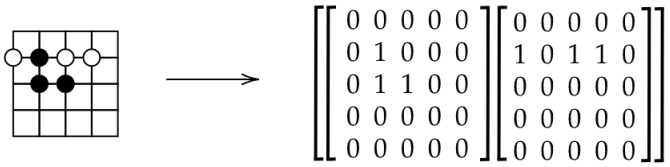


Figura 8: Representación correspondiente a un tablero 5x5. En la primera matriz tenemos un 1 en las posiciones en las que hay fichas negras (que en el go son las que juegan primero) y 0 en las que no. Ídem para las fichas blancas en la segunda matriz.

información relevante para que la red neuronal pueda calcular correctamente lo buena que es la posición y la acción a realizar. En el go no es suficiente con conocer el estado actual del tablero puesto que hay reglas que impiden la repetición de movimientos[9]. Por ello, en AlphaGo Zero no se pasa solo el tablero actual, sino los últimos 8 tableros de la partida. El propio equipo de DeepMind explicó además que no es simplemente un mecanismo para respetar las reglas del juego. Estos últimos movimientos sirven de "historia" y ayudan al algoritmo a saber qué ha jugado recientemente el oponente, indicándole dónde debería centrar su atención [21].

El tablero se proporciona indicando para cada uno de los colores las casillas que están ocupadas por fichas de ese color, por lo que cada tablero está representado por 2 matrices de tamaño 19x19, como se ejemplifica en la figura 8 para un tablero de tamaño 5x5. Además, se indica a qué jugador le toca mover mediante una última matriz de tamaño 19x19 rellena de unos si le toca jugar a las negras y ceros si es a las blancas. Esto se hace porque, al ser una red convolucional, es más sencillo pasarle otra capa entera que pasarle un único valor. Por ello, el estado s se corresponde con una matriz tridimensional de tamaño 19x19x17 (8 tableros $\times$ 2 matrices/tablero + 1 matriz indicando a quién le toca jugar).

En el apéndice A se explica en detalle cuál es la composición de cada bloque y la función que desempeña cada uno de sus elementos.

3.3. Árbol de búsqueda de Monte Carlo

La pregunta clave ahora es: ¿por qué se utiliza el MCTS si ya tenemos una red neuronal que nos proporciona una buena función de valor y una buena política? Podríamos simplemente, para cada estado, obtener el valor aproximado de los estados siguientes y movernos hacia el estado que la red considere que el oponente tiene menos probabilidades de ganar. O podríamos seleccionar la acción con probabilidad más alta. Como se resalta en el artículo de AlphaGo Zero, la combinación del MCTS con una red neuronal nos permite seleccionar movimientos notablemente mejores que la red neuronal por sí sola. La idea es que el MCTS se aprovechará de la política propuesta por la red neuronal para explorar las acciones más prometedoras y al mismo tiempo propondrá una política que en teoría es buena, para que la red neuronal pueda aprender de ella. De esta forma el MCTS y la red neuronal se ayudan para conseguir la mejor política posible.

Como ya explicamos en la sección anterior, la idea de la política y la función de valor era reducir

[9]Como se indica en las reglas del go en la introducción, bajo ciertas condiciones se pueden capturar las fichas del oponente. Esto podría llevar a ciclos infinitos si no hubiera reglas que evitasen las repeticiones.

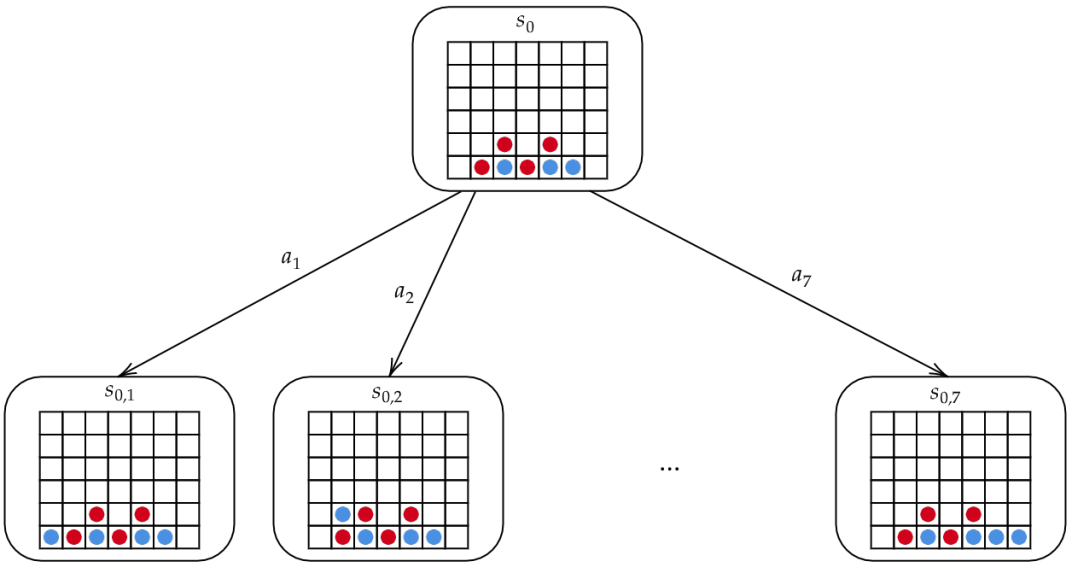


Figura 9: MCTS para el Conecta 4 que tiene como raíz el estado s_0. Desde ahí podemos realizar 7 acciones, correspondientes a colocar la siguiente ficha en cada una de las columnas con hueco. Esto nos llevará a los nodos $s_{0,i}$, dependiendo de la acción realizada. Nótese que en los nodos no necesitamos almacenar ninguna información adicional. Para un juego arbitrario, el número de aristas con origen en un estado dependerá del número de acciones válidas en ese estado.

el número de líneas de juego en las que centrarnos y utilizarlas para calcular de forma precisa la mejor acción en cada situación. Para ello, el MCTS va a realizar múltiples simulaciones guiado por la política y la función de valor, con la esperanza de estar centrándose en las mejores variantes, y va a escoger la acción que conduzca a un mejor resultado solo en estas variantes.

En esta subsección ilustraremos además el funcionamiento del MCTS sobre el Conecta 4, dado que es el juego para el que posteriormente se implementará el algoritmo. La estructura del MCTS se puede adaptar fácilmente a cualquier otro juego, teniendo en cuenta el número de acciones posibles en cada estado.

3.3.1. Estructura del árbol

El MCTS es una estructura arbórea en la que cada uno de los nodos del árbol va a representar un estado s del juego. La información importante la vamos a tener en las aristas, que van a representar las distintas acciones que se pueden realizar desde un estado. Dos nodos (representando estados s_0 y s_1) estarán conectados por una arista si existe una acción a tal que al realizar a en s_0 nos lleva a s_1. Cada arista presente en el árbol estará caracterizada por un par (s, a) y, para cada una de ellas, el MCTS va a guardar la información que necesita para decidir cuán prometedora es. En la figura 9 se muestra la estructura del MCTS y en la figura 10 una arista en detalle.

Los datos que se utilizan para calcular lo buena que es una acción son 4:

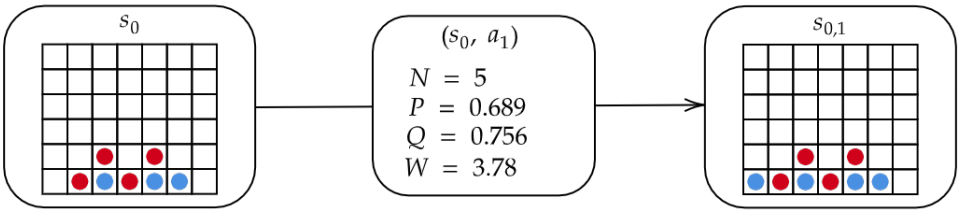


Figura 10: Detalle de una arista del MCTS, en este caso la correspondiente a aplicar a_1 desde s_0. Los números proporcionados son ilustrativos, no se corresponden con el modelo entrenado.

- $N(s, a)$: el número de veces que se ha ejecutado la acción a desde el estado s.
- $P(s, a)$: la probabilidad con la que deberíamos ejecutar la acción a desde el estado s. Esta probabilidad viene dada por la política proporcionada por la red neuronal y se emplea para dirigir el MCTS cuando aún no ha ejecutado suficientes simulaciones. Podríamos considerarla como una "política inicial", en la que tienen una mayor probabilidad aquellas acciones que la red neuronal considera más prometedoras.
- $Q(s, a)$: el valor medio para la acción a desde el estado s. Cada simulación ejecutada que pase por el estado s y realice la acción a va a terminar en una posición con un cierto valor v. Este campo contendrá la media de estos valores v, indicando cuán bueno ha sido en promedio dicho movimiento.
- $W(s, a)$: la suma de los valores de las posiciones a las que se ha llegado en las simulaciones en las que se ha ejecutado la acción a desde el estado s. Nos permitirá actualizar $Q(s, a)$ de forma más sencilla (puesto que $Q(s, a) = \frac{W(s,a)}{N(s,a)}$).

La implementación más sencilla de esta estructura es de forma recursiva. Para ello, incluimos en las aristas otro campo, *estado siguiente*, que no es más que otro MCTS que tiene como raíz el estado al que llegamos tras aplicar al estado inicial la acción indicada por la arista.

3.3.2. Ejecución

La idea es construir un MCTS al comienzo de una partida con un único nodo representando el estado inicial, la partida en la que no se ha efectuado ningún movimiento. A la hora de elegir cada movimiento, vamos primero a poblarlo mediante simulaciones. Utilizaremos la información recabada para decidir qué acción realizar y pasaremos al siguiente estado, donde repetiremos el proceso.

Para un cierto estado, el movimiento a realizar se va a escoger en dos fases. La primera fase es la de *simulación*, que es la encargada de poblar el MCTS. Se van a realizar múltiples simulaciones

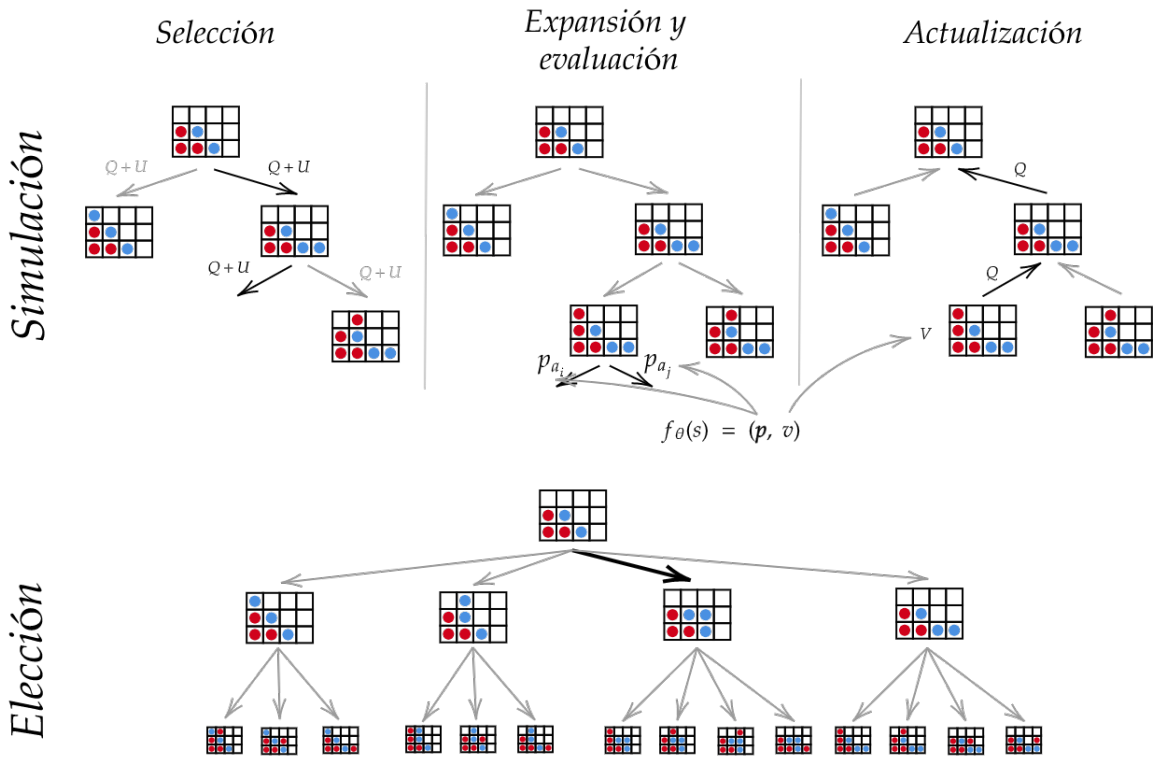


Figura 11: Esquema general de las etapas que realiza el MCTS para decidir la acción a realizar en un estado. Basada en la figura 2 de [22].

y en cada una de ellas se va a seguir la línea de juego considerada más prometedora, hasta que se llegue a un estado inexplorado. Cada simulación se divide a su vez en 3 etapas:

1. *Selección*: se escogen las acciones a realizar en base a la información que se tiene hasta llegar a un nodo sin explorar.

2. *Expansión y evaluación*: se expande y evalúa el nuevo estado.

3. *Actualización*: se actualiza el MCTS incorporando la información obtenida en esta ejecución.

Realizaremos simulaciones hasta haber hecho un número suficiente, o hasta que consumamos el tiempo disponible. Una vez poblado, en la segunda fase, de *elección*, se escoge el movimiento más prometedor en base a esta información. Se resume el proceso en la figura 11. Procedemos ahora a detallar cómo se ejecuta exactamente cada fase.

Selección

Inicialmente el MCTS solo cuenta con el nodo raíz y a medida que realizamos simulaciones iremos poblando el MCTS con nuevos estados. Denotaremos por nodo hoja aquellos nodos que no están aún en el árbol, que se corresponden con estados que aún no han sido explorados.

Desde el nodo raíz del MCTS, correspondiente al estado actual del tablero, vamos a comenzar una simulación. Seleccionamos una de las acciones posibles y la realizamos sobre el tablero actual. Esto nos lleva a un nuevo estado, que se corresponde con el nodo del MCTS que pende de la raíz actual y que está conectado con ella a través de la arista que representa la acción elegida. Esto lo podremos realizar siempre y cuando no hayamos terminado la partida o en alguna simulación anterior hayamos llegado hasta el estado actual. En esta fase vamos a descender desde la raíz hasta un nodo hoja, y por lo tanto tendremos que escoger tantas acciones como sean necesarias hasta que lleguemos a un estado que no está en el MCTS[10]. La acción seleccionada en cada paso será:

$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$

siendo:

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

y siendo c_{puct} una constante que controla cuánto queremos explorar. En la figura 12 vemos el proceso.

El origen de esta idea está en el problema conocido como bandido de varios brazos. En este problema se tienen varias máquinas tragaperras, cada una de las cuales proporciona diferentes recompensas atendiendo a una cierta distribución desconocida, y lo que queremos es obtener la mayor recompensa posible. Para ello, en cada tirada debemos escoger en qué máquina queremos realizarla y lo único que conocemos son los resultados de tiradas anteriores, pero no sabemos lo que vamos a obtener en esta. Este problema pone de manifiesto el conflicto de explotación contra exploración que introdujimos en la sección 2. Podemos o bien gastar nuestras tiradas en la máquina que hasta el momento nos ha dado mejores resultados, o podemos probar en otras máquinas para mejorar la estimación de la recompensa que obtendremos en cada máquina y así poder decidir una estrategia con más información.

Una técnica para resolverlo es la cota superior de confianza (*Upper Confidence Bound* o UCB por sus siglas en inglés)[11]. Este algoritmo se basa en cambiar el balance entre exploración y explotación a lo largo de la ejecución, de forma que exploramos cuando tenemos menos información y explotamos cuando tenemos más. En la práctica se traduce en escoger en cada momento la acción

$$a_t = \text{argmax}_a \left(Q(a) + c \sqrt{\frac{\sum_b N(b)}{N(a)}} \right)$$

[10]Podría ocurrir que en una simulación lleguemos hasta un estado terminal (en el que uno de los jugadores ha ganado) y lo incluyamos en el MCTS. Si volvemos a llegar a este estado en otra simulación, como habríamos incluido el estado terminal en el MCTS ya no tendríamos que descender hasta un estado hoja, puesto que la partida se terminaría antes. Para garantizar que siempre que descendemos va a haber un estado hoja válido al final, no incluiremos en el MCTS los estados terminales. De esta forma, aunque lleguemos varias veces a ellos en diferentes simulaciones siempre serán nodos hoja.

[11]La explicación del UCB se basa en [15].

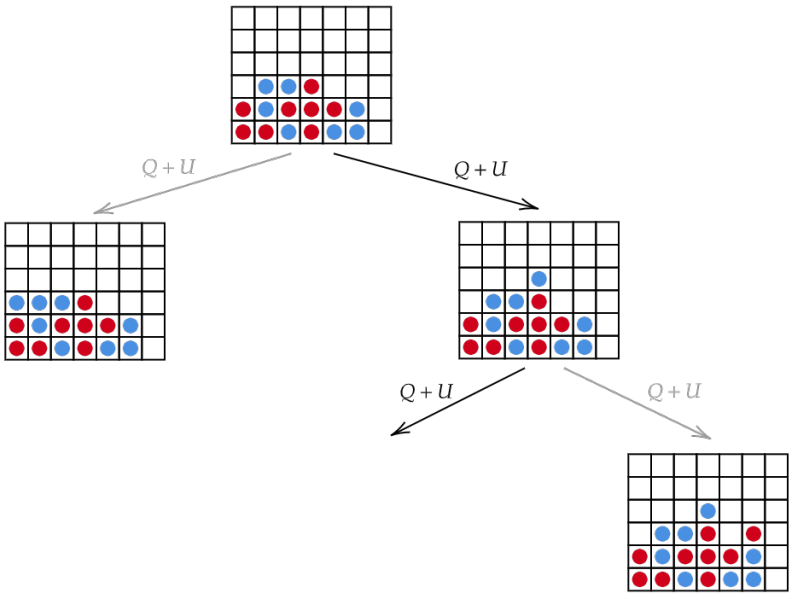


Figura 12: Fase de selección en AlphaZero. Desde la raíz, vamos seleccionando aquellas acciones que maximizan $Q + U$ hasta llegar a un estado no explorado. En negro tenemos las decisiones tomadas en cada paso. Basada en la figura 2(a) de [22].

siendo $Q(a)$ el valor estimado de la acción a en el momento actual, $N(a)$ el número de veces que hemos probado la acción a hasta el momento y c una constante que controla la exploración. Vemos que esta fórmula es prácticamente idéntica a la utilizada en AlphaZero. Esto es razonable, porque el problema es también idéntico: tenemos varios posibles lugares donde colocar nuestra pieza y queremos encontrar cuál es el que nos conducirá a una mejor posición, para lo cual tenemos que distribuir las simulaciones realizadas de la mejor forma posible.

A cada acción se le asignan dos valores que indican lo buena que es la acción y cuánto la hemos explorado, y la decisión de qué acción realizamos se basa en la suma de ambos. El primer término (Q) representa los valores observados, ya que contiene la media de los valores obtenidos en las diferentes simulaciones realizadas para esa acción. El segundo término (U) representa la exploración realizada: cúantas veces se ha simulado esa acción frente al resto. La idea es dar más importancia a uno u otro término dependiendo de la fase de la simulación en la que nos encontremos:

- Al principio de la simulación no tenemos o tenemos muy pocas observaciones, por lo que queremos probar distintas acciones.
- Al final de la simulación tenemos muchas observaciones, que medirán de forma precisa el valor de cada jugada.

Esto es lo que conseguimos con estas definiciones. Cuando hemos probado pocas veces una acción, $N(s, a)$ tiene un valor pequeño y el término $U(s, a)$ tiene mucha relevancia. A medida que probamos más veces dicha acción, la importancia del valor estimado, indicado por $U(s, a)$, decrece

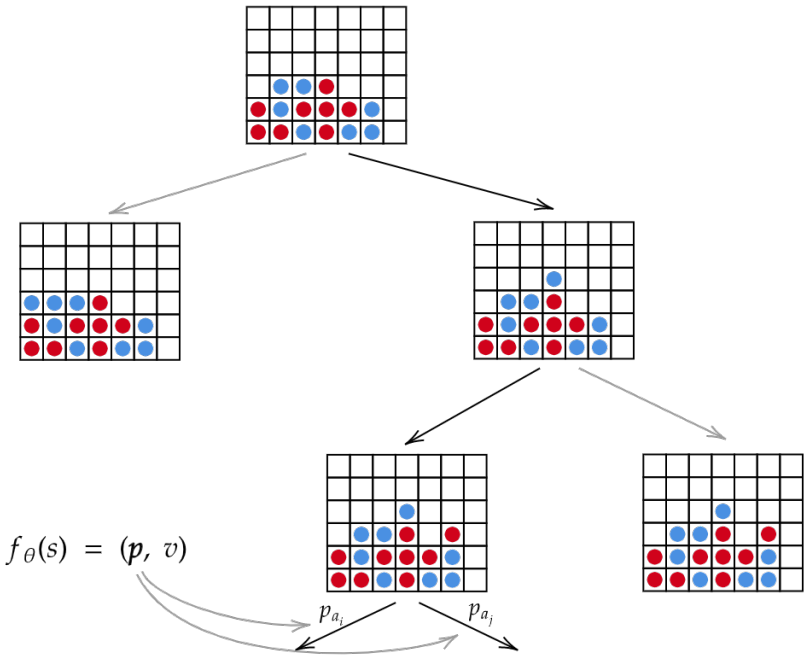


Figura 13: Fase de expansión en AlphaZero. Creamos un nuevo nodo con el estado actual s y para cada posible acción las probabilidades asignadas por la red neuronal, a la que pasamos como entrada el nuevo estado. Basada en la figura 2(b) de [22].

en pos del valor observado, indicado por $Q(s, a)$. c y c_{puct}[12] son la misma constante, que se encarga de controlar la importancia que damos a la exploración. En AlphaZero se incluye el término $P(s, a)$ generado por la red neuronal para predecir si realmente vale la pena explorarla o es mejor no perder el tiempo.

Expansión y evaluación

Una vez llegamos a un nodo hoja, representando el estado s_L, lo expandimos. Para ello tenemos que crear un nuevo MCTS con el estado s_L como raíz y del que salen aristas representando las acciones válidas en ese estado, con la probabilidad indicada por la política proporcionada por la red neuronal. Para ello, introduciremos en la red neuronal el estado s_L y obtendremos como resultado un número v con el valor de la posición y $\mathbf{p}$, un vector con la probabilidad de cada una de las acciones desde ese estado. El valor v lo reservamos para la siguiente fase.

Creamos un nuevo nodo en el árbol correspondiente a este estado hoja, como se indica en la figura 13. Para cada una de las acciones válidas a se crea una arista en nuestro árbol y se inicializa su información a:

[12]El subíndice PUCT hace referencia al algoritmo PUCT, que significa (Predictor + UCB aplicado a árboles) y los autores de AlphaZero especifican que para la simulación se está empleando una variante del mismo.

- $N(s,a) = 0$: la acción no ha sido explorada ninguna vez.
- $W(s,a) = Q(s,a) = 0$: no hemos hecho ninguna ejecución por lo que no hemos alcanzado ningún valor.
- $P(s,a) = \mathbf{p}_a$: la probabilidad de realizar la acción a asignada por la red neuronal.

Por último, para incrementar la exploración se añade ruido a las probabilidades proporcionadas por la política mediante una distribución de Dirichlet. Si podemos realizar n acciones distintas, vamos a extraer un vector $\{x_1, \ldots, x_n\}$ de tal forma que $x_i > 0$ y $\sum_{i=0}^{n} x_i = 1$, y vamos a utilizar como probabilidad de seleccionar cada acción:

$$\epsilon\, p_a + (1-\epsilon)x_a$$

siendo ϵ un valor que controla la importancia que le damos a la exploración. En el artículo original se utiliza $\epsilon = 0{,}25$ y como parámetro que controla la distribución de Dirichlet $\alpha = 0{,}03$. A pesar de que estamos introduciendo ruido en la política, el MCTS debería ser capaz de terminar decantándose por los movimientos buenos independientemente del efecto del ruido. Este ruido es en la práctica una parte bastante importante del algoritmo y volveremos sobre él más adelante.

Actualización

Una vez que hemos llegado al nodo hoja s_L y lo hemos expandido, actualizamos las estadísticas de la trayectoria que hemos seguido desde el nodo raíz actual a s_L utilizando el valor v, como se puede ver en la figura 14. Concretamente, para los 4 valores en cada arista tenemos:

- $N(s,a) = N(s,a) + 1$, puesto que hemos realizado la acción a desde s una vez más.
- $W(s,a) = W(s,a) + \tilde{v}$, añadiendo al total de valores el valor $\tilde{v}$ de la trayectoria explorada, que depende de a quién le toque jugar.
- $Q(s,a) = \frac{W(s,a)}{N(s,a)}$, incorporando a la media el valor de la última trayectoria.
- $P(s,a)$ se mantiene igual, puesto que cuando se expandió el estado se evaluó la posición y se obtuvo la política, y esta no se modifica durante la ejecución.

Nótese que al actualizar el valor de la arista no usamos v, sino $\tilde{v} = \pm v$, dependiendo del jugador al que le toca jugar en cada posición de la trayectoria y el jugador al que le toca jugar en el nodo hoja. Esto es algo razonable: si la simulación termina en un estado bueno para uno de los jugadores, la acción será mala para el otro jugador, puesto que el rival ha quedado en una posición ventajosa. En la figura 15 se explica en detalle cómo sucede la actualización.

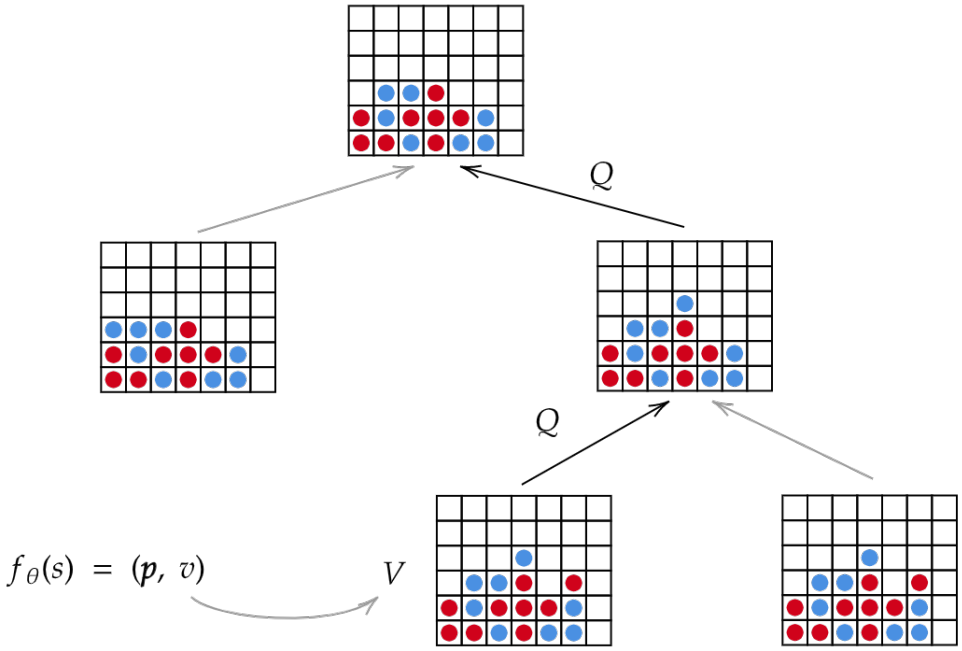


Figura 14: Fase de actualización en AlphaZero. El valor proporcionado por la red neuronal se propaga hasta la raíz, actualizando solamente las aristas recorridas en esta simulación. Basada en la figura 2(c) de [22].

Elección

Una vez hemos terminado la fase de simulacion, se elige el movimiento a realizar en base a dichas simulaciones. El algoritmo selecciona cada acción a con probabilidad:

$$\pi(a|s_0) = \frac{N(s_0, a)^{1/\tau}}{\sum_b N(s_0, b)^{1/\tau}}$$

siendo b todas las acciones posibles desde el estado actual.

El parámetro τ es un parámetro que indica la *temperatura* del sistema y nos sirve para controlar la exploración. Cuanto menor sea τ, mayor es la probabilidad de jugar la acción que ha sido visitada más veces, y que teóricamente es la jugada más fuerte. Por el contrario, valores de τ mayores favorecen la exploración permitiendo seguir líneas de juego a priori menos prometedoras, pero garantizando así que se realiza suficiente exploración.

En el artículo de AlphaGo Zero se establece $\tau = 1$ para los primeros 30 movimientos, para garantizar variedad en las aperturas. Para el resto de la partida se utiliza $\tau \approx 0$, para seleccionar la acción más visitada.

A pesar de que en un principio pudiese resultar más razonable seleccionar las acciones según

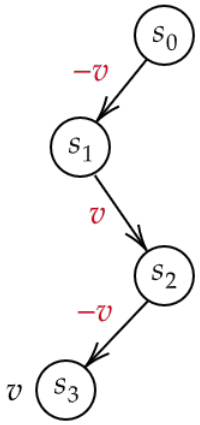


Figura 15: Funcionamiento del proceso de actualización del valor Q de cada arista mediante el valor v obtenido en una simulación. En primer lugar partimos desde el nodo s_0 y el nodo hoja será el s_3. Tras expandir el nodo s_3, la red neuronal evaluará el estado s_3 y nos devolverá un valor v que indica lo buena que es la situación **para el jugador al que le tocaría jugar en** s_3. A la hora de actualizar las aristas, tenemos que tener en cuenta a qué jugador le tocaría realizar la acción para saber si el valor con el que hay que actualizar es v o $-v$. Por ejemplo, en s_2, el jugador que realiza la acción deja a su rival en una posición con valor v, por lo que esa acción tiene para él mismo un valor $-v$. En definitiva, sumar a una arista v o $-v$ dependerá de la paridad de la longitud de la simulación y la paridad de la arista.

el valor medio de cada acción proporcionado por las simulaciones, este podría estar sesgado por la diferencia en el número de veces que simulamos cada acción. Además, como en cada paso simulamos la acción más prometedora, que hayamos simulado una acción muchas más veces que cualquier otra significará que el sistema la ha considerado prometedora en numerosas ocasiones.

Una vez elegimos la acción a realizar descendemos al siguiente nivel del MCTS. El resto de subárboles que partían del nodo anterior, correspondientes a las acciones que no hemos tomado, se descartan, puesto que corresponden a hipotéticas líneas de juego que no se han seguido. El subárbol que sí hemos seguido lo mantentemos, puesto que contiene líneas de juego que aún pueden ocurrir y de las que ya hemos obtenido información que podemos aprovechar.

3.4. Aprendizaje por refuerzo

El último componente de AlphaZero es el aprendizaje por refuerzo. El sistema va a "aprender" a jugar jugando contra sí mismo. Más concretamente, lo que queremos es entrenar la red neuronal de forma que para cada estado s aprenda la distribución de probabilidad $\mathbf{p}$ de los movimientos del MCTS y el valor v de la posición indicando para qué jugador es más favorecedora la posición.

La red neuronal se inicializa con pesos aleatorios. En cada instante t que realizamos un movimiento (fase de elección), vamos a recopilar dos datos:

1. El estado actual s_t.
2. La distribución de movimientos π_t dada por el MCTS, que como hemos visto en la sección

anterior es proporcional al número de veces que hemos visitado cada acción exponenciada al parámetro τ.

Durante la partida almacenamos los valores de estas tuplas y al terminarla se conoce el ganador. A estas tuplas añadimos entonces un tercer elemento z_t indicando la recompensa. Será $+1$ si el jugador que ha realizado la acción ha terminado ganando, -1 si ha perdido o 0 si han empatado. Por lo tanto, en cada partida obtenemos para cada acción $i \in \{0, \ldots, T\}$ la tupla (s_i, π_i, z_i) .

En el artículo original una partida de go termina por dos motivos:

1. No se pueden realizar más movimientos, los jugadores llegan a un acuerdo o la partida excede una longitud máxima[13], lo que evidentemente supone el final de la partida.

2. Cuando la probabilidad de ganar calculada por el algoritmo cae por debajo de un límite. Esto sirve para evitar realizar cómputos innecesarios y poder así realizar más simulaciones. Para rendirse, se fija un valor v_{resign} de forma que si se llega a un estado en el que su valor v es inferior a v_{resign} entonces AlphaZero se rinde. Se especifica además que este valor v_{resign} se escoge de forma automática para que el número de partidas que AlphaZero hubiera conseguido ganar y que no lo ha hecho por haberse rendido sea inferior al 5 %. Para medir esto, en un 10 % de las partidas se impide que AlphaZero se rinda y se juegan hasta el final, actualizándose v_{resign} en consecuencia.

Nuestro objetivo es minimizar el error entre el valor predicho para cada estado por la red neuronal y el ganador final de la partida, al mismo tiempo que maximizamos la similitud entre la política propuesta por la red neuronal y la política π_t proporcionada por el MCTS. Para lograr esto, vamos a utilizar como función de pérdida para el valor el error medio cuadrático y para la política la entropía cruzada. Sobre estas dos funciones se habla en detalle en el apéndice A. Además, vamos a añadir un término de regularización (regularización L2) para evitar que los pesos de la red se hagan muy grandes y se produzca sobreaprendizaje. La función de pérdida total nos queda:

$$l = (z - v)^2 - \pi^t \log \mathbf{p} + c\|\theta\|^2$$

donde c es un parámetro que va a controlar la regularización, y en el artículo se escoge $c = 10^{-4}$. Esta función l la aplicaremos sobre cada tupla (s_t, π_t, z_t), por lo que al entrenar la red neuronal con varias de estas tuplas al mismo tiempo la función de pérdida total será la suma de la función l para cada una de ellas.

Un detalle bastante importante del algoritmo es si realmente queremos entrenar nuestra red neuronal con esta función de pérdida. Por un lado, los valores z_t calculados reflejarán si una posición es buena o mala tanto para el jugador que ha ganado como para el jugador que ha perdido. Por otro lado, la política utilizada por el jugador que ha ganado también es interesante, porque esta política ha conducido al jugador a la victoria. Pero no parece muy inteligente enseñarle al jugador que ha perdido una política con la que ha perdido, pese a que esto sea lo que se consiga con esta función de pérdida. En la sección 5.2 volveremos sobre esto y lo comentaremos en detalle.

[13] En el go en cada turno se pone una ficha y cada jugador tiene un cierto número de fichas, pero el tablero puede no llenarse porque se pueden capturar piezas del oponente. Por ello tras un cierto numero de jugadas la partida finaliza obligatoriamente.

En todo momento se va a mantener una única red neuronal, que se va a ir entrenando constantemente con las partidas generadas. De esta red neuronal vamos a obtener tanto la política como el valor para la partida actual. Esto es diferente en AlphaGo Zero, donde se mantiene un *mejor jugador*, que es la red neuronal que mejores resultados nos ha dado hasta el momento, y esta es la encargada de generar las estimaciones. Tras cada iteración de entrenamiento se comparaba el mejor jugador con la red actual, y si la red actual ganaba al menos en un 55 % de las veces al mejor jugador, pasaba a ser el mejor jugador. La idea era tener en todo momento dos redes, una que sabemos que funciona bien (el mejor jugador) y otra que está aprendiendo; pero solo se utiliza la red que está aprendiendo para generar las simulaciones cuando nos hayamos asegurado de que realmente es mejor que la anterior, al menos contra ella, para no dar un paso atrás.

Ambas aproximaciones son bastante diferentes a la que se utiliza en AlphaGo, donde los ejemplos de entrenamiento se obtienen enfrentando la red neuronal actual contra una red neuronal en una iteración de entrenamiento anterior escogida de forma aleatoria de entre todas las iteraciones de entrenamiento realizadas hasta el momento. Esto es un aspecto muy delicado. El equipo de DeepMind hizo un AMA (*Ask Me Anything*) en Reddit [21], donde invitaban a los entusiastas del aprendizaje por refuerzo a preguntar lo que quisieran sobre los algoritmos de AlphaGo y AlphaGo Zero. Una de las preguntas más repetidas fue que cómo lograba AlphaGo Zero aprender de forma tan estable entrenando contra tan solo las últimas iteraciones[14], cuando en los algoritmos de aprendizaje por refuerzo es común que se produzcan *olvidos catastróficos* (olvidar lo que había aprendido el sistema en iteraciones precedentes). Para evitar esto, se suelen utilizar numerosos "puntos de guardado" (como se hacía en AlphaGo al enfrentarse a cualquier iteración anterior). La respuesta del equipo de DeepMind es que la utilización del MCTS estabiliza enormemente el entrenamiento y evita esos problemas.

Sin embargo, más interesante aún es la respuesta de Thomas Anthony, autor de *Thinking Fast and Slow with Deep Learning and Tree Search* [4], artículo en el que se discuten ideas muy similares a las de AlphaZero. En primer lugar compara su trabajo con el de DeepMind y resalta varios puntos muy interesantes:

- AlphaGo Zero entrena con movimientos de las últimas 500000 partidas. En cada iteración de entrenamiento, se retiran las 25000 partidas más antiguas y se incluyen 25000 nuevas. Esto provoca que la política vaya cambiando lentamente, puesto que la mayor parte de partidas con las que entrenamos dos iteraciones consecutivas son idénticas. Por ello, aunque las partidas de entrenamiento se generen entre dos redes neuronales similares, se siguen manteniendo ejemplos antiguos que actúan como esos puntos de guardado. Sin embargo, dice que eliminando esta idea y entrenando tan solo con las partidas de esta iteración el entrenamiento también es bastante estable.
- En su trabajo, sin utilizar el ruido de Dirichlet, el aprendizaje sigue siendo estable. Esto es importante, porque la función del ruido será ayudar a escapar de políticas malas.

Y esto le lleva a considerar también que la utilización del MCTS es la causante de esta estabilidad. Su suposición es que hay dos motivos que pueden introducir inestabilidad:

[14]Nótese la diferencia entre AlphaGo y AlphaGo Zero/AlphaZero. En AlphaZero se entrena contra la última iteración de entrenamiento, mientras que en AlphaGo Zero se entrena con el mejor jugador, que es prácticamente la misma red neuronal que la actual, tan solo hay unas pocas iteraciones de entrenamiento de diferencia. Sin embargo, en AlphaGo se puede entrenar contra **cualquiera** de las iteraciones anteriores, habiendo una diferencia mucho mayor.

1. Que se olvide de información importante de posiciones que ya no visita (porque la política otorga a las acciones que llevan a esas posiciones probabilidades bajas). Sin embargo, AlphaZero explora los movimientos iniciales que no son excesivamente malos gracias a que el parámetro τ vale 1 durante los primeros 30 movimientos. Si el oponente juega algo que AlphaZero ha probado y ha terminado considerando malo, es porque el movimiento tiene una réplica que AlphaZero encontró en el pasado. Como no hemos visitado ese estado de nuevo, no se habrá modificado la política que AlphaZero calculó en el pasado, y por lo tanto será capaz de ganar.

   Desde mi punto de vista esto no es del todo correcto. Aunque no se vuelva a visitar un estado, la política calculada para dicho estado sí se modifica porque se están modificando los pesos de la red, lo que influirá también en la política calculada para dicho estado.

2. Que AlphaZero se dedique a explotar una vulnerabilidad que encuentre en el otro jugador en un estado s en lugar de aprender a jugar contra un jugador general. Sin embargo, para ello tiene que aprender a llegar al estado s, lo que implica realizar un gran número de simulaciones en ese estado y durante el proceso encontrará mejores alternativas antes de haber aprendido a explotar la vulnerabilidad, evitando que esto suceda. La clave según el autor es que en AlphaZero se mira más allá gracias a las simulaciones, mientras que en los métodos en los que no se mira más allá sí que se aprende instantáneamente a explotar la vulnerabilidad.

El MCTS, gracias a las múltiples simulaciones que realiza amortigua (al menos teóricamente) ambos problemas. Sin embargo, en AlphaGo ya se utilizaba un MCTS y también se emplean puntos de guardado (aunque el funcionamiento del MCTS es ligeramente distinto, en la sección 3.6 se explica más en detalle el funcionamiento de AlphaGo). En la sección 5 implementaremos el algoritmo para el tres en raya y el Conecta 4 y veremos si estas teorías realmente se reflejan en nuestra implementación.

3.5. Jugando contra otros jugadores

Una vez que hayamos entrenado el algoritmo, para jugar contra jugadores arbitrarios se utiliza el mismo esquema eliminando la fase de aprendizaje. La red neuronal con los pesos aprendidos guía las simulaciones realizadas por el MCTS y una vez realizado el número de simulaciones deseado se ejecuta el movimiento utilizando exactamente el mismo procedimiento.

Una situación frecuente es enfrentarse a otras versiones de AlphaZero, en las que varían los pesos de la red[15]. Para ello, hay que mantener dos MCTS distintos, uno por cada jugador. La razón de esto es que los valores y políticas estimadas para cada estado varían entre ambas redes neuronales. Por lo tanto, si se utiliza un único MCTS como hacíamos en el entrenamiento, un jugador heredaría las simulaciones calculadas por el otro. Al no ser consistentes con las suyas propias, esto provocaría que la acción ejecutada por un jugador no solo dependa de ese jugador, sino de los cálculos del otro, puesto que al final la acción escogida depende del número de veces que se ha simulado.

Este problema no aparece cuando estamos jugando contra la misma versión de AlphaZero, como ocurre a lo largo del entrenamiento. Como la red neuronal es la misma, el valor y la política

[15]En realidad, este razonamiento es válido para cualesquiera dos AlphaZero distintos enfrentándose, ya sea con pesos en la red distintos, con arquitecturas para la red neuronal distintas, etc.

estimados para cada estado coincidirán para ambos jugadores. Por ello, en un cierto momento la secuencia de estados que se simula será exactamente la misma independientemente del jugador que la simule. De ahí que podamos utilizar un único MCTS.

Esto tendrá dos ventajas. La primera es que, pese a que realizamos las mismas simulaciones por jugada, el número total de simulaciones en el MCTS será mayor, porque tendremos las simulaciones de los dos jugadores juntas. Si tuviésemos dos MCTS por separado, muchos nodos hoja serían expandidos dos veces (una por jugador), cosa que no sucede al juntarlos. Esto permitirá llegar en las simulaciones hasta nodos un poco más lejanos, y por ende mejorar el cálculo de $Q(s, a)$. Podría ocurrir que la acción escogida en una misma situación sea diferente. Pero si esto ocurre es porque gracias a las simulaciones de ambos jugadores tenemos más información y podemos elegir la acción en base a la información extra. Por otro lado, los estados terminales (en los que la partida se acaba) no se expanden, por lo que en este MCTS combinado el consumo de memoria será ligeramente inferior que el de dos MCTS por separado. Por ello, en este caso es una ventaja utilizar un único MCTS, a diferencia de lo que ocurría en el caso anterior. Aquí estamos suponiendo además que ambos jugadores están bajo nuestro control, como es el caso del entrenamiento. En caso contrario, cada jugador necesariamente deberá contar con un MCTS distinto.

3.6. Versiones

Como ya anticipamos al principio del capítulo, DeepMind ha presentado múltiples versiones anteriores y posteriores a AlphaZero. En esta sección se entra más en detalle en las diferencias entre versiones. Es muy interesante observar estas modificaciones, puesto que permiten paliar problemas de versiones anteriores y también ver qué elementos se han mantenido constantes. Observar la familia de algoritmos en su conjunto permite entender cuáles son las ideas que hay por debajo de estos y entender realmente cómo funcionan.

3.6.1. AlphaGo Fan

Es la versión que se enfrentó a Fan Hui en octubre de 2015, al que venció 5 a 0. Es una versión con un proceso de entrenamiento bastante más complejo e inicialmente se apoya en bases de datos de partidas profesionales de go.

En vez de una única red combinada para la política y el valor se tienen dos redes neuronales separadas. El proceso de entrenamiento comienza con aprendizaje supervisado sobre una base de datos en la que para cada estado unos expertos predicen el movimiento a realizar. Comenzamos entrenando dos redes neuronales para la política: p_σ y p_π. La red p_σ tiene como objetivo proporcionar para cada estado la mejor política posible. Sin embargo, también queremos tener una red que nos permita simular partidas muy rápidamente. Por ello se entrena una red p_π mucho más rápida pero menos precisa. Concretamente, en la base de datos que utilizan los autores, p_σ acierta la jugada del experto un 55,7 % de las veces, frente a un 24,4 % para p_π. A cambio, p_π necesita tan solo 2µs para evaluar la posición, frente los 3ms de p_σ. Otro aspecto importante es que la red p_π no utiliza como entrada el estado del tablero, sino una serie de datos extraídos de la posición actual escogidos por los diseñadores del algoritmo.

Una vez tenemos estas redes entrenadas se pasa al aprendizaje por refuerzo. La red p_σ va a

adoptar el papel de predecir la política, como hacemos en AlphaZero, mientras que una red v_θ se va a encargar del valor. Para entrenar la red p_σ mediante aprendizaje por refuerzo esta juega partidas contra ella misma pero en una iteración de entrenamiento anterior. La iteración anterior se escoge de forma aleatoria para evitar el sobreaprendizaje[16]. Para cada estado se almacena la política propuesta por la red, la acción escogida y el resultado de la partida. Entonces, se entrena la red de la política para en cada estado hacer más probable la acción realizada si se ha ganado la partida o menos probable si se ha perdido.

Por otro lado, para el valor se entrena la red v_θ minimizando la diferencia entre el valor predicho para cada estado y si el jugador finalmente ganó o perdió. Para este entrenamiento se utilizaron datos generados por la política p_π tras entrenarla mediante aprendizaje por refuerzo.

Por último se combinan ambas redes mediante el uso de un MCTS, de forma ligeramente diferente a como hacemos en AlphaZero. Al igual que en AlphaZero, se utilizan las probabilidades dadas por la red neuronal y se selecciona en cada estado s la acción a que maximiza $Q(s,a)+U(s,a)$ (definida de la misma forma que en AlphaZero), se expande el nodo hoja al que llegamos y se actualizan los valores. Sin embargo, el valor v del nodo hoja s_L que expandimos se evalúa de dos formas. Por un lado, se utiliza el valor $v_\theta(s_L)$ proporcionado por la red v_θ. Por otro, se utiliza la política rápida p_π para jugar una partida completa y obtener el resultado z_L (1 si ha ganado o -1 si ha perdido). Después, estos valores se combinan mediante un hiperparámetro λ para obtener la evaluación final del estado:

$$v(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$$

Un punto importante es que en AlphaZero se elimina esta primera etapa de aprendizaje supervisado sobre bases de datos de expertos. Esto es doblemente interesante. Por un lado, se elimina la dependencia de bases de datos que pueden no existir o ser difíciles de obtener. Pero por otro, estas bases de datos están sesgadas por los conocimientos de los expertos, imponiendo un tope al rendimiento que puede alcanzar la inteligencia artificial. Si la red neuronal comienza aprendiendo de los humanos, pese a que sea capaz de encontrar ciertas líneas mejores a las que utilizan los humanos, también será más reticente a explorar ciertas jugadas diametralmente opuestas al juego humano pero que conducen a mejores resultados. Por ello, la eliminación del aprendizaje supervisado no se debe solo a una cuestión pragmática, sino también para eliminar el techo de rendimiento impuesto por imitar el juego humano.

3.6.2. AlphaGo Lee

Es la versión que se enfrentó a Lee Sedol en marzo de 2016, venciendo 4 a 1. Es una versión ligeramente mejorada de AlphaGo Fan, en la que se incrementó el tamaño de las redes neuronales y la potencia de cómputo. Sin embargo, el cambio principal es que la red de la política no se entrena contra las políticas anteriores, sino con los resultados del MCTS. Esto es algo que se utiliza en

[16]En este contexto, sobreaprendizaje significa que la política actual se centra en explotar las vulnerabilidades de la política rival más que en buscar las jugadas realmente buenas. En la sección 3.4 comentamos que esto teóricamente no debería suceder, pero aquí se utiliza una versión menos sofisticada del MCTS y los autores no habían estudiado tanto el algoritmo.

AlphaZero y, como recalcan los autores, es un cambio importante porque se acerca al aprendizaje sin intervención humana.

3.6.3. AlphaGo Master

Es la versión que en diciembre de 2016 y enero de 2017 se enfrentó a 60 jugadores humanos en servidores online, ganando 60-0 e incluyendo 3 partidas contra Ke Jie, el jugador con mayor elo del momento. Para ello utilizó la misma red neuronal, MCTS y entrenamiento por refuerzo que AlphaZero. Sin embargo, se mantiene el mismo uso de las simulaciones de p_π combinadas con la utilización de la red neuronal de valor para predecir el valor de una posición, y los datos extraídos del tablero (escogidos por los diseñadores) como entrada de esa red neuronal (en vez de introducir simplemente el tablero). Además, al igual que en las versiones anteriores el entrenamiento comenzó mediante aprendizaje supervisado a partir de ejemplos de partidas humanas.

3.6.4. AlphaGo Zero/AlphaZero

AlphaZero [20] es el algoritmo presentado en esta sección, mientras que AlphaGo Zero [22] es una versión previa especializada en jugar al go. Explicaremos aquí las diferencias fundamentales entre AlphaGo Zero y AlphaZero.

En primer lugar, en el go no se puede empatar (existen reglas específicas para evitarlo) y por ello solo se puede ganar $(+1)$ o perder (-1). En otros juegos como en el ajedrez o como es nuestro caso, el tres en raya y el Conecta 4, se puede además empatar (0). En AlphaGo Zero el valor v estimaba la probabilidad de que el jugador ganase, y lo que se buscaba era maximizar la probabilidad de ganar. En AlphaZero el valor v indica a qué jugador le es más favorable la posición y lo que se busca es maximizar el valor de nuestra posición.

La diferencia entre estas dos filosofías se puede entrever en *AlphaGo - The Movie*[17], el documental que narra el encuentro entre AlphaGo y Lee Sedol. Una de las cosas que más sorprende a los expertos es que AlphaGo no intenta ganar a su oponente por el máximo número de puntos, sino que busca maximizar la probabilidad de ganar, aunque sea por un único punto, realizando jugadas destinadas a asegurar la victoria a pesar de que a priori haya otras que le puedan reportar más puntos. Aunque es una versión anterior a AlphaGo Zero, su filosofía es la misma. Por el contrario, AlphaZero trataría de ganar por la máxima diferencia de puntos.

Por otro lado, en el go, tanto el tablero como las reglas son simétricas. Esto permite que, gracias a simetrías y rotaciones, haya para cada posición 8 que son equivalentes, como se aprecia en la figura 16. En AlphaZero no se presupone ningún invariante de este tipo sobre las reglas, por lo que esto no se puede explotar. En AlphaGo Zero sí, y se hacía de dos formas. En primer lugar, los datos de entrenamiento se aumentaban incluyendo todas las simetrías de cada posición, de tal forma que de cada posición se sacaban múltiples triplas de entrenamiento. En segundo lugar, al utilizar el MCTS se transformaba de forma aleatoria cada posición en una de las 8 posiciones equivalentes antes de evaluarla en la red neuronal. Por ello en [22] la llamada a la evaluación de la red neuronal se hace con:

[17] Disponible en `https://www.youtube.com/watch?v=WXuK6gekU1Y`.

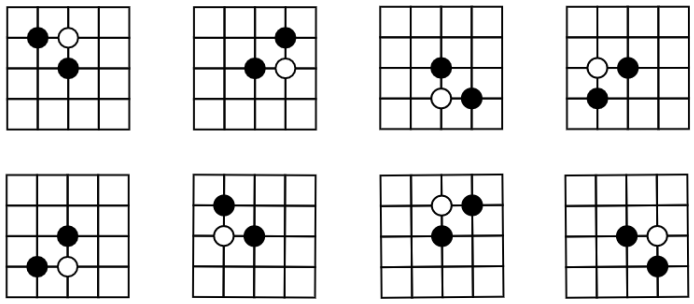


Figura 16: Ejemplo de todas las rotaciones/simetrías equivalentes sobre un tablero de go de dimensiones 5x5.

$$f_\theta(d_i(s)) = (d_i(\mathbf{p}), v)$$

siendo $d_i(s)$ uno de los estados equivalentes a s escogido de forma aleatoria.

Otra diferencia es que, como comentamos en la sección 3.4, en AlphaZero hay una única red neuronal que produce las evaluaciones y aprende al mismo tiempo, mientras que en AlphaGo Zero se separan, de forma que tenemos una red neuronal que produce las evaluaciones y otra que aprende.

Por último, ambos algoritmos utilizan la misma arquitectura de red neuronal. Aunque no sea una diferencia, es importante tener en cuenta que la arquitectura de la red en AlphaGo Zero está influenciada por las reglas del go. Concretamente, para cada pieza se almacenan los huecos libres adyacentes, de ahí que sea razonable usar filtros convolucionales de tamaño 3x3 (y así capturar la información de la pieza y las piezas que la rodean inmediatamente). A pesar de que en otros juegos esto no se cumple, se mantiene la arquitectura de la red en AlphaZero.

3.6.5. MuZero

La última evolución del algoritmo es MuZero [16]. Su objetivo no es lograr un mayor rendimiento, sino reducir el número de restricciones que ha de verificar el juego para que el algoritmo pueda ser aplicado. Concretamente, no necesita conocer las reglas del juego, como sí que necesitan sus antecesores. Para ello, en primer lugar aprende un modelo del entorno y posteriormente utiliza las ideas de AlphaZero sobre el modelo aprendido.

4. Implementación

En este capítulo vamos a implementar una versión reducida de AlphaZero en Python, que buscará resolver el juego Conecta 4. En esta sección solo vamos a comentar los fragmentos más importantes del código (por lo que algunas de las funciones se muestran incompletas)[18] y las consideraciones más importantes a la hora de implementarlo. Esta versión se puede adaptar de forma sencilla a otros juegos, y de hecho también analizaremos el comportamiento del algoritmo sobre las tres en raya.

4.1. Tablero

A pesar de que hemos presentado el algoritmo de AlphaZero mostrando ejemplos con el go (juego sobre el que se utilizó originalmente) y el Conecta 4 (porque es el juego principal para el que vamos a implementarlo), ya hemos mencionado que AlphaZero se puede adaptar a cualquier juego de mesa de dos jugadores en el que podamos proveer a AlphaZero con las reglas exactas y el estado del juego sea completamente observable. En nuestro código, de proveer a AlphaZero con las reglas se va a encargar la clase `Game`.

Los métodos que necesitamos implementar son:

- `available_moves()`: devuelve las acciones que son posibles en la posición actual.
- `make_move(move)`: se encarga de modificar el tablero realizando la acción `move`. Es el método fundamental de la clase, ya que aparte de actualizar el tablero también va a comprobar si se ha acabado la partida.
- `get_state()`: se encarga de devolver la representación del tablero que está esperando la red neuronal, que no tiene por qué coincidir con la representación interna que guardamos del tablero. Por ejemplo, en la representación interna del Conecta 4 se almacena el tablero actual (dos matrices, una por cada jugador, indicando en cada una de ellas dónde tiene colocadas las piezas ese jugador) y el jugador al que le toca jugar. Sin embargo, la red neuronal espera tres matrices, las dos primeras representando el tablero como hemos dicho anteriormente y la tercera representando a qué jugador le toca jugar.

Además de estos, hay unas pocas funciones auxiliares. Sin embargo, lo importante es que es sencillo modificar estas funciones para implementar otro juego distinto.

4.2. Red neuronal

La red neuronal utilizada es una simplificación de la red original. Aunque la mayoría de aspectos se mantienen iguales a los del artículo original, otros se simplifican para reducir el número de parámetros de la red y por lo tanto los tiempos que esta necesita para realizar las evaluaciones y ser entrenada.

[18] El código completo se puede encontrar en `https://github.com/alberto-maurel/Aprendizaje-por-refuerzo-Fundamentos-te-ricos-del-algoritmo-AlphaZero-e-implementaci-n`.

En primer lugar, como estado vamos a utilizar simplemente el tablero actual, sin incluir tableros anteriores como se hace en la implementación de DeepMind para el go. Con esto, reducimos el tamaño de la entrada y consiguientemente el número de parámetros de la red neuronal, acelerando la evaluación del estado. Este tablero va a estar representado por tres matrices binarias, cada una teniendo la misma dimensión que el tablero sobre el que jugamos (3x3 para el tres en raya, 6x7 para el Conecta 4). Cada una de las dos primeras matrices va a contener las piezas de cada uno de los jugadores: en una posición hay un 1 si tenemos una pieza o 0 si no[19]. La tercera matriz va a estar rellena de ceros si le toca jugar al primer jugador o de unos si le toca al segundo. En ambos juegos, el jugador al que le toca se puede deducir del número de piezas en el tablero. Sin embargo, si eliminamos esta tercera matriz estaríamos obligando a la red neuronal a aprender esto, lo cual sería bastante complejo. Es una "ayuda" que damos a la red neuronal para que pueda aprender de forma más sencilla a evaluar las posiciones.

En cuanto a la arquitectura de la red se realizan pocas modificaciones. Se reduce de 39 a 4 el número de bloques residuales y de 256 a 32 el número de filtros en las capas convolucionales. En el bloque de la política la capa de salida se adapta al juego, de forma que la política tenga n valores, siendo n el número de acciones posibles en el juego (7 para el Conecta 4 y 9 para el tres en raya). Por último, se incluye tras la última capa densa una capa de activación *softmax*. Esta permite convertir los valores calculados por la red en probabilidades mediante la función

$$\pi(s)(a_t) = \frac{e^{u_t}}{\sum_{k=1}^{7} e^{u_k}}$$

siendo u_t el valor calculado por la neurona t de la última capa densa de la política. En el artículo original lo que se devolvía eran directamente los valores de la red sin normalizar y posteriormente había que convertirlos en probabilidades.

En cuanto a los aspectos generales de la red, como función de pérdida se emplea la propuesta en el artículo sin utilizar regularización L2, es decir:

$$l = (z - v)^2 - \pi^t \log p$$

y se fija el ratio de aprendizaje a 10^{-4}. Se elimina la regularización L2 por dos motivos. El primero es que es problemática de implementar en Tensorflow, puesto que de forma sencilla se puede añadir regularización a una única capa, pero nosotros queremos añadirla a los pesos de toda la red. Existen formas de hacerlo pero eso implica complicar la implementación. Por otro lado, con las capas de normalización por lotes ya se introduce una cierta regularización, la cual debería ser suficiente para los juegos que vamos a estudiar. En el apéndice A se incluyen figuras con la arquitectura exacta de la red.

4.3. MCTS

La implementación del árbol de búsqueda de Monte Carlo es la parte más delicada del algoritmo. Para cada partida, AlphaZero va a tener un MCTS que va a estar guiado por una red neuronal. Si

[19]En el tres en raya y en el Conecta 4 solo hay un tipo de pieza, para otros como el ajedrez necesitaríamos más números.

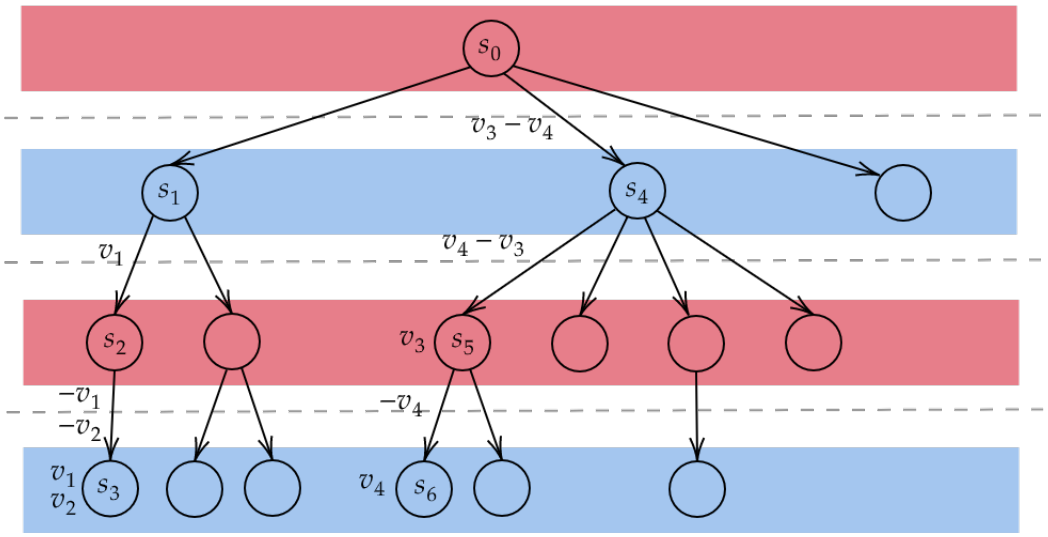


Figura 17: MCTS que tiene como raíz el estado s_0. En rojo tenemos los estados en los que le toca mover al jugador 1 y en azul al jugador 2.
Supongamos que lanzamos una simulación desde s_1 que tiene como nodo hoja el nodo s_3. Entonces, la red evaluará s_3 con valor v_1 y actualizará las aristas con los valores v_1 y $-v_1$. Supongamos ahora que hubiéramos lanzado la simulación desde s_2 y que de nuevo tenemos s_3 como nodo hoja. La evaluación nos devolvería un valor v_2 para s_3, pero como en ambos casos habríamos utilizado la misma red neuronal (incluso usando MCTS diferentes), la evaluación es la misma y $v_1 = v_2$. Por lo tanto, la arista que actualizamos en ambas simulaciones se habría actualizado en ambas ocasiones al mismo valor: $-v_1 = -v_2$. Vemos por lo tanto que la actualización de las aristas es independiente del nodo desde el que se haya lanzado la simulación.
Supongamos ahora que lanzamos una simulación desde s_0 y que tiene a s_5 por nodo hoja: s_5 se evalúa a v_3 y se actualizan consecuentemente las aristas de la trayectoria. Ahora lanzamos otra simulación desde s_0, y en este caso s_6 es el nodo hoja, evaluado a v_4. Actualizamos las aristas de la trayectoria y de nuevo los valores obtenidos son coherentes con el punto de vista del jugador que ejecuta la acción.

tuviéramos dos jugadores controlados por AlphaZero, cada uno tendría un MCTS que al principio de la partida tendría un único nodo: el estado inicial del tablero. El MCTS se va a poblar a lo largo de la partida y va a ser el que tenga la potestad última de decidir la acción ejecutada, independientemente de la acción preferida por la red neuronal (aunque la política propuesta por la red neuronal dirige al MCTS). Una vez que la partida se termina, el MCTS se descarta, y tan solo se guarda la información que necesitamos para el aprendizaje por refuerzo. Sin embargo, como ya introdujimos en la sección 3.5, a la hora de entrenar se va a utilizar un único MCTS que controla a ambos jugadores. En la figura 17 se puede ver su funcionamiento.

Serán dos funciones las que implementarán la funcionalidad del MCTS: `simulation`, donde se hace una simulación desde el estado actual del tablero hasta un estado hoja; y `make_movement`, donde en base a múltiples simulaciones se decide el movimiento a realizar.

Comenzamos por `simulation`. En primer lugar, seleccionamos las acciones a_t que maximicen $Q(s_t, a_t) + U(s_t, a_t)$ hasta que lleguemos a un nodo hoja

```
def simulation(self, board, model):
    actions = []

    while(True):
        upper_confidence_vector = #calculate the value of each action
        action_taken = numpy.argmax(upper_confidence_vector)
        actions.append(action_taken)
        win = current_board.make_move(action_taken)

        if(current_node.edges[action_taken].nextState != None):
            current_node = current_node.edges[action_taken].nextState
```

Una vez que hemos llegado a una hoja, expandimos el nodo:

```
        else:
            if win: #win
                if len(actions) % 2 == 0:
                    total_value = -1
                else:
                    total_value = 1
            elif not current_board.is_possible_to_move(): #draw
                total_value = 0
            else:
                evaluation = model.predict([current_board.get_state()])
                total_value = evaluation[1][0][0]
                if len(actions) % 2 == 1:
                        total_value = -total_value
                current_node.edges[action_taken].nextState=MCTS(evaluation[0][0])
            break
```

Al expandir el nodo, tenemos en cuenta si es un estado en el que se termina la partida o no. En el primer caso, ya conocemos el valor del estado: 0 si se empata, $+1$ si hemos ganado o -1 si hemos perdido. En el segundo caso necesitamos que la red neuronal evalúe el estado al que hemos llegado y estime su valor y la política a seguir.

De esto se encargan las líneas 21-25. Primero hacemos una llamada a `model.predict` con el estado al que hemos llegado y almacenamos en `evaluation` la política y el valor devueltos. El valor lo utilizaremos para actualizar las aristas que hemos explorado en esta simulación, y tenemos que cambiar su signo dependiendo de si en el nodo al que hemos llegado nos toca jugar a nosotros o a nuestro oponente. Por último, en la línea 25 se crea el nuevo vértice, correspondiente al nuevo estado que hemos simulado, y se asignan a sus aristas las probabilidades indicadas por la política.

Esta es una línea clave en el código. Como vimos en la sección 3.3.2, durante la fase de expansión las probabilidades proporcionadas por la política dada por la red neuronal se modifican con una distribución de Dirichlet, como se puede ver en el constructor del MCTS.

```
def __init__(self, probs):
    dirich= numpy.random.dirichlet([DIR_ALPHA for _ in range(0,NUM_ACTIONS)],1)[0]
    self.edges = []
    for i in range(0, NUM_ACTIONS):
        self.edges.append(Edge((1 - eps) * probs[i] + eps * dirich[i]))
```

En [22] indican que el α escogido para la distribución es 0,03. Sin embargo, en *Lessons from implementing AlphaZero* [14], una serie de artículos en la que estudian el artículo de AlphaZero y lo

implementan también para el Conecta 4, estudian el valor que debería tomar α para este juego. La primera observación que hacen es que el equipo de DeepMind utiliza distintos valores dependiendo del juego: 0,3 para el ajedrez, 0,15 para el shogi y 0,03 para el go. Esto lo relacionan con la media de movimientos legales en cada juego: 35 en el ajedrez, 92 en el shogi y 250 en el go, datos tomados de publicaciones previas. Es razonable pensar que el valor de α está relacionado con el número de movimientos válidos, porque la función del ruido es modificar la probabilidad de los movimientos para incrementar la exploración.

En el tres en raya y en el Conecta 4 el número de jugadas legales en promedio es 4 [3], y utilizando unas aproximaciones basadas en los parámetros usados por DeepMind prueban con valores de α entre 1 y 2,5. Valores superiores a 1 para α generan vectores con ruido más equilibrado. Esto será muy importante en el Conecta 4, puesto que querremos asegurarnos de explorar lo suficiente, y en la práctica en la implementación utilizaré $\alpha = 1{,}5$.

Por último, actualizamos el valor de los nodos recorridos de acuerdo al resultado obtenido:

```
27          for action in actions:
28              current_node.edges[action].updateEdge(total_value)
29              current_node = current_node.edges[action].nextState
30              total_value = -total_value
```

para lo cual es clave cambiar el signo del valor obtenido en cada paso para expresar el valor obtenido desde el punto de vista del jugador al que le toca jugar en ese estado (y como juegan de forma alterna tenemos que alternar los signos).

La función `makeMove` por su parte utilizará la función anterior para poblar el MCTS y posteriormente escogerá la acción realizada en base al número de veces que se haya probado cada acción, de acuerdo a lo expuesto en la sección 3.3.2, en la fase de elección:

```
1 def makeMove(self, board, model):
2     for _ in range(0, NUM_SIMULATIONS):
3         self.simulation(board, model)
4 
5     probs = #probability of each action, proportional to the exponentiated visit
      count
6     action = numpy.random.choice(a=[i for i in range(0, NUM_ACTIONS)], p=probs)
```

A la hora de calcular las probabilidades, la fórmula utilizada es:

$$\pi(a|s_0) = \frac{N(s_0, a)^{1/\tau}}{\sum_b N(s_0, b)^{1/\tau}}$$

En el artículo se especifica que para las 30 primeras jugadas se escoge $\tau = 1$ y de ahí en adelante se utiliza $\tau \approx 0$. Esto significa que en la apertura se van a efectuar las jugadas de forma proporcional al número de veces visitadas y a partir del movimiento 30 se va a seleccionar siempre la jugada más visitada. En el entrenamiento para el Conecta 4 voy a utilizar $\tau = 1$ para las primeras 8 jugadas, y simplemente coger la jugada más visitada (que equivale a $\tau \approx 0$) de ahí en adelante. Esto puede provocar que durante el entrenamiento algunas partidas se pierdan en estas 8 primeras jugadas (porque un jugador ponga simplemente 4 fichas en raya y el otro dé probabilidades altas a aquellas acciones que lo evitan pero de forma aleatoria se seleccionen las acciones menos probables). Sin embargo, esto no debería ser un problema, puesto que la red neuronal aprenderá qué posiciones son

las perdedoras y tenderá a visitar más las posiciones que lo evitan, incrementando la probabilidad de las acciones buenas en la política y evitando que esto se repita en el futuro.

Por último, actualizamos la raíz del MCTS utilizando la arista correspondiente a la acción realizada y devolvemos la acción, estado y probabilidades de cada acción que posteriormente utilizaremos en el entrenamiento.

```
    #Update the MCTS and modify its root
    return action, copy.deepcopy(board).get_state(), probs
```

4.4. Programa principal

Con todos los componentes construidos, ya solo nos queda unirlos. De esto se encargan dos funciones: `self_play`, en la que AlphaZero juega contra sí mismo y genera ejemplos de aprendizaje, y `main`, donde se recogen esos datos y se entrena el modelo.

Comenzamos analizando la función `self_play`, a la cual le pasamos como argumento la red neuronal `(model)`. Lo primero que tenemos que hacer es crear una nueva partida y el MCTS que, como ya explicamos en la subsección anterior, compartirán ambos jugadores.

```
def self_play(model):
    board = Game()
    evaluation = model.predict([board.get_state()])
    mcts = MCTS(evaluation[0][0])
```

Mientras que no se haya terminado la partida, el jugador al que le toca escoge la jugada a realizar. Actualizamos el tablero, informamos al otro jugador y guardamos los datos que necesitaremos posteriormente

```
    while not finished and not draw:
        action, state, mcts_distribution = mcts.makeMove(board, model)
        states.append(state)
        distributions.append(mcts_distribution)

        finished = board.make_move(action)
        if not finished and not board.is_possible_to_move():
            draw = True
```

A lo largo de la ejecución hemos ido almacenando tanto los estados por los que hemos pasado como las distribuciones devueltas por el MCTS. Sin embargo, no es hasta el final cuando conocemos el resultado de la partida y podemos asignar la recompensa correspondiente, dependiendo de quién ha ganado o si han empatado. Por último devolvemos los estados por los que hemos pasado, la política calculada por el MCTS y la recompensa obtenida.

```
    if finished:
        for i in range(0, len(states)):
            if i % 2 == len(states) % 2:
                rewards.append([1.0])
            else:
                rewards.append([-1.0])
    else: #draw == True
        for i in range(0, len(states)):
            rewards.append([0.0])
    return states, distributions, rewards
```

Con esto, construir la función `main` es sencillo. Realizaremos varias iteraciones de entrenamiento. En cada una de ellas realizamos varias simulaciones empleando la red neuronal actual, y luego las empleamos para entrenar la red.

```
def main():
    model = #build the neural network

    for i in range(0, NUM_ITERATIONS):
        for _ in range(0, NUM_GAMES):
            states, distributions, rewards = self_play(model)
            states_data.extend(states)
            distributions_data.extend(distributions)
            rewards_data.extend(rewards)

        history=model.fit(x=np.array(states_data),y=[np.array(distributions_data),
                          np.array(rewards_data)], batch_size = 16, epochs = 8)
        states_data = []
        distributions_data = []
        rewards_data = []
```

Tanto el número de iteraciones de entrenamiento (`NUM_ITERATIONS`) como el número de partidas jugadas en cada iteración (`NUM_GAMES`) dependerán del juego, como veremos más adelante.

5. Resultados

En esta sección se analizan los resultados obtenidos tras aplicar al algoritmo al tres en raya y al Conecta 4.

5.1. Tres en raya

Comenzaremos analizando el comportamiento del algoritmo sobre el tres en raya. Dado que es un juego más simple que el Conecta 4, podremos apreciar mejor el funcionamiento del algoritmo y su aprendizaje.

Para entrenar el algoritmo vamos a jugar 1250 partidas, divididas en 125 iteraciones de 10 partidas. En cada partida, construiremos un MCTS desde cero, jugaremos la partida y almacenaremos la información recabada. Por cada jugada se ejecutan 200 simulaciones. Para dirigir estas simulaciones usamos la política $\mathbf{p}$ proporcionada por la red neuronal, la cual modificamos introduciendo ruido por medio de una distribución de Dirichlet de parámetro α. Como vimos en la sección 3.3.2, en la fase de expansión ponderamos ambos términos con el parámetro ϵ siguiendo la expresión

$$\epsilon\, p_a + (1-\epsilon)x_a$$

Durante el entrenamiento fijamos $\alpha = 1{,}5$ y $\epsilon = 0{,}25$.

Una vez realizadas las simulaciones, escogeremos cada acción de forma proporcional al número de veces que la hemos simulado. Como introdujimos en la sección 3.3.2 en la etapa de elección, para ello se utiliza el parámetro τ

$$\pi(a|s_0) = \frac{N(s_0,a)^{1/\tau}}{\sum_b N(s_0,b)^{1/\tau}}$$

Durante el entrenamiento se fija $\tau = 1$ durante toda la partida para incrementar la exploración.

Al final de cada iteración se seleccionan de forma aleatoria 500 jugadas de entre todas las partidas jugadas hasta el momento (no solo de las de esta iteración). Esto se hace por analogía a AlphaZero, como se explicaba en la sección 3.4, que escoge sus datos de entrenamiento entre las últimas 500000 partidas jugadas. En la siguiente iteración de entrenamiento, reemplaza las partidas más antiguas por otras más nuevas, y vuelve a entrenar sobre todas ellas (y por consiguiente la mayor parte de los datos de entrenamiento en dos iteraciones consecutivas son los mismos). Aquí como la cantidad de partidas jugadas es muy inferior no vamos a eliminar las partidas más antiguas.

Para entrenar, se utilizan *batches* de tamaño 32 durante 4 épocas. El ratio de aprendizaje se fijaba a 10^{-4}. El tiempo total de entrenamiento ha sido de unas 3 horas.

Para las partidas definitivas se ha escogido en todo momento la acción que más veces había explorado el MCTS en las simulaciones ($\tau \approx 0$) y se ha eliminado el ruido (utilizando $\epsilon = 0$) para facilitar la comparación.

Primera jugada

La teoría dice que, independientemente de la posición escogida por el primer jugador para colocar su primera ficha, si ambos jugadores juegan de forma óptima a partir de la segunda jugada, el resultado final es un empate.

Si eliminamos las rotaciones y las imágenes especulares tenemos 3 posibles primeros movimientos: colocar la ficha en el centro, en una esquina o en el centro de un lateral. A pesar de que cualquiera de ellos conduce al empate en una partida contra un jugador óptimo, AlphaZero no es un jugador óptimo. Al comienzo del entrenamiento, cuando su red neuronal aún no ha aprendido y no dirige correctamente las simulaciones, el algoritmo no puede explorar todas las líneas de juego. Realizamos tan solo 200 simulaciones por jugada, frente a las 255168 partidas completas diferentes[20] que se pueden jugar, por lo que AlphaZero cometerá errores. Al final del entrenamiento, aunque la red neuronal haya aprendido y sea capaz de guiar correctamente las simulaciones, el utilizar el parámetro $\tau = 1$ puede hacer que en un reducido número de ocasiones se seleccione una jugada que nuestro algoritmo no considera óptima, con el propósito de explorar el árbol de juego. Por lo tanto, durante el entrenamiento estamos en un enfrentamiento entre jugadores que pueden confundirse, y lo que nos conviene hipotéticamente es realizar la jugada que pueda "confundir" más al rival y llevarnos a la victoria en más ocasiones.

La reflexión de *Tic Tac Toe: First Move Strategy* [13] sigue esta línea, incidiendo en que no todas las jugadas iniciales dejan al oponente las mismas posibilidades. Como se puede ver en el apéndice B, mientras que colocando la primera ficha en el centro y en el lateral nuestro oponente tiene 4 posibles jugadas óptimas, si se coloca en una esquina el oponente solo tiene 1 jugada óptima. Por ello, en este caso sería aconsejable colocar la primera ficha en una esquina. En la figura 18 se muestra la evolución de la red de política. Inicialmente, todas las acciones son equiprobables. A las 10 iteraciones (100 partidas) ya favorece al centro y a algunas esquinas, algo razonable porque estas líneas de juego son más sencillas que las que comienzan en un lateral. A partir de este momento varía ligeramente la probabilidad de cada acción hasta que termina el entrenamiento (1250 partidas), donde favorece a las esquinas. Tras eliminar el ruido y pasar a elegir de forma determinista la acción más simulada por el MCTS, AlphaZero colocará siempre su primera ficha en la casilla inferior derecha.

Segunda jugada

La segunda jugada posiblemente es el momento más delicado para el algoritmo. Como hemos mencionado previamente, independientemente de dónde haya puesto su primera ficha el rival hay ciertas jugadas que nos condenan a la derrota si el rival juega de forma óptima. Sin embargo, como

[20] Lo que aquí estamos contando son secuencias de movimientos diferentes que llevan al final de una partida (bien porque un jugador ha ganado o bien porque se ha llenado el tablero). En realidad, como AlphaZero no simula las partidas hasta el final, sino jugada a jugada (y en cada simulación expande solo una jugada que no había sido probada antes, no una partida entera), necesitaríamos realizar 549946 simulaciones para explorar todo el árbol. Si tenemos en cuenta el número de estados diferentes, este es mucho menor, 5478 [7]. Sin embargo, debido a la forma que se ha implementado el MCTS, si llegamos dos veces al mismo tablero pero realizando las acciones en un orden diferente llegaremos a un nodo diferente. Esta implementación es útil porque nos permite descartar las simulaciones de líneas de juego que finalmente no se han seguido, que en proporción deberían ser más que las líneas de juego comunes en las se jugaron las mismas acciones en orden distinto. Esto es especialmente útil para juegos más grandes como el Conecta 4.

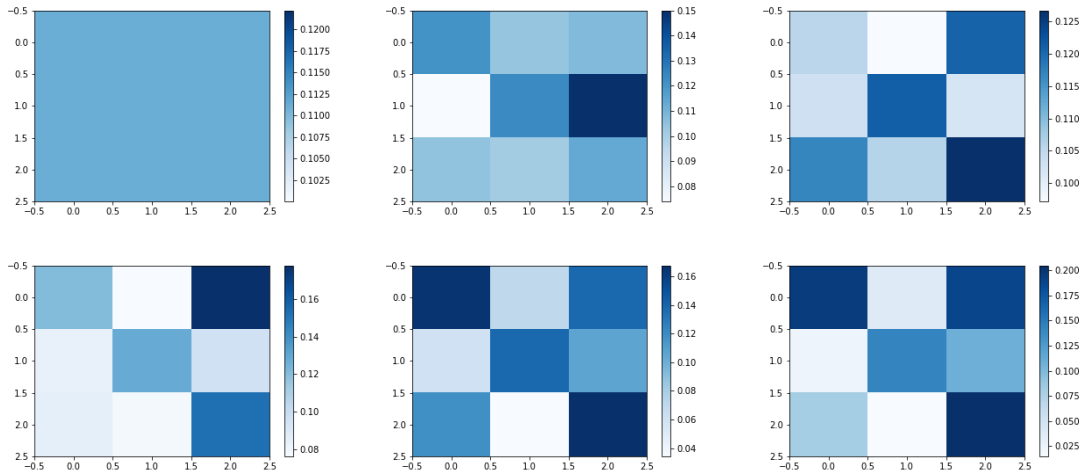


Figura 18: Probabilidad de cada acción propuesta por la red neuronal en las iteraciones 0, 5, 10, 25, 50 y 125. Los diagramas no son simétricos debido a que de forma aleatoria se han comenzado explorando unas acciones más que otras. Como esas acciones han dado buenos resultados, se ha incrementado su probabilidad frente a otras equivalentes pero que no se habían explorado tanto. Como consecuencia de esto, esas acciones se han seguido simulando en más ocasiones, provocando que se mantengan las diferencias de probabilidad. Las acciones concretas favorecidas varían si volvemos a entrenar desde cero el sistema, pero parece que suele favorecer a las esquinas.

se puede ver en el apéndice B, estas líneas de juego tienen longitud seis[21]. Como nosotros estamos realizando 200 simulaciones, si exploramos de forma equiprobable todas las acciones posibles podemos explorar hasta unas tres jugadas por delante[22]. Por lo tanto, para ser capaz de evitar la derrota, necesariamente la política y el valor proporcionados por la red neuronal tienen que dirigir correctamente al MCTS para identificar las variantes óptimas.

Un caso muy interesante es en el que el primer jugador coloca su ficha en uno de los extremos. La jugada óptima en este caso es que el segundo jugador coloque su primera ficha en el centro. En la figura 19 se representa la probabilidad propuesta por la red neuronal de colocar la ficha en el centro supuesto que el rival ha puesto la ficha en uno de los extremos, viendo que efectivamente la red neuronal aprende a defenderse. Nótese que esta es la probabilidad dada por la política de la red neuronal, pero el MCTS es el que decide finalmente la jugada en base a las simulaciones, por lo que esta probabilidad simplemente dirige las simulaciones. En el apéndice B se pueden observar las simulaciones realizadas por el MCTS y la acción final realizada.

En las figuras 20 y 21 se representa la probabilidad de escoger una acción óptima en los otros dos casos (empezando desde el centro y desde un lateral, respectivamente). Vemos de nuevo que AlphaZero es capaz de defenderse y que la red neuronal da una probabilidad mucho mayor a los movimientos buenos que a los malos, aunque para ciertos movimientos iniciales el aprendizaje es

[21] Si nos situamos en la figura 41, ahora mismo estamos en alguno de los estados de la izquierda (solo ha colocado el oponente su primera ficha). Tras colocar 4 fichas, llegamos al estado con una doble amenaza, y aún quedan dos jugadas más hasta que el primer jugador gane. Por lo tanto, para llegar al final de la partida necesitamos 6 jugadas.

[22] En la jugada 2 tenemos 8 posibles acciones, en la jugada 3 tenemos 7 posibles acciones y en la jugada 4 tenemos 6 posibles acciones, lo que hacen un total de $8 \times 7 \times 6 = 336$ simulaciones necesarias para explorar todas las líneas de juego.

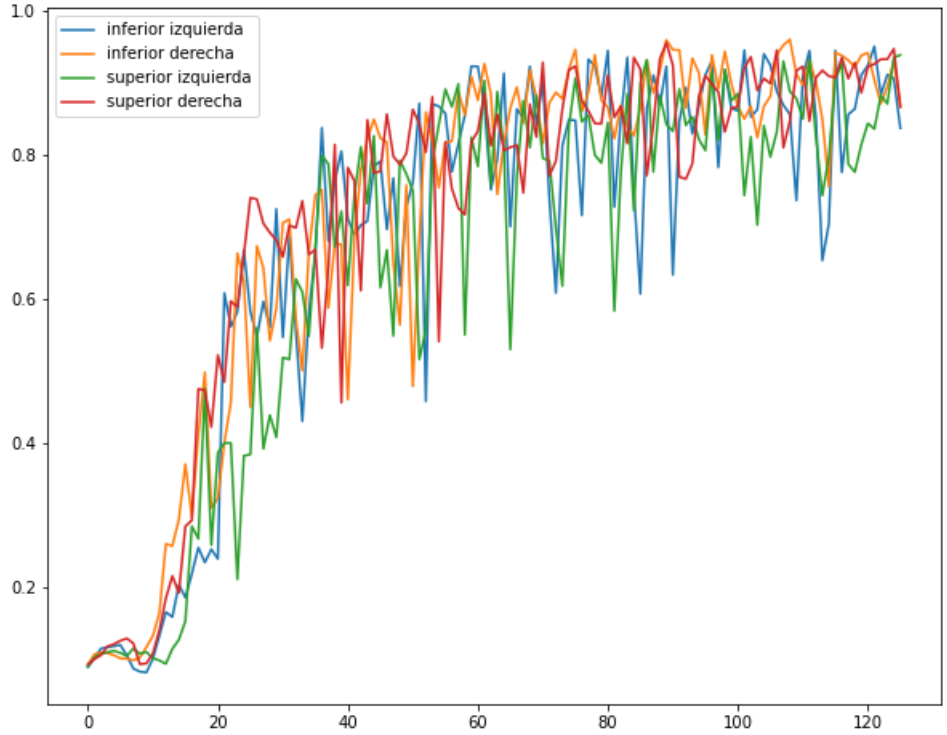


Figura 19: Probabilidad que la red neuronal de AlphaZero otorga en su política al movimiento óptimo como segundo jugador, supuesto que el primer jugador ha colocado su ficha en una esquina. Se calcula para cada una de las posibles esquinas de inicio para estudiar los efectos de las rotaciones sobre el aprendizaje de la red neuronal. La única jugada óptima es colocar nuestra primera ficha en el centro. Vemos que el aprendizaje es muy similar para todas las esquinas, y que en a la iteración 60 (600 partidas) la acción correcta tiene una probabilidad superior al 80 %.

más lento y sería interesante incrementar la exploración a lo largo del entrenamiento para asegurar el aprendizaje.

Líneas de juego subóptimas

Sin embargo, nosotros hemos entrenado el algoritmo jugando contra sí mismo. Como introdujimos anteriormente, es posible que se produzca sobreaprendizaje y que cada uno de los dos jugadores converja hacia una estrategia adaptada a su oponente y no contra un jugador general. Lo que estamos aprendiendo es una función de valor vinculada a la política predicha. Si la política propuesta es óptima, entonces la función de valor será un buen indicador de cuán bueno es un estado. Sin embargo, si en un cierto estado la política propuesta no es buena, la función de valor puede no reflejar realmente si un estado es favorable. Puede ocurrir que AlphaZero por cualquier motivo (como falta de exploración de una línea de juego) no encuentre una buena jugada, y como está jugando contra sí mismo, que el oponente tampoco considere esa jugada convergiendo hacia una política subóptima.

Esto en la práctica se puede traducir en que AlphaZero esté en una situación en la que puede forzar la victoria y decida conformarse con un empate, porque en su entrenamiento ha visto que en

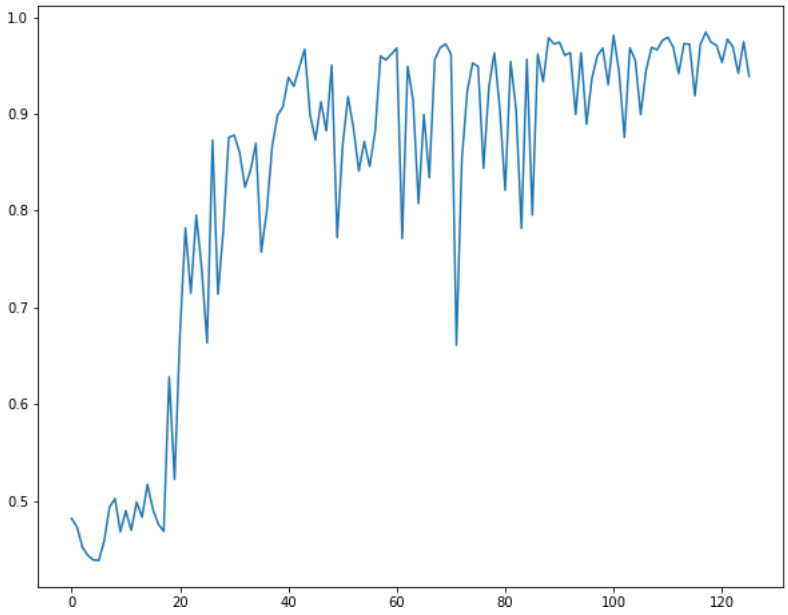


Figura 20: Probabilidad que la red neuronal de AlphaZero otorga en su política al movimiento óptimo como segundo jugador, supuesto que el primer jugador ha colocado su ficha en el centro. Las jugadas óptimas son colocar nuestra primera ficha en una esquina (4 posibilidades) y las incorrectas son colocarla en el centro de un lateral (4 posibilidades). Vemos que en la iteración 40 (tras 400 partidas jugadas), la red neuronal ya da menos de un 5 % de probabilidad a las jugadas incorrectas, y se va estabilizando este porcentaje a lo largo del entrenamiento.

todas las otras líneas en las que sí que jugaba de forma óptima el resultado esperado era empate. O que pierda si en una situación igualada ha ignorado una jugada crítica.

Para entender un poco más este fenómeno tenemos que remontarnos a la sección 3.3.2, a la fase de simulación en la que elegimos qué acción simular en cada momento. Como se vio en esta sección, la acción a_t escogida en el estado s_t en la simulación actual es

$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$

siendo

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Inicialmente $Q(s_t, a)$ es 0, puesto que no hemos hecho ninguna simulación y no hemos calculado ningún valor. En cuanto al término $U(s_t, a)$, vemos que si la probabilidad otorgada por la red neuronal ($P(s_t, a)$) es cercana a 0, entonces ese término es también cercano a cero, y por ende es posible que no simulemos nunca esa acción[23].

Ahí es donde entra en juego el ruido. A pesar de que la red neuronal otorgue probabilidad cero[24] a una acción, mediante el ruido podemos aumentar la probabilidad de dicha acción lo suficiente como

[23] Por poner un ejemplo, durante el entrenamiento la red neuronal ha otorgado a acciones óptimas una probabilidad de 10^{-3}. Si realizamos 200 simulaciones, entonces $P(s, a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \leq 0{,}02$, con lo que si el valor obtenido por otra acción es ligeramente no negativo nunca se probaría esta acción.

[24] Aunque realmente esa acción sea muy buena puede suceder que la red le otorgue probabilidad cero. Por ejemplo,

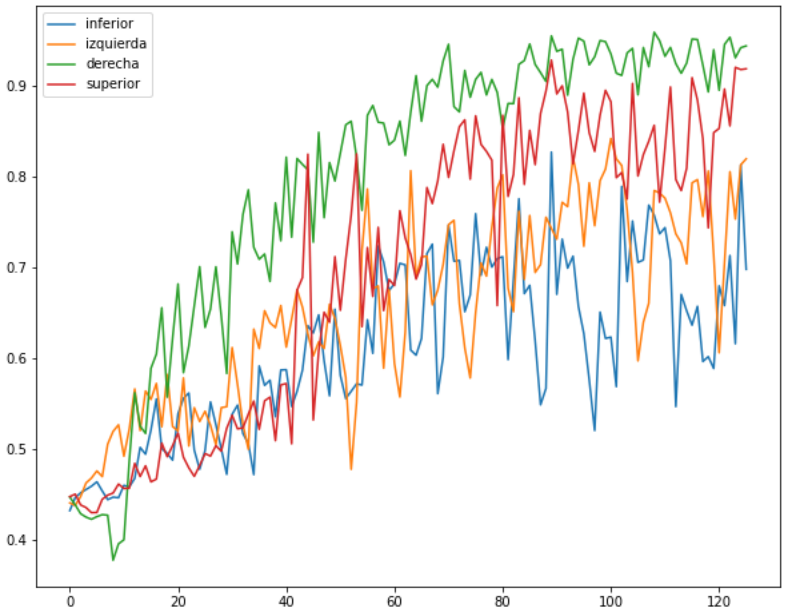


Figura 21: Probabilidad que la red neuronal de AlphaZero otorga en su política al movimiento óptimo como segundo jugador, supuesto que el primer jugador ha colocado su ficha en el centro de un lateral. Las jugadas óptimas para esta apertura se pueden consultar en el apéndice B. Para todas las rotaciones del tablero la red neuronal aprende las jugadas óptimas, aunque para algunas lo hace más lento que para otras. Si nos fijamos en la figura 18, vemos que la red neuronal ha aprendido más para aquella rotación más frecuente (la que pone la primera ficha en el centro del lado derecho), y menos para la más infrecuente (lado inferior). En el apéndice B se puede ver que independientemente de la probabilidad asignada por la red neuronal a las jugadas óptimas, el MCTS centra su simulación en las jugadas óptimas y termina realizando un movimiento acertado.

para que el MCTS la simule alguna vez. Si realmente es una acción mucho mejor, el término $Q(s_t, a)$ será grande tras varias simulaciones y obligará a que esa acción se siga simulando, convirtiéndola en la acción seleccionada. En la versión final se elimina el ruido para facilitar su estudio, pero manteniendo el ruido debería seguir funcionando correctamente.

Para comprobar que no sucedía esto, he enfrentado a la versión final contra humanos[25]. De las 28 partidas jugadas, 14 han sido como primer jugador y 14 como segundo jugador. De las partidas como primer jugador, ha ganado 7 y ha empatado otras 7. De las victorias, 5 se han debido a acciones subóptimas del rival a partir de las cuales AlphaZero ha sido capaz de forzar la victoria y 2 debido a errores claros del rival. De las partidas como segundo jugador, AlphaZero ha empatado 11 y ha ganado 3, aunque las victorias se han debido a errores claros del rival. No he apreciado

si el ratio de aprendizaje de la red es muy grande puede suceder que tras unas pocas iteraciones la probabilidad de una acción sea baja y dé la casualidad de que en ninguna de esas iteraciones se haya seleccionado la acción buena. También puede suceder que, dado que no entrenamos con todas las jugadas simuladas sino solo con unas pocas, que a pesar de que hayamos visto en una simulación que una línea es buena, a la hora de entrenar no se utilice ese ejemplo y la red neuronal no llegue a aprenderlo. En la práctica el proceso de aprendizaje está sujeto a mucha aleatoriedad y nuestro objetivo es tratar de garantizar que estos errores no ocurran, y que en caso de que sucedan se puedan enmendar.

[25] En la página `https://papergames.io/es/tres-en-raya`, contra jugadores aleatorios y variando tras varias partidas contra el mismo.

además que no haya sido capaz de aprovechar un error del rival. Por otro lado, en el apéndice B se encuentran las líneas de juego que sigue cuando el segundo jugador no hace una acción óptima, y en todas ellas logra la victoria.

Conclusión

En definitiva, vemos que con estos parámetros el algoritmo ha aprendido a jugar de forma óptima. Es capaz de empatar la partida contra jugadores óptimos y de ganar la partida cuando el contrario realiza un movimiento incorrecto. Si entramos más al detalle vemos que la mayor parte de las evaluaciones de la política y el valor de una posición son coherentes con lo esperado y reflejan la situación del tablero. El único problema apreciable es que aprende de forma desigual para ciertas posiciones que son equivalentes. Esto posiblemente se debe a la exploración realizada durante el aprendizaje, y se podría solucionar rotando/reflejando el tablero a la hora de entrenar y evaluar la posición, al igual que se hacía en AlphaGo Zero. Sin embargo, aquí no se ha implementado para respetar el algoritmo original y para ver el efecto de las rotaciones en el mismo. Por último, en esta sección hemos estudiado principalmente las predicciones de la red neuronal, pero no se ha visto el MCTS (que es el que realmente dirige el juego) en funcionamiento. En el apéndice B se incluyen varias figuras ilustrando su funcionamiento en detalle.

5.2. Conecta 4

En el caso del Conecta 4, para entrenar el algoritmo vamos a jugar 2000 partidas divididas en 200 iteraciones de 10 partidas, de igual manera a como hacíamos para el tres en raya. Realizaremos 200 simulaciones por movimiento y fijamos $\alpha = 1,5$ y $\epsilon = 0,25$. Durante el entrenamiento se utiliza $\tau = 1$ para los primeros 8 movimientos y $\tau \approx 0$ para el resto. Esto nos permite explorar durante los 8 primeros movimientos diferentes variantes y posteriormente escoger siempre la acción que ha sido simulada más veces.

Al final de cada iteración se seleccionan de forma aleatoria 750 jugadas de entre las últimas partidas jugadas hasta el momento (no solo de las de esta iteración). Para entrenar se utilizan batches de tamaño 32 durante 8 épocas. El ratio de aprendizaje se fija a 10^{-4}, y el tiempo de entrenamiento ha sido de unas 58 horas.

Enfrentamientos entre distintos modelos

Dado que hay 7 columnas, tenemos $7^3 = 343$ posibilidades para los siguientes 3 movimientos. En el caso de que probemos todos de forma equiprobable (como se espera que suceda al inicio, cuando la red neuronal aún no ha entrenado), nuestro modelo estudiará todas las líneas de longitud 2 y algunas de longitud 3. Esto se traduce en que para todas las jugadas que podamos hacer probaremos todas con las que puede responder el rival, por lo que si el oponente puede ganarnos en el siguiente turno en una única posición deberíamos ser capaces de bloquearla. Sin embargo habrá situaciones insalvables (en las que el rival tiene varias posiciones donde nos puede ganar e independientemente de la acción que realicemos perdemos) que el sistema será incapaz de evitar a no ser que sea capaz de estudiar la situación desde varios turnos antes, cosa que no puede hacer sin entrenamiento. Sin

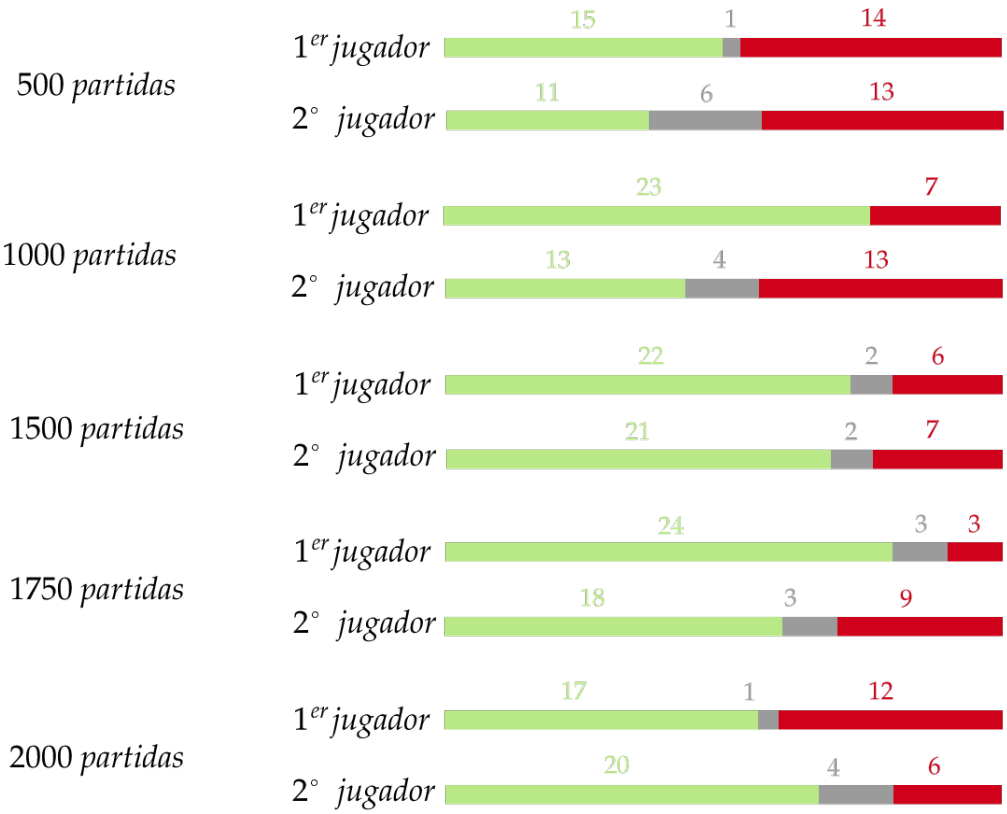


Figura 22: Resultados del enfrentamiento de 30 partidas entre AlphaZero entrenado y sin entrenar tras un número distinto de partidas de entrenamiento. Para cada versión entrenada se indica el número de victorias jugando primero y jugando segundo.

embargo, si se decrementa la probabilidad de las acciones buenas puede suceder que no seamos capaces de evitar la derrota aunque esta sea evidente.

Para estudiar la evolución del modelo vamos a comparar los AlphaZero entrenados con el AlphaZero sin entrenar (en el que la red neuronal sin entrenar no dirigirá realmente al MCTS y simplemente se probarán más o menos todas las jugadas de forma equiprobable). Para esta comparación se jugarán 30 partidas entre los modelos (1 hora aproximadamente) por cada tipo de fichas (30 como primer jugador y 30 como segundo), con $\tau = 1$ para las 4 primeras jugadas (para que las partidas jugadas sean diferentes, si no todas las partidas serían la misma) y $\tau \approx 0$ para el resto. El resto de parámetros se mantienen iguales a los del entrenamiento, incluido el ruido (a diferencia de lo que hacíamos en el tres en raya, donde lo eliminábamos). Sobre esto volveremos más adelante. En la figura 22 se muestra el resultado de las partidas.

Vemos que el algoritmo claramente aprende frente a la versión inicial, a la que es capaz de vencer la mayor parte de las ocasiones, especialmente tras 1500 y 1750 partidas de entrenamiento. Sin embargo, si analizamos en detalle el comportamiento del algoritmo vemos varias cosas que no parecen estar funcionando bien. Durante el entrenamiento, en las últimas iteraciones la mayor parte de las partidas eran ganadas por el segundo jugador, cuando es el primero el que tiene la

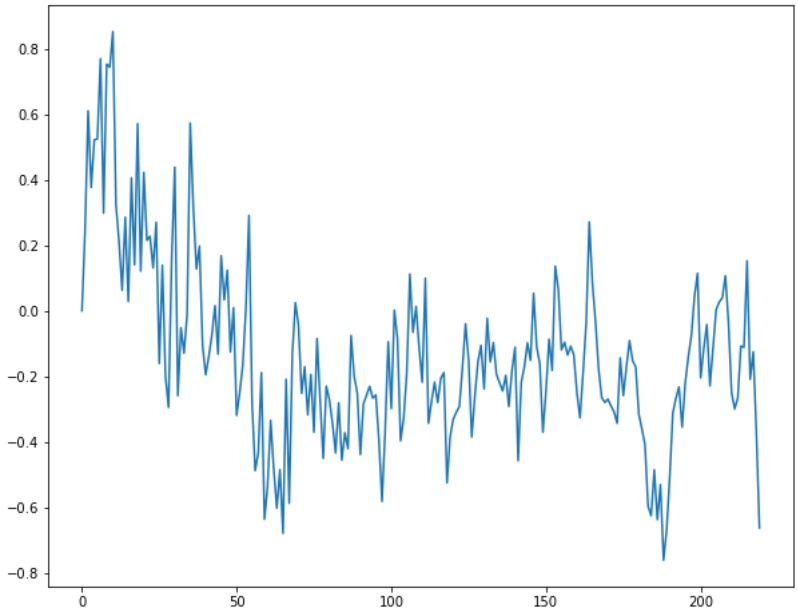


Figura 23: Evaluación del valor inicial por la red neuronal de AlphaZero durante las 2150 partidas de aprendizaje. Vemos que los valores son bastante inestables, variando entre $-0{,}8$ y $0{,}8$, con tendencia a otorgar al segundo jugador ventaja en la posición inicial. Se añaden 150 partidas extra de entrenamiento para observar la tendencia final.

ventaja teóricamente, y al no ser jugadores óptimos se podría esperar que en el peor de los casos el porcentaje de victoria de ambos jugadores (que están utilizando la misma red neuronal) sea similar. Esto se ve reflejado en la evaluación que realiza la red neuronal sobre el estado inicial: le otorga un valor de $-0{,}20$ tras las 2000 partidas de entrenamiento (tras 1500 y 1750 partidas los valores son de $-0{,}24$ y $-0{,}25$), favoreciendo claramente al segundo jugador. En la figura 23 podemos ver la evolución de este valor a lo largo del entrenamiento. Esto además no ha ocurrido solo en esta ocasión: en otros modelos que he entrenado desde cero también han terminado convergiendo hacia una política y evaluaciones no muy buenas (y que daban ventaja al segundo jugador).

Por otro lado, en partidas en las que claramente hay que realizar una jugada (porque o bien ganas inmediatamente o bien si no la realizas el oponente gana en el siguiente turno), en ocasiones AlphaZero no la realiza. Esto puede deberse a dos problemas, o bien a que la política da una probabilidad ínfima a dicha acción (y por lo tanto no se prueba en las simulaciones) o bien a que la función de valor no está reflejando la realidad del tablero. En la práctica, parece que están ocurriendo ambas. En la figura 24 podemos observar este comportamiento. En esta figura en el primer tablero le toca mover al primer jugador (rojas), que claramente tiene que colocar la ficha en la segunda columna por la izquierda para evitar la victoria inminente del segundo jugador. Sin embargo, la red neuronal otorga una probabilidad pequeña a dicha jugada (0,03). La evaluación del estado además le otorga valor $-0{,}76$, indicando que la situación es muy favorable al oponente.

Tras evaluar el MCTS la situación se decanta por colocar la ficha en la columna central (que tiene incluso probabilidad inferior a la jugada correcta), probablemente porque el estado al que llegamos tiene un valor incoherente con la situación ($-0{,}64$, cuando realmente el jugador tiene la victoria en sus manos). Además, otorga una probabilidad de 0,01 a la jugada correcta. Sin embargo,

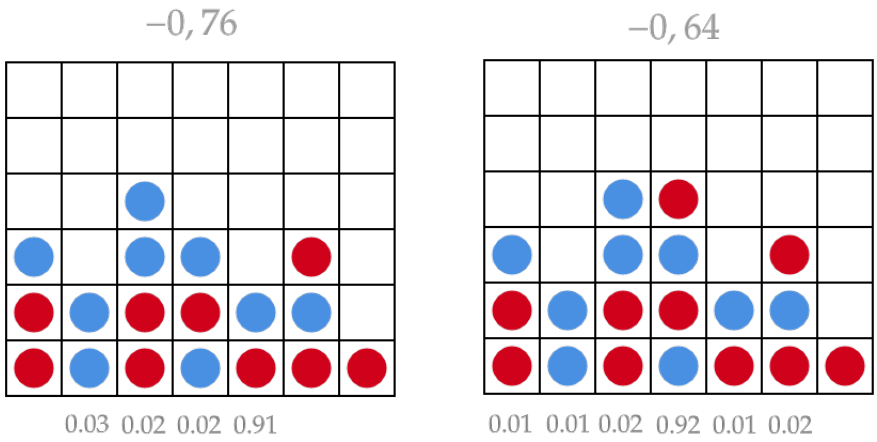


Figura 24: Ejemplo del comportamiento de AlphaZero para una cierta situación escogida de forma aleatoria, indicando la evaluación que hace la red neuronal del tablero (número sobre el tablero) y la política proporcionada por la red (debajo de cada columna se marca la probabilidad de aquellas acciones con probabilidad superior al 0,01).

el MCTS es capaz de sobreponerse a esta probabilidad tan baja (en parte gracias a la incorporación del ruido, que con alta probabilidad incrementa la probabilidad de probar dicha acción), y termina seleccionando la jugada correcta para el segundo jugador, ganando la partida.

Por último, vamos a fijarnos en las probabilidades otorgadas por la red neuronal a la primera jugada, al igual que hacíamos con el tres en raya. Si nos fijamos en los dos mejores modelos (1500 y 1750 partidas), otorgan a la posición inicial una probabilidad de 82,79 % y 81,02 % a la columna a la derecha de la central (que bajo juego óptimo conduce al primer jugador al empate, cuando la columna central conduce a la victoria). En la figura 25 se puede ver la evolución de la política estimada, y que tras las 1750 partidas colapsa y pasa a favorecer a la columna a la izquierda de la central, jugada también subóptima.

Por último, probamos el algoritmo contra humanos. En 10 partidas[26] (6 como primer jugador y 4 como segundo) ha ganado 4 (todas como primer jugador), ha empatado 1 (como segundo jugador) y ha perdido 5. Dos de las victorias se han debido a errores flagrantes de los rivales y en todas las partidas las evaluaciones del estado eran incorrectas e inconsistentes. Mi hipótesis es que hay 2 problemas que están provocando que esto suceda:

1. Nosotros entrenamos a nuestra red neuronal con tuplas de la forma (s_t, π_t, z_t). Estas tuplas son útiles cuando el jugador que la ha realizado ha ganado o ha empatado, es decir, $z_t = 1$ o $z_t = 0$. Sin embargo, cuando $z_t = -1$ estaremos enseñando a nuestra red neuronal una jugada que se ha comprobado que es mala (puesto que nos ha conducido a la derrota)[27]. De hecho,

[26] Se utiliza la versión con 1500 partidas de entrenamiento porque en las pruebas es la que ha obtenido mejores resultados y establece $\tau \approx 0$ todas las jugadas para utilizar siempre el mejor movimiento. Las partidas se han jugado en `https://papergames.io/es/conecta-4`.

[27] En realidad, no tiene por qué ser una jugada mala si el jugador estaba en una posición en la que iba a perder de cualquier manera, pero en la práctica muchas veces simplemente aprende una jugada que sí es mala.

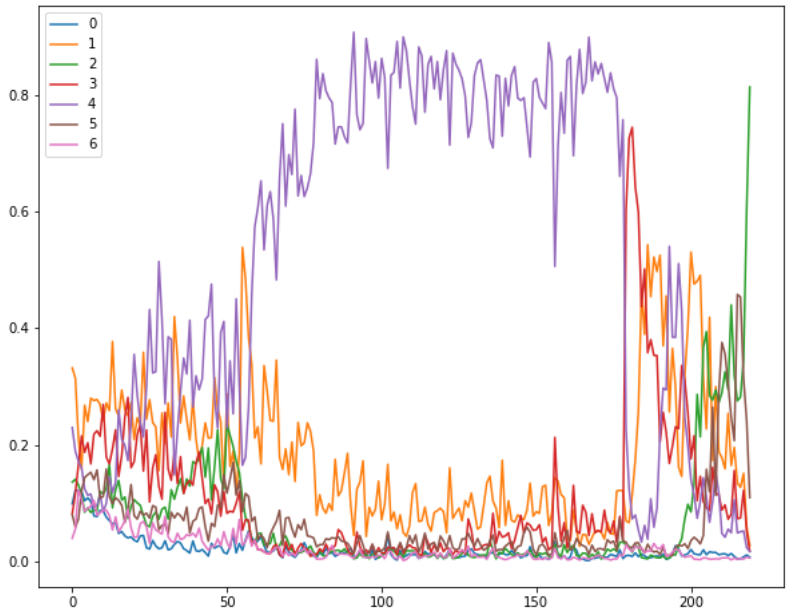


Figura 25: Política estimada por la red neuronal para el tablero inicial en cada iteración de AlphaZero. Vemos que la probabilidad de varias acciones cae rápidamente al 0 %. Esto es algo problemático, porque las acciones con probabilidad cercana a 0 no van a ser probadas si el ruido no las favorece. El algoritmo se decanta primero por la columna a la derecha de la central (lila) y al final del entrenamiento por la que está a su izquierda (verde), ambas jugadas subóptimas.

en AlphaGo [19] sí que tenían esto en cuenta y durante el entrenamiento de la red de política optimizaban siguiendo la fórmula

$$\Delta p \propto \frac{\delta \log p(a_t|s_t)}{\delta p} z_t$$

Lo importante de la misma es que al incluir el z_t el sentido a optimizar será contrario si hemos ganado $(+1)$ o perdido (-1). Mi idea es que en AlphaZero eliminan el tener el cuenta el resultado de la partida en la política porque la red neuronal compartida fuerza a crear una representación interna con las características más importantes del estado, y que al calcular la política a partir de esta representación intermedia en vez de directamente del estado puede contrarrestar este efecto (al poder calcular la política desde una serie de características intermedias del tablero de mayor calidad extraídas por la red neuronal). Además, si el jugador que pierde ha jugado bien, no es perjudicial aprender la mayor parte de las jugadas realizadas.

2. Por otro lado, hay una idea muy interesante que se explica en *Lessons from AlphaZero, part 4* [14]. Supongamos que el primer jugador ha jugado una partida impecable durante las 10 primeras jugadas y que el segundo jugador ha cometido varias imprecisiones. Sin embargo, en la jugada 11 el primer jugador efectúa una jugada horrible que le lleva a perder la partida en la siguiente jugada. Esto es algo que puede ocurrir perfectamente a lo largo del entrenamiento, y provocaría que todas las jugadas del primer jugador se etiquetasen como malas ($z_t = -1$ con $t = 0, 2, \dots$), cuando realmente no lo son; y todas las jugadas del segundo jugador como buenas ($z_t = 1$ con $t = 1, 3, \dots$), cuando la única buena es la última. De esta forma, para

todos los estados estaremos enseñando un valor incorrecto y para el segundo jugador una política incorrecta (puesto que, a pesar de que ha ganado, realmente las jugadas no eran óptimas).

En mi opinión (y también la de los autores de [14]), esto no afecta a la implementación de DeepMind porque ellos realizan una cantidad ingente de simulaciones durante el aprendizaje (gracias a más de 5000 TPU[28] y un código muy bien paralelizado). Estos dos problemas mencionados se solucionarían realizando una cantidad suficiente de simulaciones.

Por un lado, mi intuición es que el valor óptimo (la evaluación que haríamos del estado jugando siempre la mejor acción) se "propaga" desde los estados terminales hacia el resto del espacio de estados. En los estados terminales sabemos cuál es su valor exacto (-1, porque el jugador al que le toca jugar ha perdido). Si estamos justo en el estado anterior, si probamos todas las acciones veremos que habrá una que hace ganar al oponente, por lo que la política evitará esta acción y en el cálculo del valor tendrá solo en cuenta el resto de acciones (a no ser que con todas perdamos). De esta forma, en las últimas jugadas tendremos la política óptima y por lo tanto el valor óptimo. Tras suficiente entrenamiento, no solo los estados terminales tendrán el valor óptimo, sino también los estados anteriores. Repitiendo este proceso los valores precisos se propagan hasta el estado inicial y se ha completado el entrenamiento[29].

Por el otro, aunque una política sea horriblemente mala (y dé probabilidad 0 a la única acción buena), gracias a la introducción del ruido se puede "reconducir" hacia una buena política si la evaluación de los estados es correcta. Si de forma aleatoria el ruido favorece a la acción óptima y la evaluación de los estados corrobora que es buena, el MCTS al probar esa acción obtendrá valores altos, mientras que para el resto de las acciones obtendrá valores menores y terminará prefiriéndola con el tiempo.

La política y el valor se aprenden al mismo tiempo. Como ya introdujimos en la sección 2.1, la función V del aprendizaje por refuerzo depende de la política, y la función v que tratamos de aprender no es más que esta función V. Por otro lado, la política está dirigida por la función del valor: si la función del valor no refleja la situación actual, la política tampoco lo hará. Poniendo todo esto junto, la idea es que durante el entrenamiento se puede llegar a una política y evaluaciones malas (por la aleatoriedad del proceso), pero que realizando un número suficiente de simulaciones se pueden "arreglar", logrando obtener un valor y una política buenos y que permitan al MCTS calcular la mejor jugada.

En el caso del tres en raya no tuvimos estos problemas. Esto probablemente se deba a que la profundidad del tres en raya es muy inferior (a lo sumo 9 jugadas, y es muy probable que si la política es mala se termine antes), y la cantidad de partidas jugadas en relación a la complejidad del juego es mucho mayor que la empleada para el Conecta 4.

[28] TPU es la abreviación de Unidad de Procesamiento Tensorial, un hardware específico diseñado para acelerar la ejecución de código de inteligencia artificial. En [20] se especifica que se usaban 5000 TPU de primera generación para generar las partidas de entrenamiento y 16 TPU de segunda generación para entrenar las redes neuronales.

[29] Al ser una red neuronal, realmente no se aprende exactamente el valor del estado sino una aproximación.

AlphaZero Q

Estos problemas sin embargo son devastadores para implementaciones en las que no se realiza tanto entrenamiento, como es nuestro caso. Una solución que proponen en [14] para evitar la gran inestabilidad del entrenamiento es modificar el valor z_t utilizado para indicar si una jugada es buena o mala. En vez de utilizar el resultado del juego lo que se va a utilizar es el valor $Q(s_t, a_t)$, que hemos calculado en las aristas.

El objetivo de ambos valores es prever lo bueno que es el estado en base a estimaciones. Utilizar el resultado de la partida como hace el equipo de DeepMind, si la política que tenemos es buena es mucha mejor opción porque refleja cuál será realmente el resultado final de la partida. Pero si la política que tenemos es mala, es mejor ponderar las 200 evaluaciones de los nodos a los que hemos llegado para tratar de estabilizar su valor. De esta forma, reducimos el impacto del segundo problema mencionado anteriormente. Por ello, con que la evaluación de los estados sea moderadamente buena lograremos tener una predicción buena y sobre todo mucho más estable, estabilizando también las políticas propuestas por el MCTS para el entrenamiento[30]. A esta variante nos referiremos bajo el nombre de AlphaZero Q.

Vamos a estudiar el comportamiento de AlphaZero Q. Entrenaremos de nuevo con 2000 partidas, aunque esta vez divididas en 400 iteraciones de 5 partidas cada una. En cada iteración se utilizan *batches* de tamaño 16 y tan solo 3 épocas. La razón de esto es que al utilizar la función Q en vez del resultado final de la partida, la función v cambiará mucho más lentamente (puesto que inicialmente las predicciones del valor de cada estado serán 0, por lo que la magnitud de la Q con la que entrenamos es muy pequeña en comparación con utilizar el resultado de la partida que desde el inicio es 1, 0 o -1). Además, queremos utilizar siempre las últimas iteraciones porque en este caso sí que buscamos que se vayan acercando poco a poco las estimaciones al valor correcto, por lo que en cada iteración solo entrenamos con las partidas de dicha iteración y después las desechamos.

Analizamos de nuevo las estimaciones tanto del valor como de la política en el estado inicial, información recogida en las figuras 26 y 27 respectivamente. Vemos que efectivamente al inicio el valor es mucho más estable, aunque en iteraciones finales es igual de inestable que la versión estándar (nótese la diferencia de rango en los ejes en comparación con AlphaZero). El rango alcanzado por los valores está entre $-0{,}5$ y $0{,}2$, aunque de nuevo vemos preferencia por los valores negativos. Además, vemos que la inestabilidad aumenta a lo largo del entrenamiento, posiblemente como consecuencia de que los valores estimados van aumentando su magnitud a lo largo del entrenamiento (que es lo que pretendíamos con esta construcción), aunque no se ha logrado el efecto deseado.

En cuanto a la política, vemos que en este caso sí que aprende la política óptima, con bastantes buenos resultados. Además, la evolución de la política es la que nosotros deseamos. Para las acciones subóptimas, desciende la probabilidad de forma paulatina, desde la iteración inicial (en la que todas las acciones son equiprobables) hasta la última iteración. Además, su probabilidad tarda mucho más en llegar a valores cercanos a cero que con AlphaZero, permitiendo que se prueben más, aunque esto también se puede deber al menor número de épocas de entrenamiento. En ciertos puntos del entrenamiento algunas acciones aumentan su probabilidad en detrimento de la acción óptima, pero rápidamente la red neuronal se recupera.

[30] Como las simulaciones realizadas por el MCTS dependen en gran medida de los valores estimados por la red neuronal, al estabilizar los valores posiblemente mantendremos también las estimaciones de la política más estables.

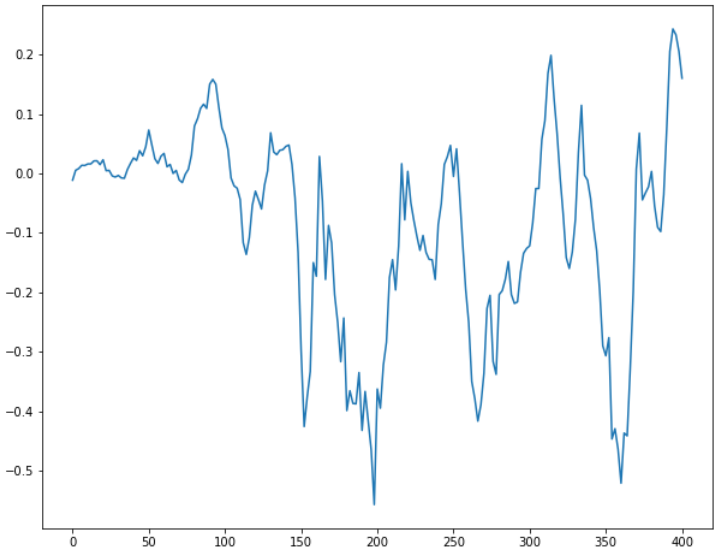


Figura 26: Valor estimado por la red neuronal para el tablero inicial (vacío) en cada iteración de AlphaZero Q.

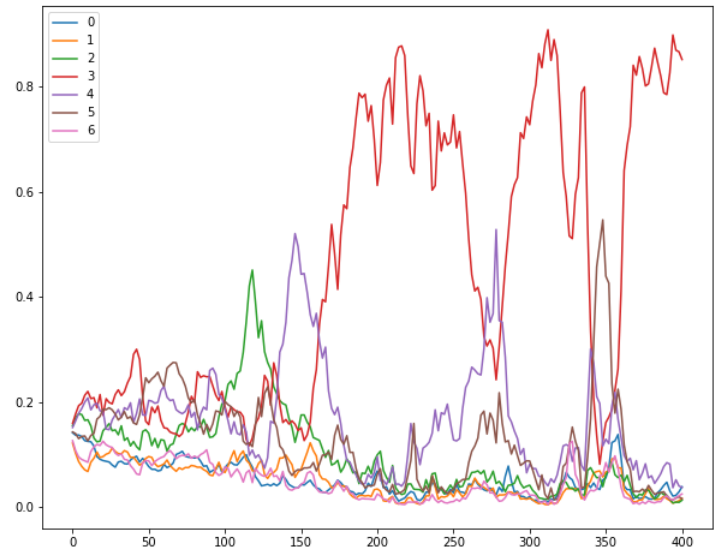
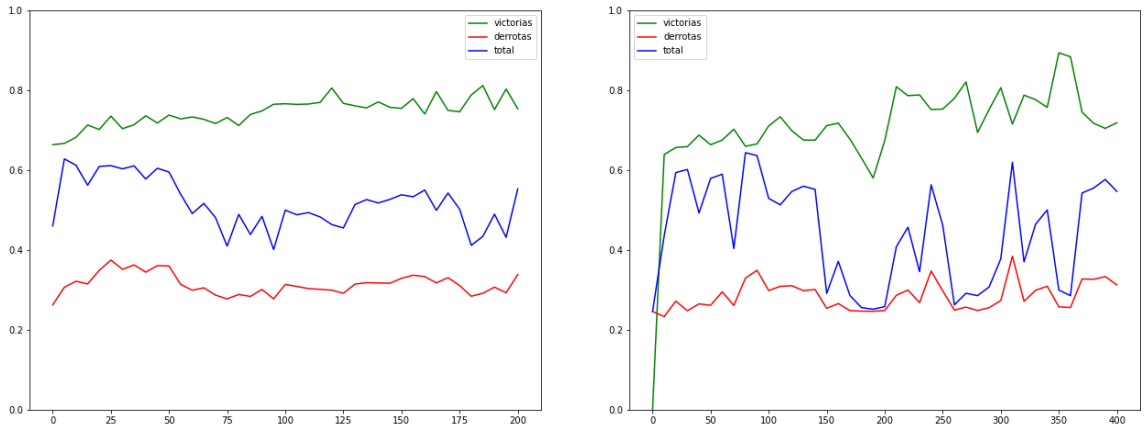


Figura 27: Política estimada por la red neuronal para el tablero inicial en cada iteración de AlphaZero Q. Vemos que en esta ocasión sí que encuentra la jugada óptima (colocar la ficha en la columna central).



(a) AlphaZero **(b)** AlphaZero Q

Figura 28: En verde, el porcentaje de veces que la red neuronal predice correctamente un estado ganador (le otorga una etiqueta en $(0, 1]$), en rojo el porcentaje de veces que predice correctamente un estado perdedor (le otorga una etiqueta en $[-1, 0)$) y en azul el porcentaje de muestras predicho correctamente, todo ello a lo largo de las iteraciones de entrenamiento. En AlphaZero se entrena con iteraciones de 10 partidas, mientras que en AlphaZero Q con iteraciones de 5 partidas: por ello ambas escalas son equivalentes ($200 \times 10 = 400 \times 5 = 2000$ partidas de entrenamiento).

Por último, para comprobar la calidad de las predicciones realizadas utilizaremos el *Connect-4 Data Set*[31] de [8]. Este conjunto de datos de entrenamiento contiene las 67557 posiciones legales del Conecta 4 en las que le toca jugar al primer jugador, ninguno de los jugadores ha ganado y el siguiente movimiento no está forzado. Para cada posición se incluye si se gana, empata o pierde suponiendo que ambos jugadores juegan de forma óptima. De estas posiciones se escogen 2000 de forma aleatoria y se calcula el valor estimado por la red neuronal. Aquellos valores mayores a 0 se interpretan como posiciones ganadoras y aquellos menores a 0 como posiciones perdedoras. Por simplicidad no se predicen los empates (aunque entre las muestras hay empates) porque lo lógico sería entonces que los empates se correspondiesen con las posiciones a las que la red neuronal da un valor de $(-u, u)$, siendo u un umbral que habría que determinar. Hay que recalcar que estas son las posiciones más difíciles de predecir, puesto que son aquellas en las que no hay ninguna amenaza y la partida está prácticamente comenzando. En la figura 28 se puede ver el resultado, tanto para AlphaZero como para AlphaZero Q.

Vemos que AlphaZero es más consistente con sus predicciones, manteniéndose siempre en el rango $[0{,}4, 0{,}6]$. Sin embargo, en la segunda mitad del entrenamiento (a partir de las 1000 partidas), no consigue superar en ningún momento una precisión superior al 0,55. AlphaZero Q por su parte es mucho más inconsistente, variando en el rango $[0{,}2, 0{,}6]$, incluso en las etapas finales de entrenamiento. Sin embargo, sí que logra llegar cerca del 0,6 en las iteraciones 310 (1550 partidas) y 390 (1950 partidas).

En este gráfico hay dos cosas fundamentales que comentar. La primera es que entre todas las

[31] Disponible en `http://archive.ics.uci.edu/ml/datasets/connect-4`.

posiciones proporcionadas por el conjunto de entrenamiento, en un 65,83 % de las posiciones gana el primer jugador, en un 24,62 % gana el segundo y 9,55 % se llega a un empate bajo juego óptimo, que es lo que estamos tratando de predecir. Por ello, hay un claro desequilibrio entre las clases (y lo esperable es que esta distribución se mantenga en las 2000 muestras escogidas aleatoriamente), con lo que para mejorar la función de pérdida es normal que la red neuronal se centre en acertar más las victorias que las derrotas.

El segundo punto fundamental es que hay grandes similitudes entre estos gráficos y las figuras 23 y 26. Esto secunda el punto anterior y nuestro interés en que en el tablero inicial el valor favorezca al primer jugador: en aquellas iteraciones en las que lo hace el número de aciertos en el valor **de todas las posiciones** aumenta.

Sin embargo, estos dos gráficos son bastante desalentadores, puesto que muestran que realmente no está habiendo aprendizaje para estas posiciones, y que necesitamos más entrenamiento o una red neuronal más grande (como es el caso del AlphaZero original) para capturar mejor la situación del tablero.

Por último, recalcar que estas son las posiciones más difíciles de predecir, puesto que no hay ninguna jugada forzada y se corresponden a los momentos iniciales de la partida, que es para los que se debería aprender más tarde la jugada óptima según nuestra hipótesis. Además, el conjunto de entrenamiento no incluye posiciones en las que juega el segundo jugador.

Por último, enfrentamos las distintas versiones de AlphaZero Q entrenado con la misma versión sin entrenar que utilizábamos de AlphaZero (los mismos pesos en la red neuronal), para facilitar la comparación. En la figura 29 se pueden ver los resultados.

En vista de lo discutido, dos de los factores que pueden estar provocando problemas son la utilización de una red neuronal excesivamente sencilla, que no sea capaz de capturar la complejidad del tablero, y la falta de entrenamiento.

Para estudiar el impacto de la red neuronal, se entrenará AlphaZero con una red neuronal más grande, con 14 bloques residuales en vez de 4, lo que hace que el número de parámetros de la red neuronal pase de unos 90000 a ligeramente por encima del millón. Se entrena con 2000 partidas, aunque se reduce el número de épocas a 4, el tamaño de los subconjuntos de entrenamiento a 16 y se seleccionan 1000 jugadas de entre las últimas simulaciones. Todos estos cambios tienen como objetivo estabilizar el entrenamiento y evitar el colapso que vimos con la otra red neuronal. Sin embargo, en la figura 30 podemos ver que tras 2000 partidas se produce un colapso similar al que se producía con la red más pequeña. La red sin entrenar contra la que se enfrenta es la misma que en ocasiones anteriores.

Para estudiar el impacto del entrenamiento, vamos a entrenar a seguir entrenando AlphaZero Q hasta las 5000 partidas, dado que no se ha apreciado dicho colapso en AlphaZero Q. Tras el entrenamiento, la versión con 5000 partidas consigue ganar solventemente a la versión de 2000 partidas con 20 victorias y 10 derrotas como primer jugador y 20 victorias y 9 derrotas como segundo.

En 20 partidas contra humanos, logra 10 victorias y 10 derrotas. Como primer jugador ha jugado 8, de las cuales ha ganado 3 y perdido 5. Como segundo jugador ha jugado 12, de las cuales ha ganado 7 y perdido 5. A pesar de que el el porcentaje de victorias es similar al de AlphaZero con 2000 partidas de entrenamiento, se aprecia una mejoría, logrando vencer a oponentes bastante

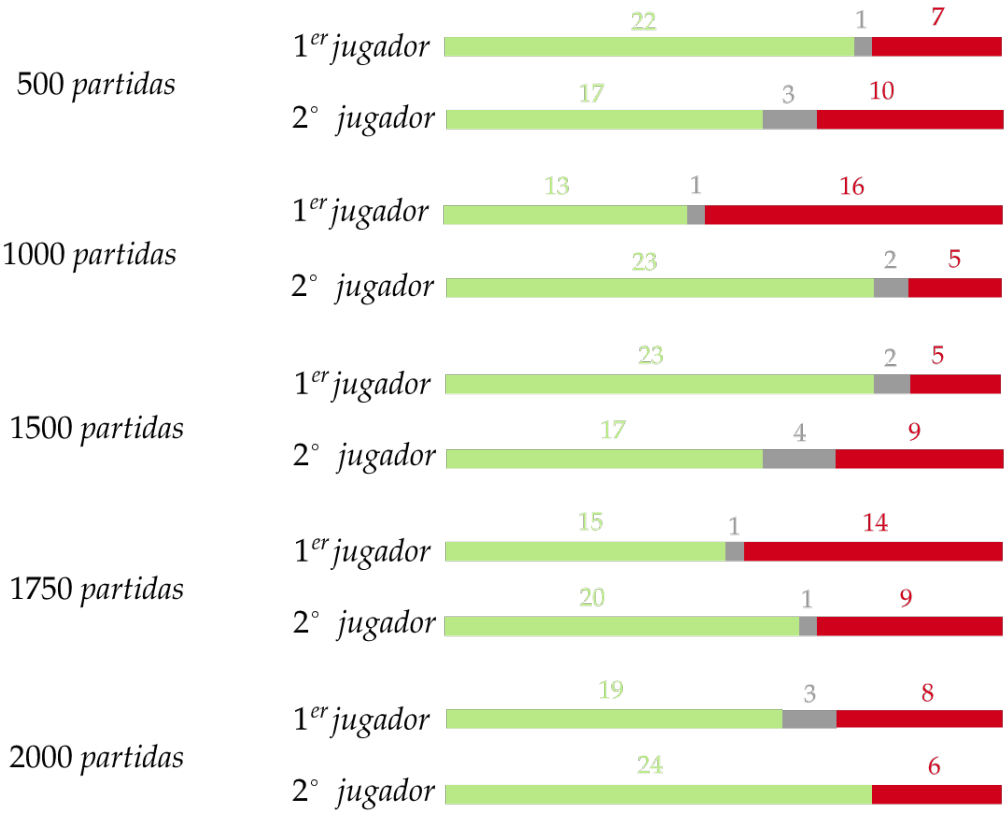


Figura 29: Resultados de las 30 partidas entre cada versión de AlphaZero Q y la versión sin entrenar. De nuevo vemos que la versión entrenada logra vencer a la versión sin entrenar, pero en varias ocasiones con poco margen.

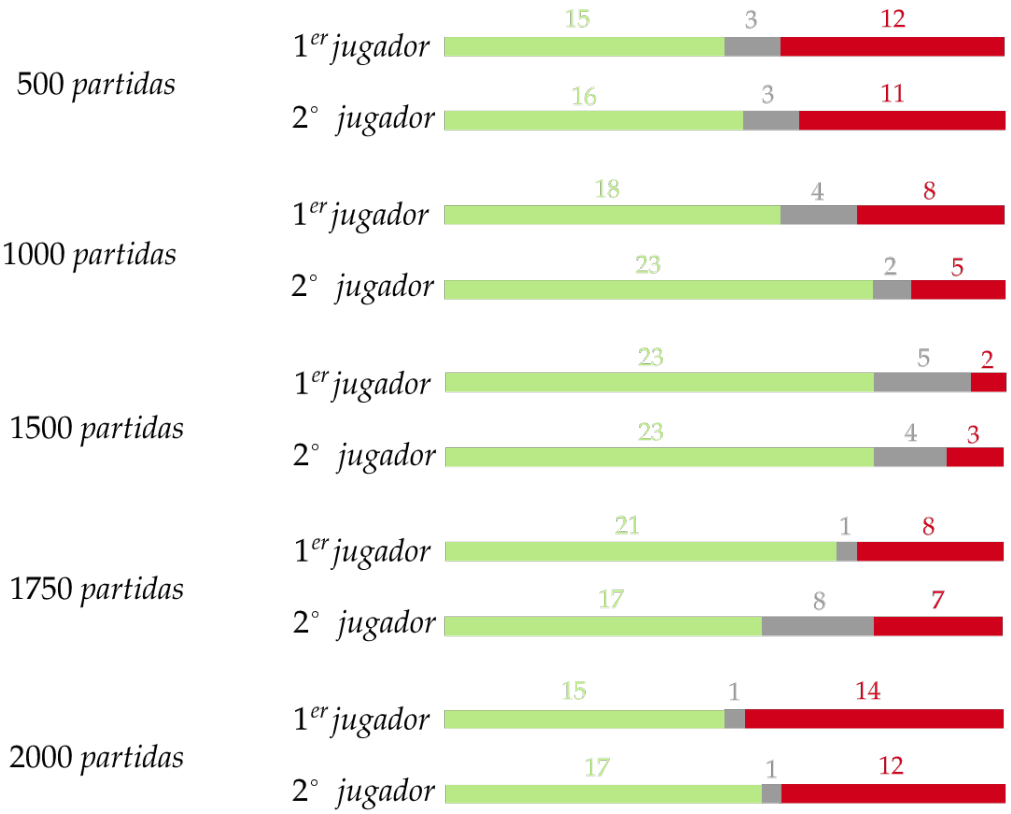


Figura 30: Resultados de las 30 partidas entre cada versión de AlphaZero con la red neuronal grande y la versión sin entrenar. Hasta las 1500 partidas vemos una buena progresión pero a partir de ahí decae enormemente su rendimiento.

buenos. En el apéndice C se estudian varias de estas partidas y el comportamiento de AlphaZero Q en ellas. Las predicciones de la política han mejorado bastante, aunque las predicciones del valor siguen siendo inconsistentes, y hay todavía bastante margen para el aprendizaje, especialmente al comienzo de la partida.

Conclusiones

Vemos que los resultados obtenidos no son tan buenos como los que se obtuvieron en el tres en raya. Todavía está lejos de jugar de forma óptima, pero se aprecia un aprendizaje y algunos destellos de calidad en las partidas jugadas contra humanos, como se describe en el apéndice C. Se ha explorado también la variante AlphaZero Q, con la que se logra evitar el colapso durante el entrenamiento, pero el reducido número de partidas de práctica siguen lastrando su rendimiento. Además, hay que tener en cuenta que esta arquitectura de la red neuronal (basada en convoluciones de tamaño 3x3) es especialmente favorecedora para el go y el tres en raya, pero no para el Conecta 4 (donde necesitaríamos una convolución de al menos 4x4 para capturar de forma sencilla la posibilidad de colocar cuatro fichas en raya), lo que puede perjudicarnos.

6. Conclusión

El algoritmo AlphaZero se apoya en muchas ideas teóricas preexistentes (redes neuronales, árboles de búsqueda de Monte Carlo, algoritmos de bandidos, etc) sobre las que construye de forma muy inteligente un sistema de aprendizaje por refuerzo. La genialidad de la familia de algoritmos de AlphaZero no está en crear algo nuevo desde cero, sino en juntar componentes que ya se sabía que funcionaban para construir algo mejor.

En el aprendizaje para el tres en raya se ha podido apreciar la enorme potencia y versatilidad del algoritmo, siendo capaz de aprender en tan solo 3 horas a jugar de forma óptima (y muchos de los conceptos básicos los aprendía mucho antes).

Para el Conecta 4 sin embargo se aprecian las dificultades que tiene el algoritmo para aprender en espacios de estados grandes ($1{,}6 \cdot 10^{13}$ estados en el Conecta 4 [2], frente al cuarto de millón para el tres en raya). A pesar de que el algoritmo aprende, sería necesario mucho más entrenamiento que el realizado para lograr jugar de forma óptima. A pesar de esto, hemos podido ver cómo se produce el aprendizaje, cuáles son los problemas que surgen en la práctica y distintas modificaciones para solventarlos.

Conclusion

AlphaZero algorithm is built on top of several preexisting ideas (neural networks, Monte Carlo Tree Search, bandit algorithms, etc) and brilliantly adding reinforcement learning. The geniality of AlphaZero and its variants is not building an entire algorithm from scratch, but putting together some pieces that had been tested previously to build something better.

In Tic Tac Toe's learning, we have seen the huge power and versatility of the algorithm, being able to play optimally in just 3 hours (and some basic concepts were learnt much earlier).

However, in Connect 4 we have spotted the problems the algorithm has to learn in big states spaces (there are $1{,}6 \cdot 10^{13}$ different states in Connect 4 [2] against a quarter of a million in Tic Tac Toe). Despite the fact the algorithm learns, it would need much more training to learn how to play optimally. We have seen how the learning happens, what problems arise during training and different solutions to solve them.

A. Red neuronal

A.1. Arquitectura

En este apéndice se desarrolla en detalle la arquitectura de la red neuronal utilizada por los autores y qué labor desempeña exactamente cada uno de sus elementos. Tras ello se especifica la arquitectura que implementamos nosotros (con menos capas para acelerar las evaluaciones de los estados). La arquitectura de la red neuronal se ha extraído de [22] y la explicación de los componentes de la red neuronal se basa en el libro *Neural Networks and Deep Learning: A textbook* [1]. Para la explicación de las redes neuronales residuales se utilizará el artículo con el que se popularizó la idea: *Deep Residual Learning for Image Recognition* [10].

A.1.1. Bloque convolucional

El primer bloque es el convolucional, que se compone de 3 operaciones: convolución, normalización y activación. En la figura 31 tenemos la representación gráfica del mismo.

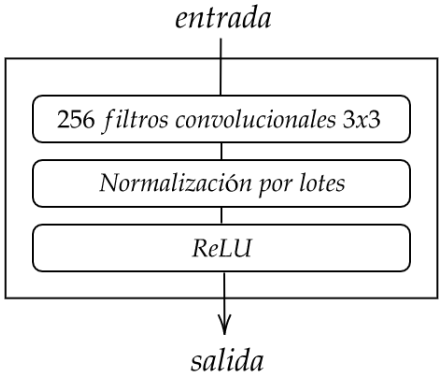


Figura 31: Estructura del bloque convolucional.

La operación principal es la convolución, que nos permite capturar las características espaciales de una imagen, representada en forma de matriz de valores. Para ello se operan los valores de pequeñas regiones de celdas de la matriz en busca de patrones "especiales", que indiquen la presencia de un elemento. Por ejemplo, si queremos reconocer una línea en nuestra imagen, lo que buscaremos será una serie de píxeles alineados de color oscuro que a ambos lados tengan píxeles más claros.

En primer lugar tenemos el tamaño del área sobre el que se aplica cada convolución, al que se llama tamaño del kernel. Un tamaño 1x1 significa que la convolución la aplicamos solo sobre la posición actual, mientras que 3x3 (que es el tamaño escogido para la red) significa que la aplicamos sobre el área 3x3 que tiene como centro la casilla actual.

Sobre el área dada por el tamaño del kernel (en este caso 3x3) vamos a aprender una matriz con pesos en cada una de las posiciones. La convolución consistirá en realizar el producto escalar entre la matriz con pesos (filtro o *filter* en inglés) y el área de la entrada seleccionada (multiplicar las mismas

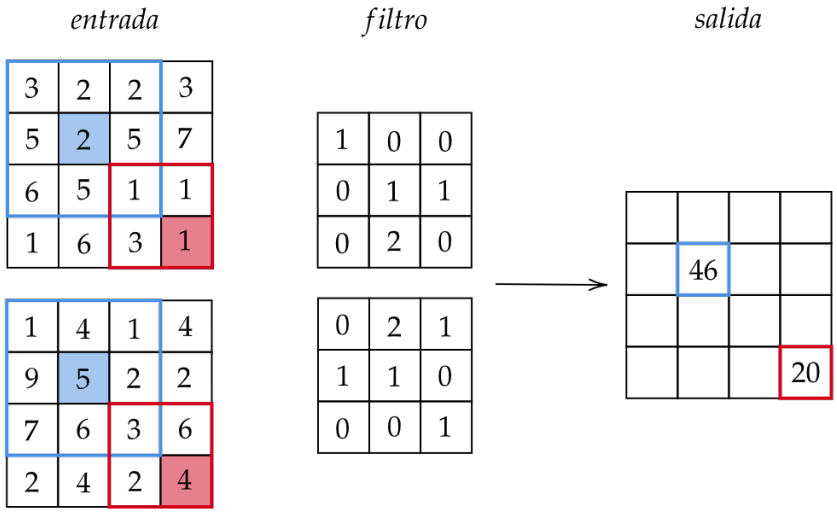


Figura 32: Ejemplo de convolución sobre una matriz arbitraria. La matriz tiene tamaño 4x4x2 y el área sobre la que aplicamos el filtro es 3x3 (3x3x2), obteniendo una salida de tamaño 4x4x1.

posiciones de las matrices y sumar los resultados). En este caso, como la matriz es tridimensional, el filtro también tendrá 3 dimensiones: las dos primeras dimensiones son las que le indicamos y la tercera coincide con la de la entrada. En la figura 32 se muestra un ejemplo de convolución de tamaño 3x3. En azul tenemos el área sobre la que aplicaríamos el filtro si usamos como centro el elemento en la segunda fila y la segunda columna. El resultado sería $3{\cdot}1+2{\cdot}1+5{\cdot}1+5{\cdot}2+4{\cdot}2+1{\cdot}1+9{\cdot}1+5{\cdot}1+3{\cdot}1 = 46$. El área roja representa el área obtenida tomando como centro el extremo inferior derecho: rellenando las posiciones exteriores al tablero con ceros el resultado es 20.

La idea es que cada filtro "aprenda" una característica y la busque a lo largo del tablero. Por ello son necesarios varios filtros, para que cada uno busque un patrón particular en el tablero. Concretamente, en la red neuronal original se utilizan 256 filtros distintos.

Hemos descrito la convolución para una única área (con una casilla central), pero se van a realizar varias convoluciones, con múltiples casillas como casilla central. Aquí aparecen 2 conceptos clave. El primero es el relleno (*padding* en inglés). No es posible aplicar la convolución directamente sobre los bordes, puesto que parte del filtro quedaría fuera de la entrada. Sin embargo, se pueden añadir ceros alrededor de la entrada para poder aplicar la convolución sobre esta área, que es la opción empleada.

El segundo es el desplazamiento (*stride* en inglés). Podemos no utilizar todos las casillas de la capa como casilla central para realizar una convolución y en vez de esto saltar casillas entre operación y operación. El desplazamiento es el número de casillas que nos saltamos. En caso de no saltarnos ninguno tenemos desplazamiento igual a 1, como es el caso. En la figura 33 se puede ver el funcionamiento de la primera capa convolucional de AlphaZero, con 256 filtros de tamaño 3x3.

En segundo lugar tenemos una capa de normalización por lotes. Estas capas actúan normalizando la entrada, permitiendo acelerar la convergencia de la red. Se idearon para mitigar un problema

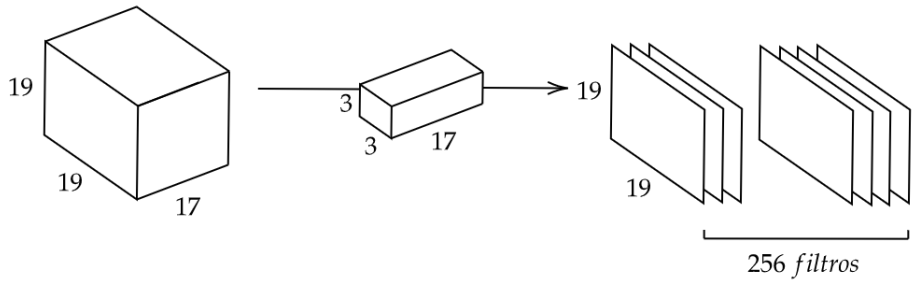
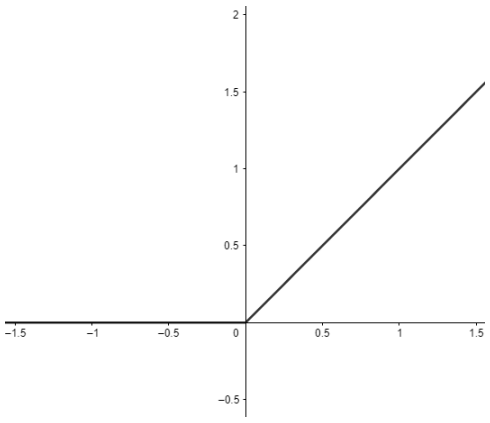


Figura 33: La entrada proporcionada a la red consiste en 17 matrices de tamaño 19x19. Sobre cada punto de las matrices 19x19 (rellenando con 0 en los bordes) se aplica una convolución con un filtro de tamaño 3x3 (y de profundidad 17), lo que da lugar a una salida de tamaño 19x19x1. Se aplican 256 filtros distintos para obtener una salida (de la primera capa) de tamaño 19x19x256.

conocido como *internal covariate shift*. A lo largo del entrenamiento las capas van a modificar los pesos de cada conexión de la red, para ajustarlos a los datos con los que estamos entrenando. Esto lleva a que, a pesar de introducir los mismos datos en dos iteraciones (épocas) de entrenamiento distintas, estos van a ser distintos en las capas intermedias (puesto que al llegar a estas capas ya habrán sido procesados por las capas anteriores, que han modificado sus pesos). En particular habrá cambiado la distribución de los datos y esto es lo que provoca que la convergencia sea más lenta. Para solucionar esto se van a normalizar cada uno de los lotes de datos de entrenamiento de forma que todos tengan una cierta media α_i y desviación típica β_i. Estos valores α_i y β_i no se toman fijos, sino que se aprenden para que la normalización no haga perder expresividad a los datos.

Por último, como función de activación se emplea la función ReLU (Rectified Linear Unit), que se define como:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & si\ x < 0 \\ x & si\ x \geq 0 \end{cases}$$



Esta función se aplica a la salida de la normalización por lotes para introducir no linearidad en el proceso, permitiendo crear funciones más complejas. Además, es una función poco costosa, por lo

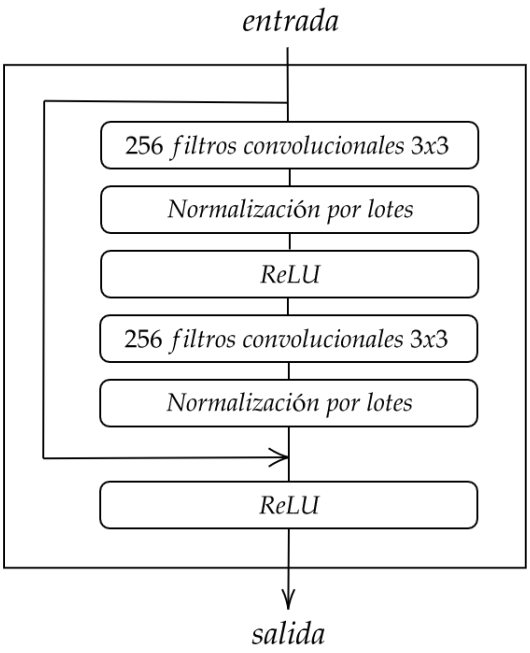


Figura 34: Estructura del bloque residual.

que acelera este proceso y se ha comprobado empíricamente que funciona muy bien, convirtiéndose en uno de los estándares *de facto*.

A.1.2. Bloque residual

A continuación, la red contiene varios bloques residuales. En la figura 34 se puede apreciar en detalle su composición. Cada bloque residual contiene 6 capas, que se corresponden con 2 bloques convolucionales consecutivos. Sin embargo, se añade un "atajo" desde la entrada hasta la 6ª capa que permite a la entrada sortear las 5 primeras capas del bloque y solo pasar por la segunda capa de activación ReLU.

Aunque estos "cortocircuitos" habían sido propuestos en trabajos anteriores, no fue hasta la aparición de ResNet en 2015 cuando se demostró empíricamente que la idea funcionaba [10]. Tras su victoria en el ILSVRC2015, una competición de reconocimiento de objetos en imágenes, las redes neuronales residuales se popularizaron y han sido ampliamente utilizadas desde entonces. En particular, mientras que en las primeras versiones de AlphaGo se utilizaban redes neuronales convolucionales (sin cortocircuitos) en AlphaGo Zero se introducen las redes residuales, logrando una notable mejoría.

En las redes convolucionales se había visto la importancia de la profundidad de la red en el resultado final y que con profundidades mayores se podían obtener mejores resultados. La idea es que a medida que alcanzamos una mayor profundidad, las capas serán capaces de capturar cada vez más abstracción. Si nosotros estamos tratando de detectar una cara, la primera capa será capaz de detectar la comisura de los labios, la pupila o el borde de las cejas. Unas capas más adelante,

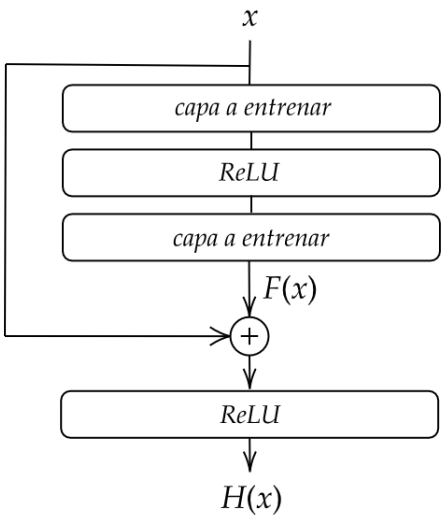


Figura 35: Esquema del cortocircuito en una red residual. Figura basada en la figura 2 de [10].

la red será capaz de detectar los ojos, la boca o la nariz, y en las siguientes capas será capaz de reconocer la cara al completo.

Sin embargo, esto en la práctica no sucedía en todos los casos. En [10] comparan dos redes neuronales de 20 y 56 capas en el conjunto de imágenes CIFAR-10, un conocido conjunto de entrenamiento[32]. Lo sorprendente es que la red de 56 capas obtiene peores resultados que la de 20. Esto es algo contradictorio, puesto que la red con 56 capas es la red de 20 capas a la que hemos introducido algunas capas extra en el centro. La red de 56 capas podría obtener exactamente el mismo error copiando los parámetros de las 20 capas de la red sencilla y aplicando la función identidad en el resto de capas intermedias.

La hipótesis de los autores es que esto se debe a que la red más profunda es más compleja de optimizar, y su solución es modificar la expresión que se optimiza con la esperanza de que la nueva expresión resulte más simple para la red neuronal.

En la figura 35 se puede ver la disposición del cortocircuito. La entrada del bloque son los datos x, que pueden seguir dos caminos. El primero son las dos capas con la función de activación ReLU en medio, y que tras aplicar los parámetros producirán la salida $F(x)$. Por otro lado, el cortocircuito no modifica la entrada, comportándose como la función identidad. Obviando la segunda función ReLU, la salida de este bloque es $H(x) = F(x) + x$. La idea de los autores es que optimizar esta función residual $F(x)$ será más sencillo que optimizar directamente la función $H(x)$. En el peor de los casos, si lo que hubiese que implementar en estas capas es la función identidad (para que al menos así el resultado sea igual de bueno que en la red menos profunda), entonces posiblemente sería más sencillo optimizar $F(x)$ a 0 que $H(x)$ a la identidad, teniendo en cuenta que tenemos capas no lineales en medio.

Como se describe en [1], esta idea en la práctica aporta dos ventajas principales. La primera

[32]El dataset se puede encontrar en `https://www.cs.toronto.edu/~kriz/cifar.html`. Contiene 60000 imágenes de 10 categorías distintas.

es que, anteriormente, era imposible entrenar redes con una gran profundidad debido a la que la convergencia de los pesos de la red era muy lenta. Con estos caminos sí que se logra el objetivo de los autores, acelerarla. Por otro lado, ResNet se concibió para el reconocimiento de imágenes. Los objetos pueden tener naturalezas muy variadas, siendo algunos de ellos muy sencillos de detectar y otros muy complejos. Por lo tanto, algunos de ellos necesitarán muy pocas capas para ser detectados y otros necesitarán muchas. En una red neuronal sin cortocircuitos, todos los conceptos deberán ser encontrados en una cantidad fija de capas. Si el concepto es muy complejo y la red poco profunda, no podrá ser localizado, mientras que si el concepto es simple y la red muy profunda la convergencia será muy lenta. Sin embargo, con los cortocircuitos "se deja" que la red elija cuantas capas necesita para representarlo, evitando estos problemas.

En el artículo de AlphaGo Zero se estudia la influencia de dos factores en el rendimiento de la red neuronal: si la red es convolucional (sin cortocirtuitos) o residual y si se utilizan 2 redes diferentes, una para el valor y otra para la política; o una única red combinada. Para ello calculan el elo que tendrían las redes neuronales como un jugador independiente (es decir, sin utilizar el MCTS). Utilizar dos redes convolucionales separadas obtiene un elo ligeramente superior a 3000, mientras que utilizar o bien una red convolucional común o dos redes residuales separadas obtiene alrededor de 3750 de elo. Por último, utilizar una única red residual logra unos 4300 de elo. Vemos por lo tanto que utilizar redes residuales supone un aumento importante del rendimiento frente a las redes convolucionales. Además, como ya anticipábamos, combinar las redes también produce una mejora sustancial en el rendimiento. Los autores la achacan principalmente a que al obligar a la parte común de la red a que genere una representación intermedia del tablero que sea adecuada para calcular tanto el valor como la política provoca que las características del tablero en las que se centra la red neuronal sean más generales y adecuadas a casos más variados[33].

A.1.3. Bloque de la política

Este bloque se va a encargar de tomar como entrada la representación intermedia del estado que ha calculado la red neuronal en todas sus capas compartidas y va a proponer una política (distribución de probabilidad sobre los movimientos posibles) indicando qué movimientos son los más prometedores para cada estado. En la figura 36 se puede ver más en detalle.

Las 3 primeras capas son convolución, normalización por lotes y ReLU, al igual que en el bloque convolucional. Sin embargo, la capa convolucional cambia la configuración, y tan solo tenemos 2 filtros, con tamaño de convolución 1. Esto nos permite reducir de 256 filtros en la capa anterior a 2 en la actual (es decir, de una matriz de tamaño 19x19x256 a una de 19x19x2).

La última capa es una capa densa. Las neuronas de una capa densa están conectadas a todas las neuronas de la capa anterior. La salida de este bloque será un vector de tamaño $19 \cdot 19 + 1$, en el que cada posición indica la probabilidad de colocar la pieza en una de las $19 \cdot 19$ posiciones del tablero, más una posición extra para indicar la probabilidad de pasar (un movimiento válido en el

[33]Por ejemplo, supongamos que estamos estudiando un juego en el que el primer jugador tiene mayores posibilidades de ganar. Si tenemos dos redes separadas, la red neuronal del valor podría simplemente centrar toda su atención en calcular a qué jugador le toca jugar y predecir el valor del estado en consecuencia. Sin embargo, si la red combinada hiciera eso no tendría información para calcular la política, por lo que la pérdida total de la red sería muy alta. Esta red combinada necesita capturar información muy variada del tablero y la idea es que a la larga la red del valor se beneficiará de esta información más general.

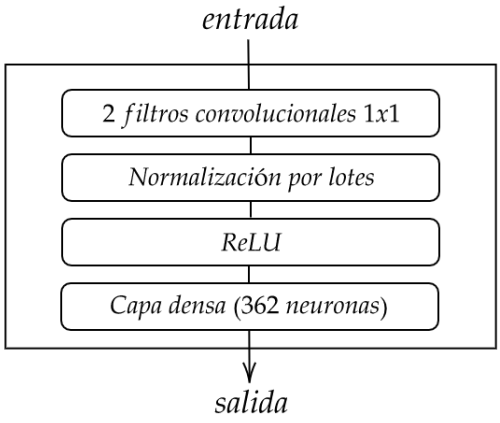


Figura 36: Estructura del bloque de la política.

go es no hacer nada y pasar el turno). En realidad, no se proporcionan probabilidades, dado que su suma no es 1 y los valores pueden ser incluso negativos. Para obtener las probabilidades a partir de la salida se debe aplicar la función *softmax*

$$\pi(s)(a_t) = \frac{e^{u_t}}{\sum_{k=1}^{362} e^{u_k}}$$

siendo u_t el valor calculado por la neurona t de la última capa densa de la política.

Para medir la similitud entre la política propuesta por la red neuronal $\mathbf{p}$ y la política π que nosotros le proporcionamos durante el entrenamiento se emplea la entropía cruzada, función que mide la diferencia entre dos distribuciones discretas de probabilidad, y se define como

$$\textit{entropía cruzada}(p,\, \pi) = -\sum_{x \in X} p(x) \log \pi(x)$$

En primer lugar nos vamos a fijar en la distribución de probabilidad p que nosotros consideramos correcta. Utilizando esta fórmula, ponderaremos la contribución de cada elemento x en función de su probabilidad $p(x)$, de forma que cuanto mayor sea la probabilidad $p(x)$, más contribuirá al total. Por otro lado, como estamos trabajando con distribuciones de probabilidad se cumple que $0 \leq p(x), \pi(x) \leq 1$. Por ello, el valor del logaritmo estará en el rango $(-\infty, 0]$, y nosotros lo que queremos es que cuanto mayor sea la diferencia entre las probabilidades $p(x)$ y $\pi(x)$, mayor sea la entropía cruzada. Esto lo logramos cambiando el signo del logaritmo. De esta forma, si un elemento tiene una probabilidad alta en p pero al mismo tiempo tiene una probabilidad alta en π, el valor $\pi(x)$ será cercano a 0 y el producto de ambos será pequeño. Si por el contrario su probabilidad en π es baja, el término $p(x) \log \pi(x)$ será grande y la entropía cruzada crece. Por último, si la probabilidad de un elemento es baja en p, independientemente de su probabilidad en π este término contribuirá poco al total.

En cualquier caso, esta es una idea general que justifica por qué esta función de pérdida se

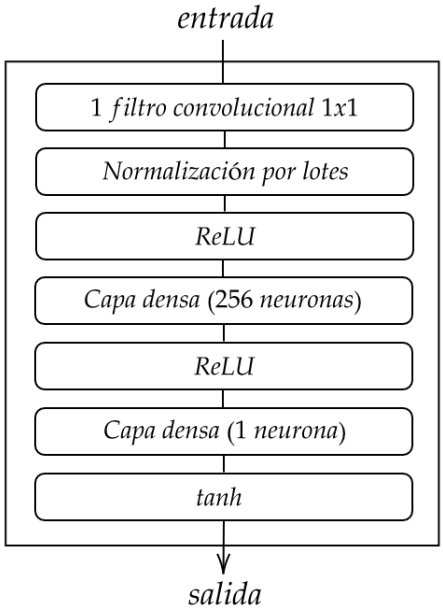


Figura 37: Estructura del bloque valor.

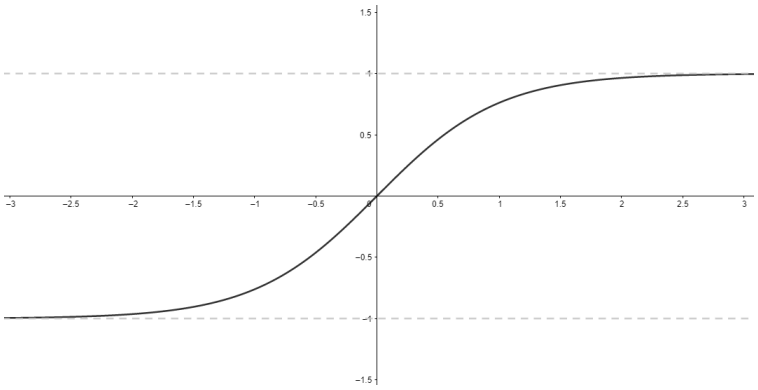
comporta como esperamos. La idea tras ella es más profunda y tiene origen en la teoría de la información [6].

A.1.4. Bloque valor

El bloque valor por su parte calcula el valor de una posición a partir de la representación intermedia, al igual que hacía el bloque de la política con la política. En la figura 37 se aprecia su estructura.

Todas las capas que lo componen ya han sido explicadas previamente, a excepción de la tanh, que calcula la tangente hiperbólica. Al igual que la ReLU, se trata de una función de activación que en este caso nos devuelve un valor en el rango $[-1, 1]$. Concretamente, corresponde a la fórmula:

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



La ventaja que tiene esta función es que el valor que le asignamos a un estado es $+1$ en caso de ganar el jugador actual, -1 en caso de ganar el jugador contrario y 0 en caso de empate. Valores intermedios permiten indicar qué jugador es más probable que gane, y la tangente hiperbólica lleva de forma natural los resultados de la red al rango $[-1, 1]$ deseado.

Como función de pérdida para este bloque se utilizará el error cuadrático medio, definido como

$$error\ cuadrático\ medio = \frac{1}{n} \sum_{i=0,\dots,n} (x_i - y_i)^2$$

siendo y_i el valor predicho para el dato i-ésimo y x_i la etiqueta para ese dato. El objetivo de esta definición es tratar de que las predicciones para el valor sean lo más acertadas posibles y para ello se penalizan más los errores mayores.

A.2. Implementación

En esta sección se incluye la arquitectura exacta de la red utilizada para el Conecta 4, aunque es fácilmente extrapolable al tres en raya. Como ya se indicaba en la sección 4.2, la arquitectura es muy similar a la original de AlphaGo Zero y simplemente se reduce el número de parámetros en la red y se adapta al juego con el que vamos a trabajar. En la figura 38 aparece la implementación del bloque convolucional, en la figura 39 la implementación de un solo bloque residual y en la 40 la implementación de tanto el bloque de la política como el del valor.

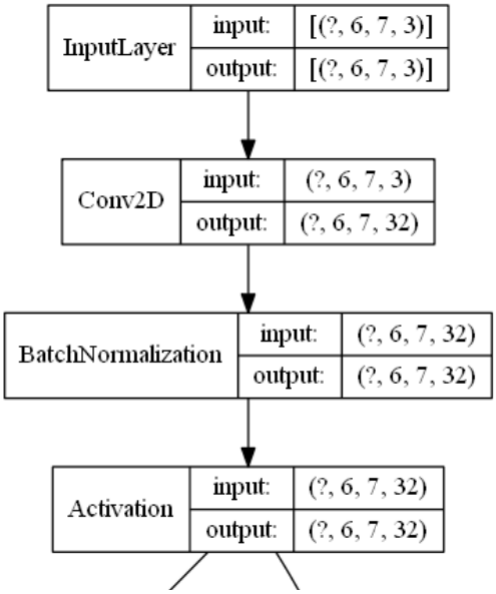


Figura 38: Bloque convolucional utilizado en nuestra red neuronal. Para cada capa se indica el tipo de capa y una tupla con las dimensiones de entrada y salida de los datos. La primera dimensión es un ? porque nuestra red neuronal puede recibir como entrada un número n de tableros (es decir, más de uno a la vez), y en consecuencia producirá n salidas diferentes, una por tablero.

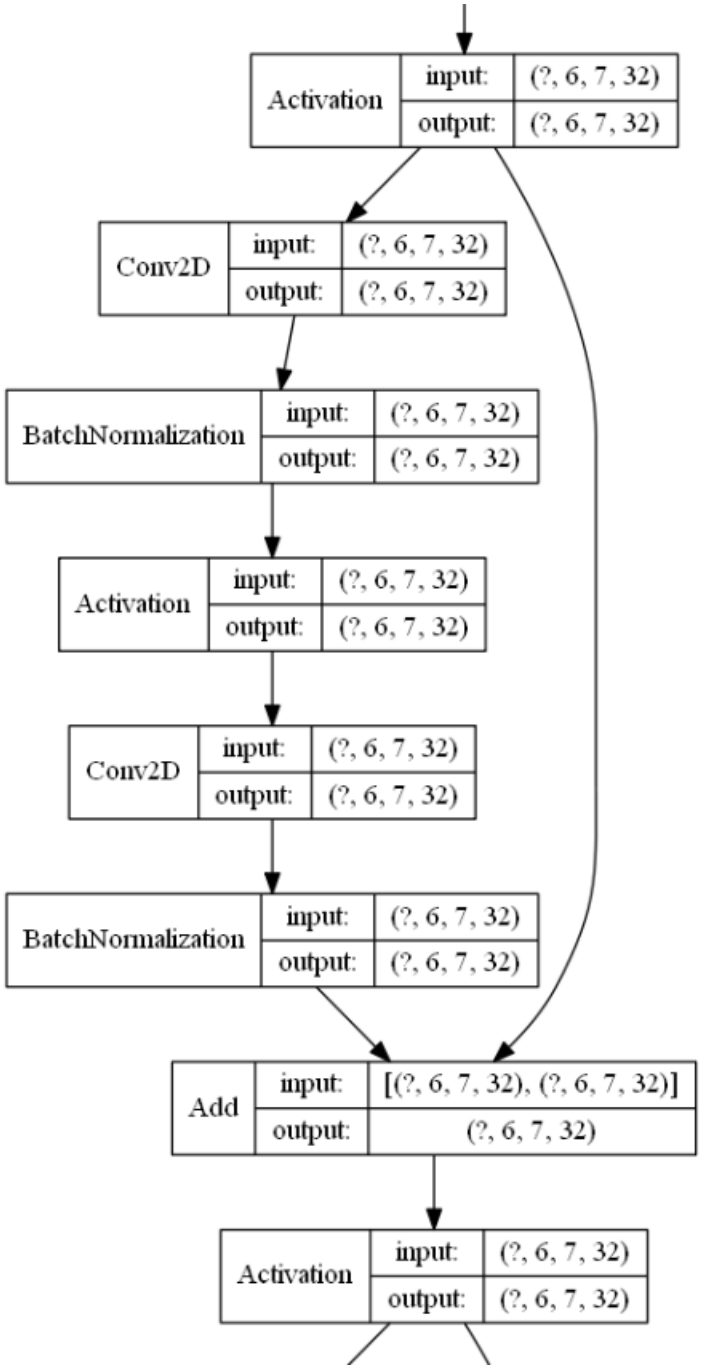


Figura 39: Bloque residual utilizado en nuestra red neuronal. La primera capa pertenece al bloque anterior, pero se mantiene para apreciar el cortocircuito. El bloque residual se compone de dos bloques convolucionales incluyendo un cortocircuito. En el segundo la capa Add se encarga de crear el cortocircuito. Aquí aparece representado un único bloque, la red neuronal tiene cuatro como estos colocados consecutivamente.

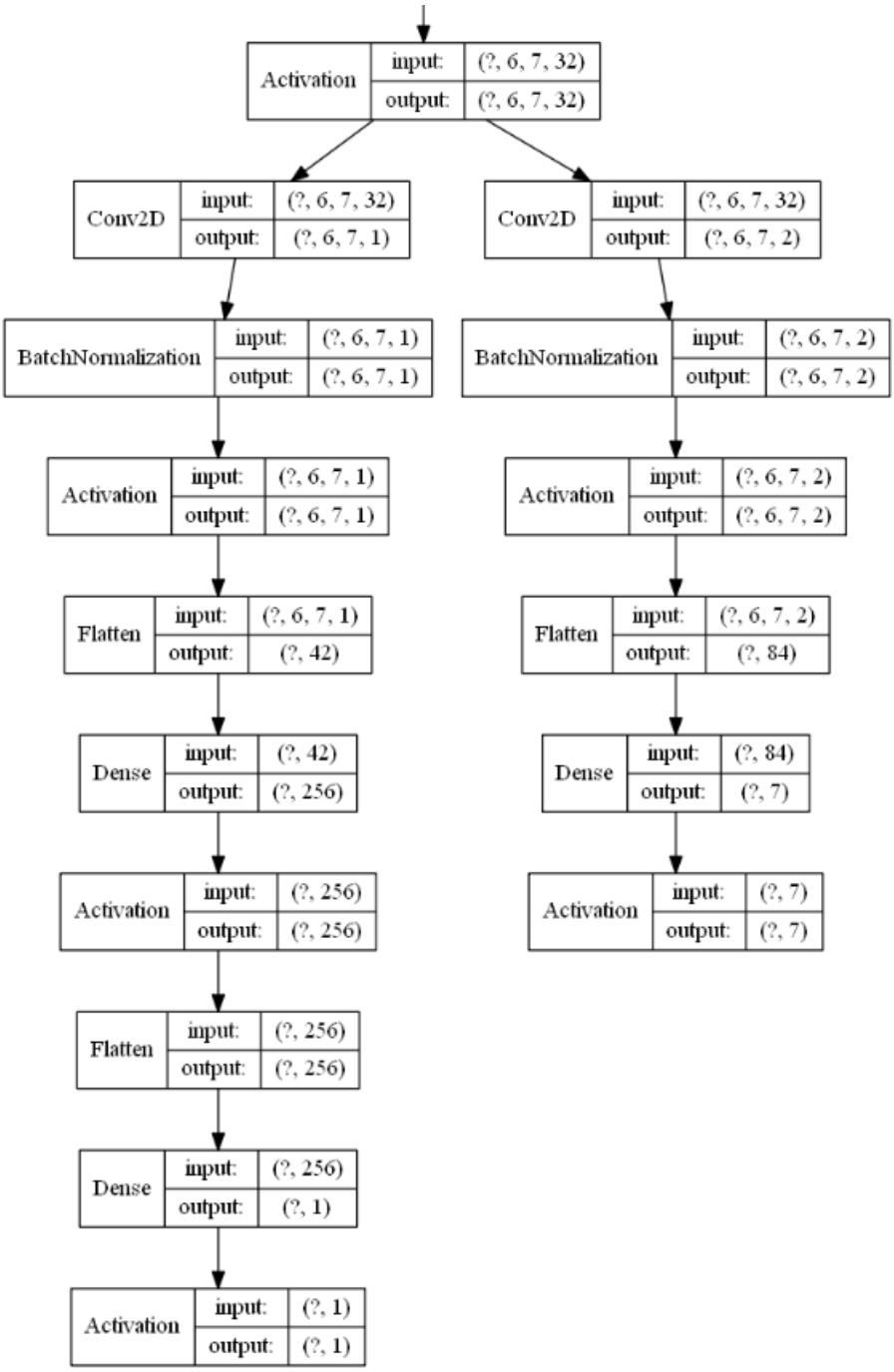


Figura 40: Bloques del valor y de la política utilizados en nuestra red neuronal. La primera capa corresponde al último bloque residual, pero se mantiene aquí para ver la estructura de la red. La parte izquierda corresponde al bloque del valor y la derecha al bloque de la política. Al tratarse de la red para el Conecta 4 el bloque de la política tiene como salida 7 números: la probabilidad de cada acción.

B. Tres en raya

En este apéndice se incluye información adicional sobre el tres en raya y el comportamiento de nuestra implementación.

En la figura 41 se detallan los movimientos incorrectos en la apertura para el segundo jugador. Para cada jugada subóptima (marcada en rojo) del segundo jugador (2ª pieza colocada en el tablero) se incluye una línea de juego que fuerza la victoria para el primer jugador. En el segundo movimiento del primer jugador (3ª pieza colocada) se puede forzar el movimiento del segundo jugador (4º movimiento) y en el 5º movimiento crear una doble amenaza marcada con línea discontinua roja (es decir, dos casillas distintas en las que el primer jugador puede lograr el tres en raya). De esta forma, independientemente de lo que juegue el segundo jugador (6º movimiento), el jugador inicial gana en el 7º movimiento. La línea de juego que lleva a la victoria puede no ser única.

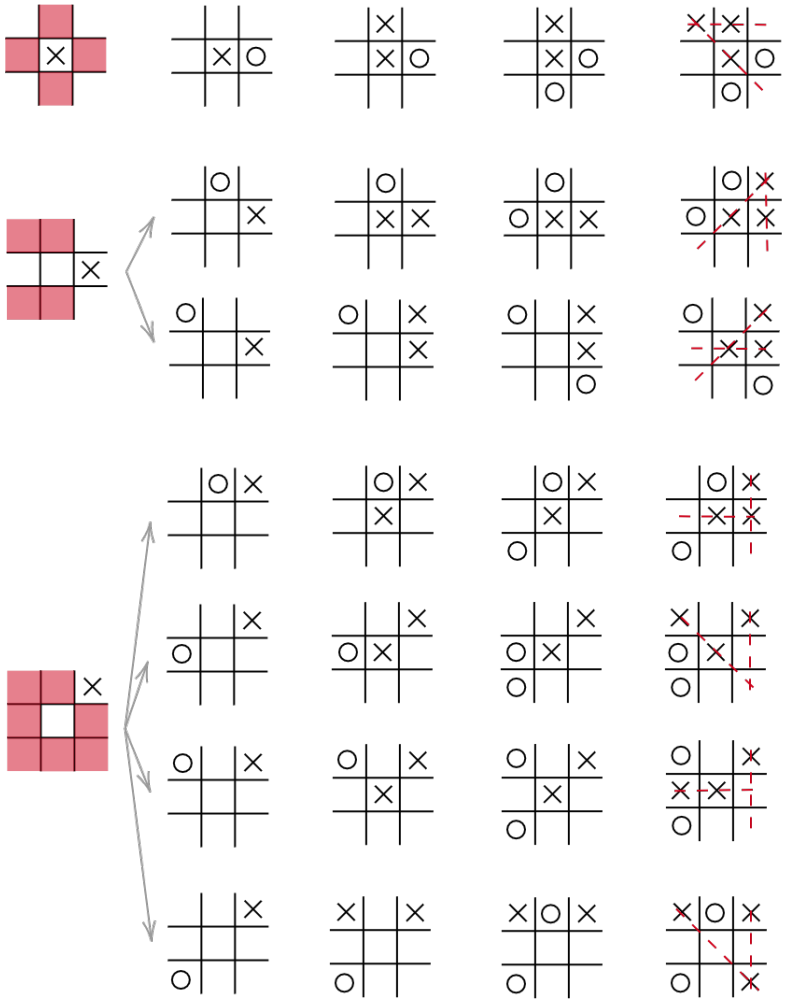


Figura 41: Líneas de juego en las que el primer jugador fuerza la victoria.

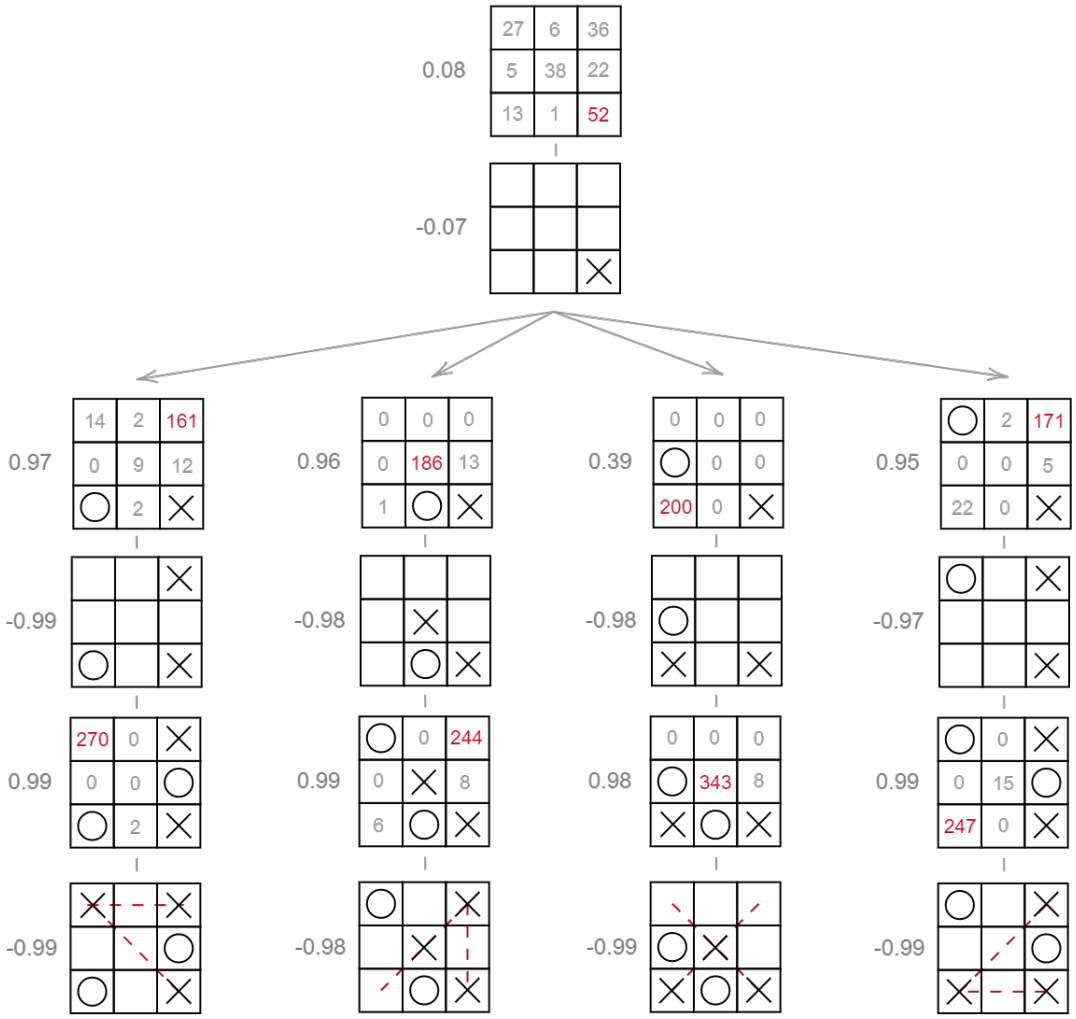


Figura 42: Árbol de exploración de nuestro algoritmo para las partidas en las que AlphaZero juega primero y el segundo jugador realiza un movimiento incorrecto (y por ende tiene que ser capaz de forzar la victoria). Junto a cada estado se muestra su valor (cercano a 1 si el estado es favorable a la persona a la que le toca jugar y a -1 si es desfavorable). Cuando es el turno de AlphaZero se indica el número de veces que se simula cada acción, y como $\tau \approx 0$ se escoge siempre la que se ha simulado más veces.

En la figura 42 se ve el comportamiento cuando nuestro algoritmo juega primero y el segundo jugador efectúa una jugada incorrecta, viendo que efectivamente nuestro AlphaZero es capaz de aplicar lo visto en la figura 41. A pesar de que hagamos 200 simulaciones por movimiento, para ciertos estados se simulan más veces las acciones porque el MCTS es el mismo durante toda la partida y si hemos predicho bien el movimiento del rival habremos anticipado y simulado en turnos anteriores las jugadas actuales. Vemos que tanto las acciones más simuladas por el MCTS como los valores de las posiciones son coherentes con la situación y que AlphaZero identifica rápidamente que el rival ha cometido una imprecisión y la aprovecha para forzar la victoria.

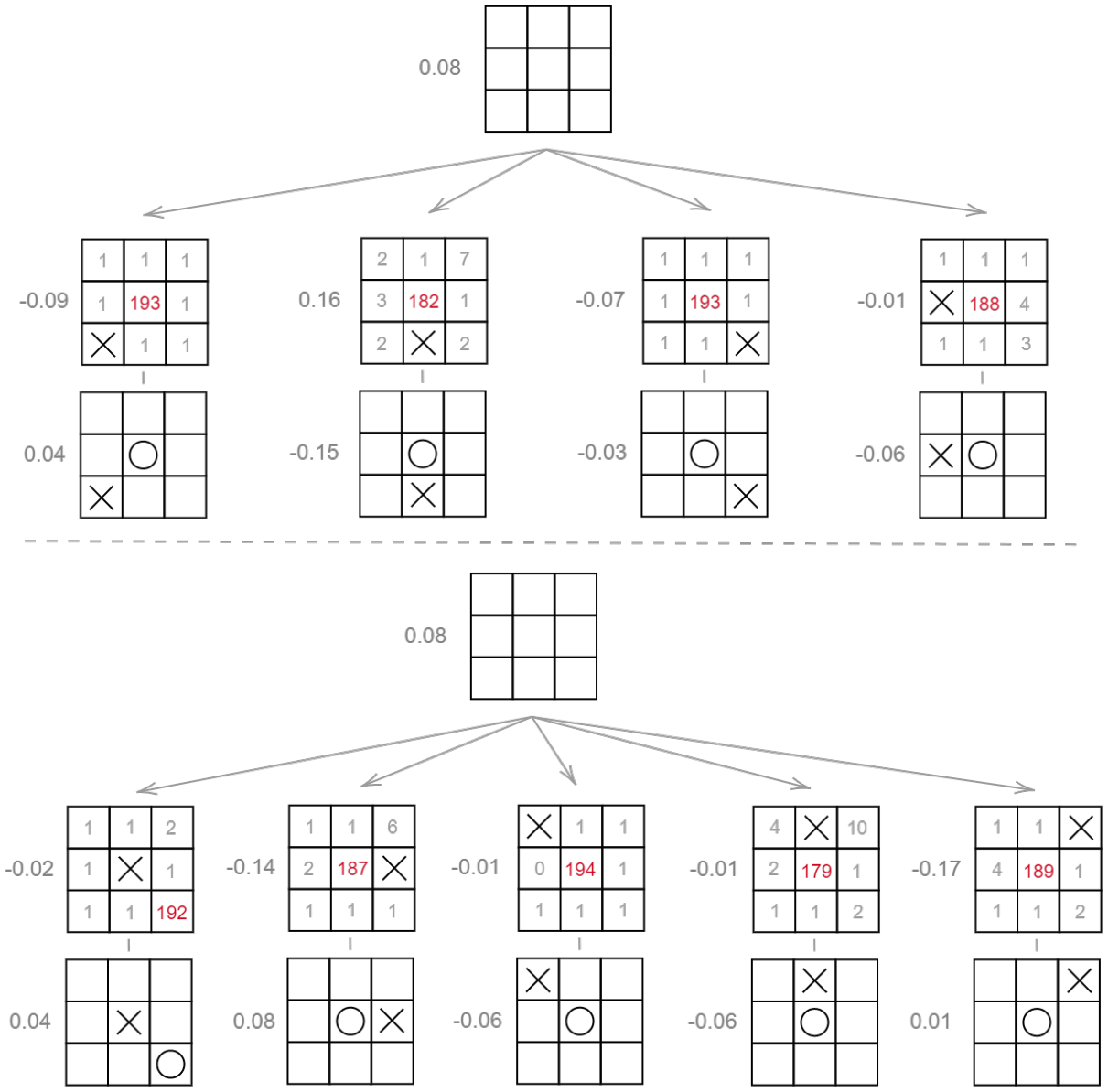


Figura 43: Árbol de exploración de nuestro algoritmo para las partidas en las que AlphaZero juega segundo. Junto a cada estado se indica el valor que asigna AlphaZero al estado y cuando es el turno de AlphaZero se indica el número de simulaciones realizadas para cada acción.

Por último, en la figura 43 se describe la respuesta del algoritmo ante cualquier jugada inicial. AlphaZero es capaz de realizar siempre una jugada óptima y reconoce que la situación está en todo momento muy igualada (asignando a los estados valores cercanos a 0).

C. Conecta 4

En este apéndice se analiza el comportamiento de AlphaZero Q tras 5000 partidas de entrenamiento. El objetivo es estudiar el aprendizaje de la red neuronal y el comportamiento del algoritmo contra diferentes jugadores. Las partidas utilizadas son algunas de las 20 partidas finales jugadas con esta versión, que como se mencionó en el capítulo 5 logró 10 victorias y 10 derrotas.

En la figura 44 vemos cuatro estados a los que se ha llegado en las partidas contra humanos. En todos ellos había al menos una amenaza, y la red neuronal tras el entrenamiento debería ser capaz de detectarla y otorgar una mayor probabilidad a aquella acción que la evita. La posición a pertenece a la partida 1, donde la red neuronal otorga una probabilidad ligeramente mayor a la columna correcta, situación similar a la que ocurre en la posición d, perteneciente a la partida 10. En la posición b, perteneciente a la partida 3, la red neuronal detecta claramente las tres fichas en vertical del rival y otorga una probabilidad muy superior a la columna correcta. Por último, la posición c, correspondiente a la partida 4, es muy similar a la a. Sin embargo, aquí el oponente tiene dos amenazas, por lo que es imposible evitar la derrota. Sin embargo, la red neuronal detecta qué dos columnas son las amenazadas, otorgándolas una probabilidad muy superior al resto. Es además un gran ejemplo porque la red neuronal, pese a tener dos situaciones prácticamente idénticas (a y c), predice políticas muy diferentes.

En la figura 45 se puede ver el desarrollo de la partida 10, en el que se enfrenta AlphaZero Q como primer jugador (rojas) contra un humano (azules). En esta partida hay dos momentos muy interesantes. El primero es tras la jugada 14 (primer tablero). El oponente está amenazando colocar cuatro fichas en raya en la tercera columna por la derecha. La red neuronal otorga una probabilidad superior a la acción correcta, y el MCTS la corrobora, bloqueando AlphaZero Q en la jugada 15 la amenaza del oponente. Sin embargo, el oponente tiene una clara ventaja en el centro. AlphaZero Q es capaz en las jugadas 16-21 de sortear el peligro y en la jugada 21 forzar a que el oponente tenga que bloquear sus cuatro en raya en la jugada 22, al mismo tiempo que le permite bloquear la amenaza enemiga en la jugada 23.

La partida transcurre tranquilamente hasta la jugada 27, donde AlphaZero Q comete un error, cediéndole la victoria al rival en la jugada 28. En esta posición, coloque donde coloque la pieza AlphaZero Q, si el rival coloca en esa misma columna habrá ganado gracias a las cuatro fichas que puede colocar en horizontal en la última fila. De esto se percata la red neuronal de AlphaZero Q en la jugada 35 (7 jugadas antes de perder), otorgándole probabilidades muy grandes de victoria al segundo jugador (azul). En realidad, ya en la jugada 30 (12 jugadas antes), los valores Q que otorga AlphaZero Q a la acción que va a realizar son inferiores a $-0{,}60$, indicando que sabe que la posición es muy desfavorecedora. En las posiciones finales el valor Q se expresa en morado, donde la derrota es ineludible (valores entre $-0{,}95$ y $-0{,}99$). Esta partida es además una muestra del gran nivel de los oponentes contra los que se ha enfrentado AlphaZero Q, que controlan estrategias de alto nivel.

La paridad es un concepto muy importante en el Conecta 4, ya que nos permite conseguir victorias siguiendo estrategias como la anterior, y que se pueden detectar muchos turnos antes. Esto lo ilustra la increíble victoria que consigue AlphaZero Q en la partida 16, en la que juega como segundo jugador (azul) y se puede ver en la figura 46. Tras 15 movimientos centrados en dos columnas, la red neuronal otorga un valor de 0.91 para el segundo jugador. Este es un valor extraño,

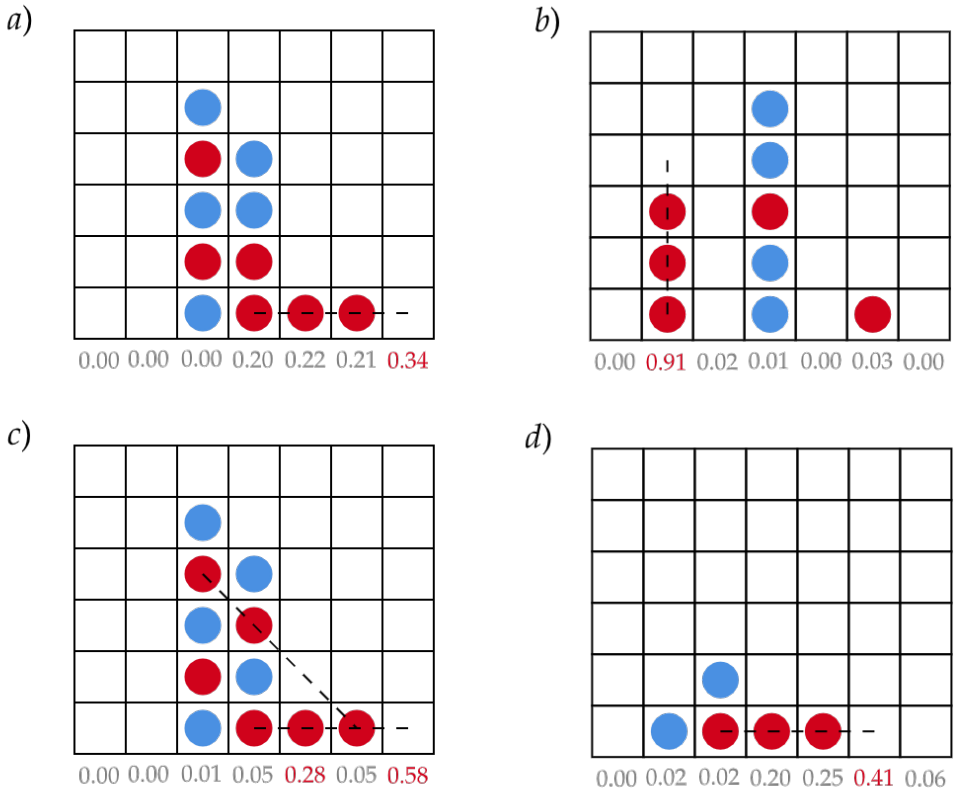


Figura 44: Diferentes posiciones presentes en las partidas en las que AlphaZero Q se ha enfrentado a humanos. Bajo cada columna se indica la probabilidad que otorga la red neuronal en su política a cada acción.

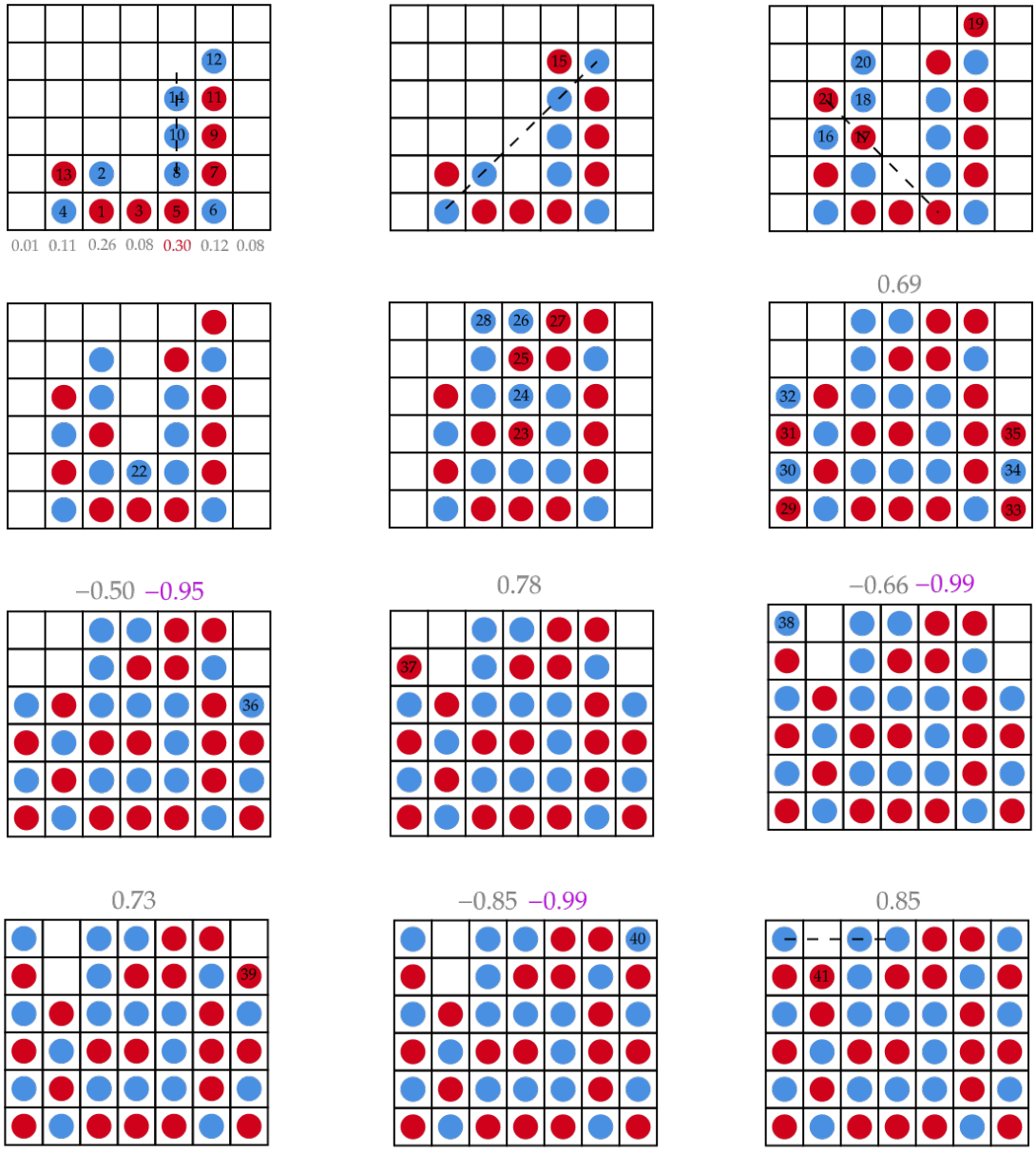


Figura 45: Partida 10, en la que AlphaZero Q juega primero (rojas). Se indica en el interior de cada ficha el orden en el que fue jugada. Bajo el primer tablero se indica la probabilidad otorgada por la red neuronal a cada acción, mientras que sobre los tableros finales se indica en gris el valor dado por la red neuronal a la posición desde el punto de vista del jugador al que le toca jugar. Cuando procede se indica en morado el valor Q de la acción que se va a realizar.

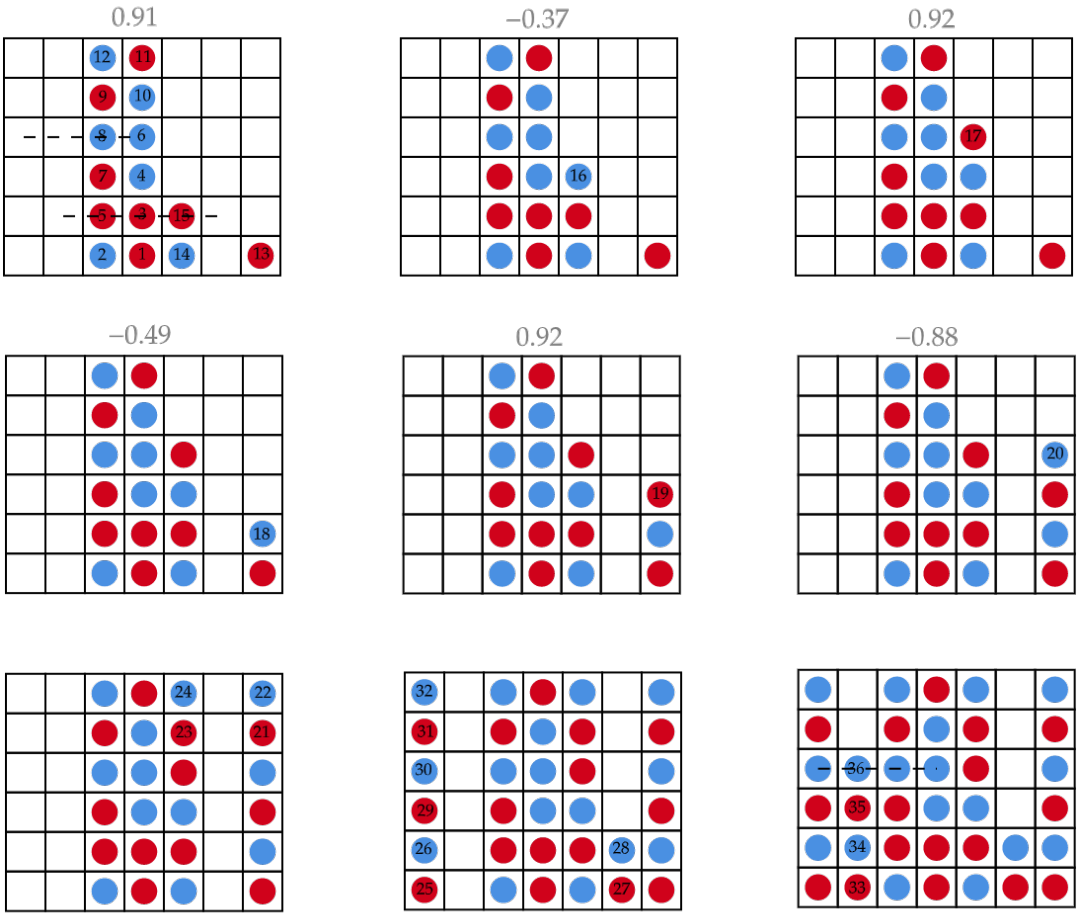


Figura 46: Partida 16, en la que AlphaZero Q juega segundo (azules). Se indica en gris sobre el tablero el valor calculado por la red neuronal para el jugador al que le toca jugar.

puesto que el primer jugador tiene una segunda línea muy fuerte, estando cerca de completar cuatro en raya en dos columnas. A pesar de que los valores estimados por AlphaZero Q en esta versión son bastante imprecisos, es muy raro que otorgue valores tan altos. Además, para el primer jugador unas pocas jugadas más tarde ya asigna también valores muy cercanos a -1 indicando que tiene altas probabilidades de perder.

La justificación de esto se puede entender unas pocas jugadas más tarde. En la primera posición hay una estrategia que da la victoria al segundo jugador. Si nos fijamos en la segunda posición, si el segundo jugador copia las jugadas hechas por el primer jugador en la primera, segunda y sexta columnas (las vacías), mientras que coloca fichas en las columnas cinco y siete cuando lo hace el oponente (lo cual puede hacer porque hay $3 + 5 = 8$ huecos) gana gracias a un cuatro en raya en la cuarta fila, al mismo tiempo que bloquea las amenazas de su oponente en la segunda fila. Vemos que AlphaZero se ciñe a esta estrategia y consigue en la jugada 36 una victoria que había vislumbrado ya en la jugada 16 (20 jugadas antes). Esta partida es un gran ejemplo de la potencia de combinar una red neuronal bien entrenada con múltiples simulaciones para dirigir a AlphaZero

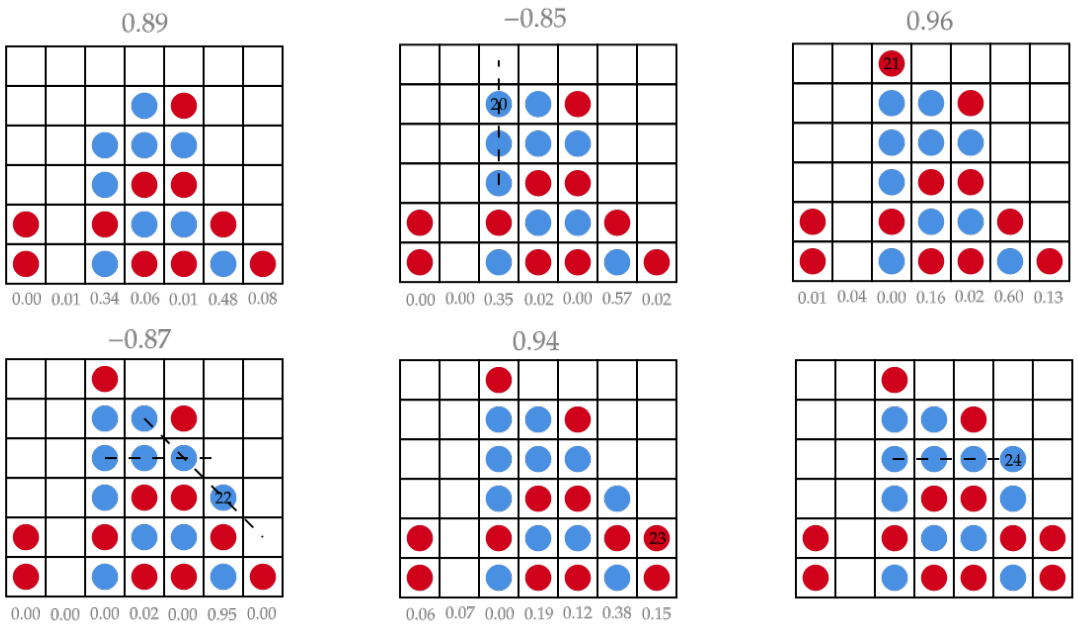


Figura 47: Partida 20, en la que AlphaZero Q juega segundo (azules). Se indica en gris sobre el tablero el valor calculado por la red neuronal para el jugador al que le toca jugar, y bajo cada columna la probabilidad otorgada a esa acción por la red neuronal.

a la victoria. También es interesante porque parece que AlphaZero Q está aprendiendo un concepto de alto nivel.

Por último, tenemos en la figura 47 los últimos movimientos de la partida 20. Vemos que AlphaZero Q es capaz de forzar la victoria con las fichas azules mediante una trampa en la jugada 22. De este ejemplo es especialmente interesante observar las predicciones que hace para el valor (siempre correctas) y para las acciones (dado en casi todo momento una probabilidad mucho mayor a la acción óptima). A pesar de que en la jugada 20 ya se podría haber jugado la trampa y a que la red neuronal otorga una probabilidad mayor a la sexta columna, el MCTS termina decantándose por jugar en la tercera columna, con lo que fuerza a bloquear al oponente antes de tender la trampa. Además, vemos que cuando se llena la tercera columna, la probabilidad de esa acción es cercana a 0, lo que significa que la red neuronal aprende que en una columna llena no se puede introducir una ficha.

Referencias

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, 2018.

[2] Victor Allen. A Knowledge-based Approach of Connect-Four. Master's thesis, Vrije Universiteit, 1988.

[3] Victor Allis. Searching for solutions in games and artificial intelligence. Master's thesis, 1994.

[4] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. arXiv:1705.08439, 2017.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016.

[6] Jason Brownlee. A Gentle Introduction to Cross-Entropy for Machine Learning. `https://machinelearningmastery.com/cross-entropy-for-machine-learning/#:~:text=Cross%2Dentropy%20is%20commonly%20used,difference%20between%20two%20probability%20distributions.`, 2020.

[7] Mark Dominus. Tic Tac Toe State Space Choose Calculation. `https://math.stackexchange.com/questions/485752/tictactoe-state-space-choose-calculation`, 2014.

[8] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[11] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2018.

[12] OpenAI. Part 2: Kinds of RL Algorithms. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`, 2018.

[13] Paper and Pencil Games Developer. Tic Tac Toe: First Move Strategy. `https://funpaperandpencilgames.blogspot.com/2019/02/tic-tac-toe-strategy-tutorial.html`, 2019.

[14] Aditya Prasad. Lessons From Implementing AlphaZero, 2018.

[15] Steve Roberts. The Upper Confidence Bound (UCB) Bandit Algorithm. https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f, 2020.

[16] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, pages 604–609, 2020.

[17] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. arXiv:1502.05477, 2015.

[18] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv:1707.06347, 2017.

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, pages 484–489, 2016.

[20] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, pages 1140–1144, 2018.

[21] David Silver and Julian Schrittwieser. AMA: We are David Silver and Julian Schrittwieser from DeepMind's AlphaGo team. Ask us anything. `https://www.reddit.com/r/MachineLearning/comments/76xjb5/ama_we_are_david_silver_and_julian_schrittwieser/`, 2017.

[22] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, pages 354–359, 2017.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2018.