BackgammonAI: árboles de búsqueda de Montecarlo y redes neuronales
BackgammonAI: Monte Carlo tree search and neural networks

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Miguel Carlos de Arpe Renard

Director
Miguel Palomino Tarjuelo

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Nota: 7/10

Resumen

En este trabajo se exploran los conceptos del aprendizaje automático, profundizando en el aprendizaje por refuerzo. Posteriormente, se estudian árboles de búsqueda de Montecarlo y redes neuronales. Por último se crea un agente de aprendizaje por refuerzo capaz de jugar al tradicional juego inglés *backgammon* mediante una combinación de árboles de búsqueda de Montecarlo y redes neuronales.

Palabras clave

- Aprendizaje automático
- Aprendizaje por refuerzo
- Árboles de búsqueda de Montecarlo
- Redes neuronales
- Tres en raya
- Backgammon

Abstract

In this paper machine learning is explored, going deeper into reinforcement learning. Then, Monte Carlo tree search and neural networks are studied. Finally, a reinforcement learing agent which is able to play the traditional English game backgammon is created, using a combination of Monte Carlo tree search and neural networks.

Keywords

- Machine learning
- Reinforcement learning
- Monte Carlo tree search (MCTS)
- Neural networks
- Tic tac toe
- Backgammon

Índice

1. Introducción **5**

1.1. Motivación 5

1.2. Objetivos 5

1.3. Plan de trabajo 5

2. Aprendizaje Automático **7**

2.1. Introducción 7

2.2. Tipos de Aprendizaje Automático 7

2.3. Aprendizaje por refuerzo 8

2.4. Formalización del aprendizaje por refuerzo 8

2.5. Tipos de aprendizaje por refuerzo 10

2.5.1. Basados en la política 10

2.5.2. Basados en el valor 10

2.5.3. Basados en el modelo 10

2.5.4. Combinaciones 10

3. Redes neuronales **11**

3.1. Introducción 11

3.2. Redes neuronales monocapa 11

3.3. Regresión lineal 11

3.3.1. Función de pérdida 12

3.3.2. Descenso del gradiente 12

3.3.3. Red neuronal 13

3.4. Redes neuronales multicapa 13

3.4.1. Funciones de activación 14

3.4.2. ReLU 14

3.4.3. Sigmoide 14

3.4.4. Tanh 14

4. Árboles de búsqueda de Montecarlo **16**

4.1. Introducción 16

4.2. Funcionamiento 16

5. Implementación **18**

5.1. Estructura del árbol de búsqueda de Montecarlo 18

5.1.1. Aristas 18

5.1.2. Política del árbol 19

5.1.3. Política de despliegue 19

5.1.4. Selección de acción 20

5.1.5. Código 20

5.2. Aprendizaje de la red neuronal 21

5.3. Tres en raya 21

5.3.1. Reglas 21

5.3.2. Implementación con red monocapa 22

5.3.3. Implementación con red multicapa 23

5.4. *Backgammon* 27

5.4.1. Reglas 27

5.4.2. Modificaciones en el árbol de búsqueda de Montecarlo 28

5.4.3. Infraestructura de implementación 29

5.4.4. Resultados primera red 29

5.4.5. Resultados segunda red 30

6. Conclusión **31**

1. Introducción

1.1. Motivación

Desde pequeño me han apasionado los juegos de mesa, especialmente los juegos 1 contra 1 con información completa, ya que suelen ser muy estratégicos y lógicos. Siempre me ha gustado el ajedrez, hace unos años comencé a jugar a través de internet y fue entonces cuando tuve mi primer contacto con la inteligencia artificial: los motores de ajedrez. Al investigar un poco más sobre ellos, observé que eran muy superiores a los mejores humanos del mundo y desde entonces me he preguntado cómo funcionan.

1.2. Objetivos

La empresa *DeepMind* ha creado agentes de aprendizaje por refuerzo que juegan a cualquier juego 1 contra 1 con información completa (incluyendo el ajedrez), utilizando un enfoque combinado de árboles de búsqueda de Montecarlo y redes neuronales. El objetivo de este trabajo es entender cómo funcionan ambos componentes e implementar un agente que los use para jugar un tradicional juego inglés, llamado *backgammon*.

1.3. Plan de trabajo

Para empezar vamos a estudiar el concepto del aprendizaje automático, centrándonos en el aprendizaje por refuerzo, que va a ser el núcleo de este trabajo. Posteriormente, se van a analizar los conceptos de redes neuronales y árboles de búsqueda de Montecarlo. Después, vamos a demostrar lo aprendido creando un agente que juegue al tres en raya, un juego muy simple que nos va a ayudar a asentar conceptos. Por último, se implementará un agente que juegue al *backgammon*.

Introduction

Motivation

From a young age, I've been passionate about board games, especially 1 vs 1 full information games, as they are very logical and strategical. I've always enjoyed chess, years ago I started playing on the internet and had the first interaction with machine learning: chess engines. Researching more about them I found they are superior to the best humans and I've wondered how they work since then.

Goals

The company *DeepMind* created reinforcement learning agents which play any 1 vs 1 full information game (including chess), using a method which combines Monte Carlo tree search and neural networks. The goal of this thesis is understanding how both elements work and implementing an agent which uses them to play a traditional English game, known as backgammon.

Work plan

We will begin by studying the concept of machine learning, focusing on reinforcement learning, which is the focus of this paper. Then, we will learn about neural networks and Monte Carlo tree search. After that, we will show what was learnt by implementing an agent which plays tic tac toe, a very simple game which will help us establish the concepts. Finally, an agent which plays backgammon will be implemented.

2. Aprendizaje Automático

Para elaborar esta sección, se han utilizado como referencias principales la sección 1 de [GK20] y las secciones 1 y 2 de [SB20]. Para entender cómo estructurar un trabajo de fin de grado y afinar mi entendimiento de alguna cuestión, he utilizado [Alb].

2.1. Introducción

Hasta hace poco, la gran mayoría de programas informáticos diseñados para ejecutar tareas se pensaban como un conjunto de acciones deterministas y reglas. Esta forma de enfocar la resolución de problemas funciona muy bien cuando se puede modelar un problema a la perfección y se sabe cómo actuar ante cualquier estado, pero resulta problemática en ciertos casos; veamos un ejemplo.

Digamos que queremos un programa que identifique imágenes de dígitos escritos a mano. Una tarea tan simple para un humano como esta es muy complicada de formalizar y por tanto muy difícil de escribir un programa clásico que la resuelva. Es en estos casos en los que el aprendizaje automático resulta muy útil.

El aprendizaje automático (*machine learning*) es un campo de la inteligencia artificial en el que se desarrollan algoritmos estadísticos cuyo propósito es tomar decisiones o realizar acciones sin instrucciones explícitas. A continuación, veremos los tipos principales de aprendizaje automático.

2.2. Tipos de Aprendizaje Automático

Hay tres tipos de aprendizaje automático: el aprendizaje supervisado (*supervised learning*), el aprendizaje no supervisado (*unsupervised learning*) y el aprendizaje por refuerzo (*reinforcement learning*). Hay otros tipos de aprendizaje que son mezclas de estos tres, como por ejemplo el aprendizaje semi-supervisado (*semi-supervised learning*) que no se describirán en este texto.

Un algoritmo de aprendizaje supervisado obtiene datos etiquetados de entrada con sus respectivos datos de salida y trata de deducir patrones y relaciones en dichos datos. El objetivo es que al recibir nuevos datos etiquetados de entrada, el algoritmo sea capaz de deducir sus datos de salida esperados.

Veamos un ejemplo en el que dicho tipo de aprendizaje es útil. Digamos que una agencia inmobiliaria quiere estimar el precio de las casas que pone a la venta. Para ello, la agencia cuenta con una base de datos con casas que ya se han vendido y una serie de información de entrada sobre dichas casas: metros cuadrados, ubicación, número de habitaciones... y también los datos que van a querer estimar posteriormente, es decir, el precio de venta de las casas. Para este propósito se podría entrenar a un agente de aprendizaje supervisado con la base de datos de casas ya vendidas y dados datos de entrada de las próximas casas que quieran vender, el agente podrá estimar el precio.

El aprendizaje no supervisado es parecido al aprendizaje supervisado en tanto que trata de deducir patrones y relaciones en los datos que le son suministrados, pero dichos datos no están etiquetados.

Un ejemplo en el que podríamos usar un algoritmo de aprendizaje supervisado es la agrupación de datos. Digamos que un almacén de una empresa almacena objetos muy distintos, que tienen una relación no trivial entre sí. Una manera de agrupar dichos objetos sería dárselos a un

algoritmo de aprendizaje no supervisado para que los agrupe.

El aprendizaje por refuerzo es el que vamos a explorar en profundidad en este trabajo, por tanto, merece un epígrafe aparte.

2.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo es radicalmente distinto a los otros dos tipos. En este tipo de aprendizaje, el agente trata de maximizar una recompensa numérica a través de sus interacciones con el entorno a lo largo de un intervalo de tiempo.

Como en todos los tipos de aprendizaje automático, el agente no recibe instrucciones concretas sobre cómo realizar su objetivo, en este caso sobre cómo maximizar la recompensa, lo que implica que debe descubrir la acción más óptima que puede realizar sobre un estado determinado.

Además, la acción que realiza el agente afecta al estado del entorno, con lo cual el agente no solo ha de optimizar la toma de decisiones teniendo en cuenta la recompensa inmediata sino también cómo esto afectará al entorno, y por tanto, a las posibles recompensas futuras.

Un uso de los algoritmos de aprendizaje por refuerzo es crear jugadores extremadamente buenos en juegos como el ajedrez o el go, que desde hace años superan, con creces, a los mejores humanos.

Un problema central al aprendizaje por refuerzo es el conflicto entre explotación y exploración. El agente ha de encontrar la acción que maximice la recompensa a lo largo del tiempo, pero no tiene información completa, por tanto debe elegir entre hacer la acción que piensa que es la mejor con la información actual (explotación) o realizar otra acción con el fin de entender si le puede dar una mejor recompensa (exploración), obteniendo más información en dicho proceso.

2.4. Formalización del aprendizaje por refuerzo

Hasta ahora todo lo visto sobre el aprendizaje por refuerzo es muy abstracto y poco específico; veamos, concretamente, cuáles son los elementos del aprendizaje por refuerzo.

1. Elementos internos al problema:
    - *a*) Estado: un modelo del entorno en el que se realiza el problema.
    - *b*) Función de recompensa: una función que indica la recompensa inmediata que el agente recibe al realizar una acción sobre un estado determinado.
2. Elementos externos al problema:
    - *a*) Política: una función (que suele ser estocástica) que dicta cómo se comporta el agente, es decir, cuál es la acción que toma en cada estado.
    - *b*) Función de valor: asocia a cada estado un valor. Esta función indica la recompensa esperada que podemos recibir de un estado determinado, en contraste con la función de recompensa, que muestra la recompensa inmediata de una acción.

Además, definamos:

- s_t el estado en el instante t,

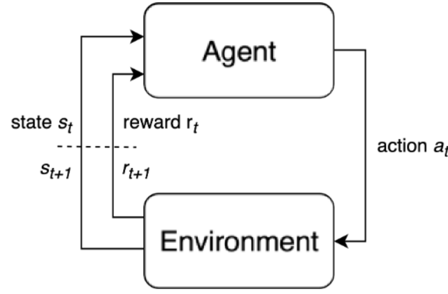

Figura 1: Bucle del aprendizaje por refuerzo, figura 1.2 de [GK20]

- a_t la acción que se toma en tiempo t,
- $\mathcal{R}(s_1, a, s_2)$ la función de recompensa, que indica la recompensa obtenida al pasar de s_1 a s_2 a través de la acción a,
- $P(s_2 \mid s_1, a)$ la función de transición, que indica la probabilidad de pasar de s_1 a s_2 a través de la acción a,
- $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$ la recompensa en el instante t.

Definimos pues, el problema de aprendizaje por refuerzo como un bucle: en el instante t el estado es s_t, el agente toma una acción a_t dictada por su política, lo cual lleva al estado s_{t+1} y da una recompensa r_t. Podemos ver este proceso en la figura 1.

Un episodio $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ \ldots \ , (s_T, a_T, r_T)$ es una instancia del problema que queremos que el agente resuelva. Teóricamente, T puede ser infinito, pero en la práctica definimos un T máximo, tras el cual damos el problema por terminado. Un agente de aprendizaje por refuerzo necesita muchos episodios para aprender, desde cientos hasta millones dependiendo de la complejidad del problema y del método usado para aprender a resolverlo.

Ahora hemos de formalizar la función que el agente de aprendizaje por refuerzo maximiza. Para ello, definimos la siguiente función para un episodio $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), ..., (s_T, a_T, r_T)$:

$$R(\tau) = \sum_{t=0}^{T} \gamma^T r_t = r_0 + \gamma r_1 + \gamma^2 r_2 + \ \ldots \ + \gamma^T r_T \qquad \gamma \in [0, 1] \tag{1}$$

En esta ecuación, γ sirve para ajustar el peso que tienen las recompensas dependiendo de su proximidad al instante inicial. Cuanto más pequeña hagamos γ, menos nos importan las recompensas más distantes. Cuanto más grande sea γ, más difícil será resolver el problema, ya que es complicado entender cómo el efecto de las acciones sobre el entorno afecta a las recompensas muy distantes en el tiempo.

Como tanto la política de un agente como la transición entre estados y la función $\mathcal{R}$ de recompensa pueden ser probabilísticas, definimos el objetivo $J(\pi)$ a maximizar de la siguiente manera. Sea π una política:

$$J(\pi) = \mathbb{E}_{\tau \sim \pi}[R(\tau)] \tag{2}$$

Vemos que la función objetivo no es más que la media de la función R a lo largo de muchos episodios.

2.5. Tipos de aprendizaje por refuerzo

Ahora que hemos definido formalmente el problema del aprendizaje por refuerzo y la función que queremos maximizar, veamos los diferentes tipos de agente que podemos utilizar: cada uno tiene sus pros y contras, y cuál sea mejor o peor depende de la naturaleza del problema específico.

2.5.1. Basados en la política

Este tipo de algoritmos trata de aprender una buena política π. Una buena política genera acciones que dan lugar a episodios que maximizan la función objetivo J. Es el tipo de algoritmos más intuitivo, y también el más general, es decir, puede resolver problemas muy variados. Además, esta clase de algoritmos garantizan convergencia local.

2.5.2. Basados en el valor

Estos algoritmos buscan aprender una función que asocia a cada estado un valor, dependiendo de las recompensas que espera poder obtener de dicho estado. La función de valor puede definirse de dos maneras. Sean s estado, π política, a acción:

$$V^{\pi}(s) = \mathbb{E}_{s_0=s,\tau\sim\pi}[R(\tau)] \tag{3}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s_0=s,a_0=a,\tau\sim\pi}[R(\tau)] \tag{4}$$

La primera ecuación mide cómo de bueno es el estado s asumiendo que seguimos la política π. La segunda es ligeramente diferente, indica lo bueno que es s asumiendo que la acción que se toma en este instante es a y para las siguientes se sigue π.

Esta clase de algoritmos buscan construir una función de valor que evalúe cómo de bueno o malo un estado es en realidad y, utilizando esta función, eligen las acciones que llevan a los estados con mayor valor.

2.5.3. Basados en el modelo

Para utilizar este tipo de algoritmos es necesaria una muy buena comprensión de la función de transición P. Cuando esta función es sabida (o se tiene una aproximación muy cercana) el agente puede simular secuencias de acciones sobre el estado actual, sin producir cambios en el entorno real, ver en qué estado puede acabar después de cada secuencia y tomar la acción más prometedora.

La mayor limitación de este tipo de algoritmos es la dificultad de poder deducir P en escenarios reales.

2.5.4. Combinaciones

Esta clase de algoritmos combinan 2 o más de los vistos anteriormente y son en los que se va a centrar este trabajo. En concreto, veremos algoritmos que combinan el enfoque basado en el valor y en el modelo.

3. Redes neuronales

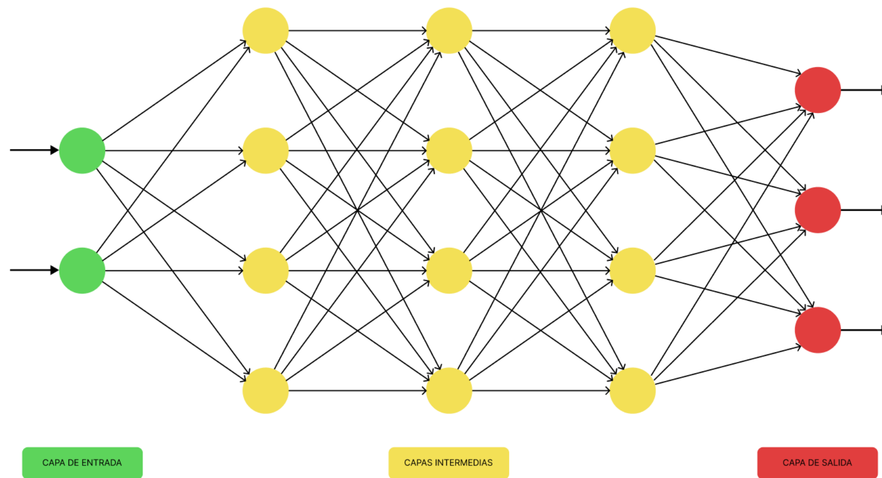

Figura 2: Estructura genérica de una red neuronal

La referencia principal utilizada para escribir esta sección ha sido las secciones 1, 2, 3, 4 y 5 de [Zha+].

3.1. Introducción

Actualmente, una herramienta central a todos los tipos de aprendizaje automático son las redes neuronales.

Una red neuronal es un grupo de neuronas interconectadas que se mandan señales. Toda red neuronal tiene una capa de neuronas de entrada, otra de salida y opcionalmente capas intermedias, denominadas capas ocultas. Podemos ver la estructura genérica de una red neuronal en la figura 2. Comencemos viendo el tipo más simple de red neuronal, el que no tiene ninguna capa intermedia.

3.2. Redes neuronales monocapa

Las redes neuronales de una sola capa, también llamadas redes neuronales lineares conectan la capa de entrada con la de salida directamente.

Para introducir la mayoría de los conceptos clave de las redes neuronales, vamos a estudiar el problema más simple que estas redes son capaces de resolver, la regresión lineal.

3.3. Regresión lineal

La regresión lineal consiste en encontrar la mejor aproximación a través de una función lineal $f : \mathbb{R}^n \to \mathbb{R}$, de una serie de puntos, es decir, pares (x, y) donde $x \in \mathbb{R}^n, y \in \mathbb{R}$. Dado que buscamos una f función lineal, podemos escribir $f(x) = x\,W^T + b$ donde $W \in \mathbb{R}^n$.

Para ver un caso en el que esto puede ser útil, volvamos a un ejemplo de la sección 2.2: la predicción de precios de casas. En este caso, los pares (x, y) serian las características de las casas que ya se han vendido y sus precios respectivamente. Si asumimos que la relación entre cada característica de las casas y su precio es lineal, resulta evidente que conseguir hallar los valores W y b que mejor aproximen $f(x) = x\,W^T + b \approx y$ nos ayudaría mucho a predecir el precio de otras casas.

Para hallar dichos valores, necesitamos dos cosas: una manera de decidir cómo de mala o buena es una aproximación y una manera de actualizar W y b para acercarnos a la mejor aproximación. Para lo primero, necesitamos una función que cuantifique el error que estamos cometiendo, l, comúnmente denominada *función de pérdida*.

3.3.1. Función de pérdida

La función de pérdida es una noción general en el aprendizaje con redes neuronales que nos permite cuantificar el error cometido.

Digamos que tenemos K pares de datos conocidos: $(x_i, y_i)\ \forall i \in 1, ..., K$, denotamos $\hat{y_i} = f(x_i)$. Podríamos pensar que la función de pérdida es $|\hat{y_i} - y_i|$, pero queremos que cuanto más grande sea el error más penalice, de forma cuadrática, por tanto, la definimos como $l_i(W, b) = \frac{1}{2}(\hat{y_i} - y_i)^2$ (el $\frac{1}{2}$ es por comodidad, para cuando derivemos posteriormente). Definir la función de pérdida con el componente cuadrático tiene sus pros, ya que fuerza a que el modelo evite errores muy grandes, pero también sus contras, ya que una anomalía en los datos conocidos puede descuadrar mucho el modelo.

Definimos $L(W, b) = \frac{1}{K}\sum_{i=1}^{K} l_i(W, b)$ y es evidente que buscamos $W', b' = argmin\, L(W, b)$. Al contrario que la gran mayoría de problemas que resuelven las redes neuronales, la regresión lineal tiene solución analítica, sin embargo, necesitamos un enfoque más general que podamos aplicar a diversos problemas.

3.3.2. Descenso del gradiente

La idea principal de este método de minimizar el error es hacerlo de manera iterativa, desplazándonos en el sentido negativo de la derivada. Para ello, hay distintos enfoques: se puede hacer por cada par (x_i, y_i), es decir, desplazarnos en el sentido negativo de la derivada de l_i; para todos los pares, es decir, desplazarnos en el sentido negativo de la derivada de L; o agrupar los pares en subconjuntos y desplazarnos en el sentido negativo de la derivada de las sumas de sus l_i.

Se suele utilizar el ultimo método, ya que el primero puede producir variaciones demasiado grandes en los pesos y el segundo es demasiado lento. Sean $\mathcal{B}$ los índices de un subgrupo, aplicamos modificaciones sobre W y b de la siguiente manera:

$$W \leftarrow W - \frac{\eta}{|\mathcal{B}|}\sum_{i \in \mathcal{B}} \partial_W l_i(W, b) \tag{5}$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|}\sum_{i \in \mathcal{B}} \partial_b l_i(W, b) \tag{6}$$

donde $\eta > 0$ es un valor pequeño que se denomina el factor de aprendizaje: cuanto más grande

sea este valor, más rápido aprende el modelo, pero si es demasiado grande se puede aprender de manera demasiado brusca y no tender nunca a la solución optima.

En el caso particular de la regresión lineal, las fórmulas para actualizar W y b son:

$$W \leftarrow W - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_i (x_i W^T + b - y_i) \tag{7}$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (x_i W^T + b - y_i) \tag{8}$$

El proceso general para entrenar una red neuronal utilizando el método del descenso del gradiente es el siguiente:

- Dividir los K elementos conocidos en subgrupos.
- Para cada subgrupo, actualizar W y b según las fórmulas vistas.
- Calcular la función de pérdida total L. Si la diferencia con la L de la anterior iteración es más pequeña que una cierta constante, hemos acabado, en caso contrario, volver al paso anterior.

3.3.3. Red neuronal

La estructura de la red neuronal para este problema son n neuronas de entrada y una de salida. Cada uno de los n valores de entrada es recibido por una neurona de entrada y el cómputo de f es la salida. Lo que en este problema hemos llamado W se denominan pesos de la red y b se denominan sesgos. Cuando una neurona (que no es de la capa de entrada) recibe valores de entrada multiplica cada uno por su peso, suma el sesgo y lo pasa a la siguiente capa.

Pese a que hemos visto la regresión lineal de $f : \mathbb{R}^n \to \mathbb{R}$ es fácil entender como este concepto se puede extender a una función $g : \mathbb{R}^n \to \mathbb{R}^d$ lineal.

Sin embargo, nos encontramos con el principal obstáculo que hace que casi todos los problemas de aprendizaje automático requieran redes multicapa: la gran mayoría de estos problemas no pueden ser modelados por una función lineal. Podríamos forzar a que esta red compute una función cuadrática, o de orden 3, pero en realidad la mayoría de veces ni siquiera sabemos qué clase de función puede mostrarnos la relación entre los datos y se ha demostrado que el enfoque más efectivo para solucionar este problema es la utilización de redes con múltiples capas.

3.4. Redes neuronales multicapa

Las redes neuronales multicapa introducen capas entre la de entrada y la de salida, llamadas capas ocultas. Es sencillo ver que si hacemos que la funcionalidad de las neuronas sea la misma de antes (coger las entrada, multiplicarlas pesos, sumar el sesgo y mandar el resultado por la salida) la red neuronal seguiría siendo una transformación afín.

Por ello, para realmente conseguir redes que puedan expresar mayor complejidad, hemos de introducir otra noción: las funciones de activación.

3.4.1. Funciones de activación

Las funciones de activación son funciones no lineales que se aplican tras multiplicar por los pesos y sumar el sesgo. Son un concepto clave en las redes neuronales ya que permiten que estas creen transformaciones no afines. A la hora de elegir qué función de activación usar buscamos dos cosas: que la función sea sencilla y que su derivada también lo sea (ya que la necesitaremos para el descenso del gradiente).

A continuación se presentan las tres funciones de activación más usadas.

3.4.2. ReLU

La función ReLU (*rectified linear unit*) es:

$$ReLU(x) = max(x, 0) \tag{9}$$

Es decir, si el valor es positivo, no lo cambia, y si es negativo toma el valor 0. Además, tiene una derivada muy sencilla: 0 si x es negativo y 1 si es positivo. El único problema de esta función es que no es derivable en 0, pero simplemente asumimos que su derivada en ese punto es 0.

$$\frac{d}{dx}ReLU(x) = \begin{cases} 1 & \text{si } \; x > 0 \\ 0 & \text{si } \; x \leq 0 \end{cases} \tag{10}$$

ReLU es la función de activación más utilizada, ya que es muy simple y suele dar resultados muy satisfactorios.

3.4.3. Sigmoide

La función sigmoide es:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

Esta función de activación convierte valores en el rango $(-\infty, +\infty)$ al rango $(0, 1)$. Su derivada es:

$$\frac{d}{dx}sigmoid(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = sigmoid(x)(1 - sigmoid(x)) \tag{12}$$

Tanto la función sigmoide como su derivadas son más caras de calcular que las de ReLU, pero pueden ser útil en casos en los que queremos tener en cuenta las salidas negativas de las neuronas.

3.4.4. Tanh

La función tanh (tangente hiperbólica) es:

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{13}$$

Es muy similar a la sigmoide, ya que convierte valores en el rango $(-\infty, +\infty)$ al rango $(-1, 1)$, y tanto los gráficos de sus funciones como los de su derivadas son parecidos.

$$\frac{d}{dx} tanh(x) = 1 - tanh^2(x) \tag{14}$$

4. Árboles de búsqueda de Montecarlo

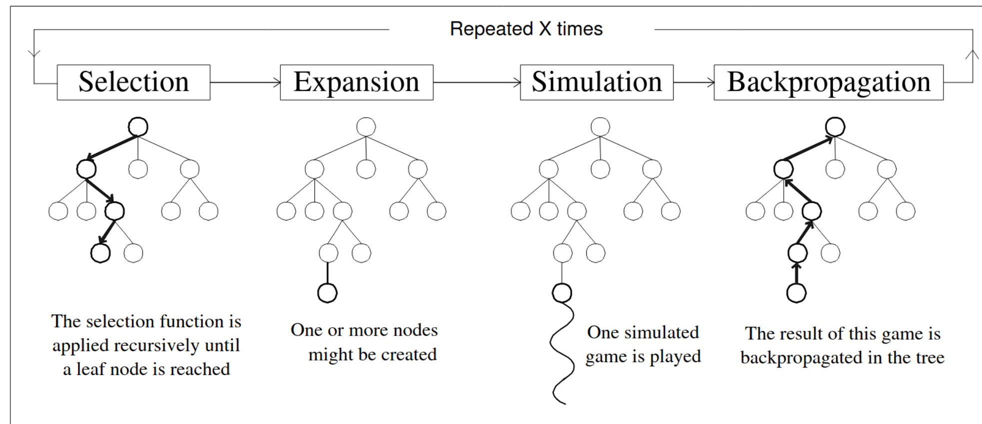

Figura 3: Bucle del árbol de búsqueda de Montecarlo, figura 2 de [VCU07]

Para elaborar esta sección se han utilizado como referencias principales las secciones 8.10 y 8.11 de [SB20], la sección 4.1 de [Mur24] y el video [Tho].

4.1. Introducción

Los árboles de búsqueda de Montecarlo son un tipo de algoritmo de despliegue (*rollout algorithm*) que se utilizan para tomar decisiones. Lo principal que necesitamos para poder utilizar esta clase de algoritmos es un conocimiento total de la función de transición del entorno. Si la conocemos, a la hora de tomar una decisión, podemos antes simularla y ver su resultado.

De hecho, en caso de que las decisiones que se toman sobre el entorno no sean solo nuestras, sino que más agentes lo puedan modificar, también podemos simular cada una de las posibles acciones de dichos agentes y ver cómo todo ello afecta al entorno antes de tomar nuestra decisión. Esto es precisamente lo que hacen los algoritmos de despliegue. Veamos concretamente los árboles de búsqueda de Montecarlo.

4.2. Funcionamiento

El árbol de búsqueda de Montecarlo parte de un nodo, el estado actual, cada nodo añadido representa un estado, y cada arista entre dos nodos la transición de un estado a otro. Las aristas también pueden almacenar información de la transición, que dependen de la implementación específica, como por ejemplo cuántas veces se ha simulado o el valor medio de las simulaciones. A partir del nodo base, se ejecuta el siguiente bucle (que podemos observar en la figura 3) hasta un límite de tiempo o de pasos por el bucle.

1. Selección: Utilizando la *política del árbol*, se selecciona un nodo hoja.
2. Expansión: Si la hoja seleccionada no está en el límite de profundidad del árbol (que puede no existir), se expande, creando un hijo para cada posible acción.

3. Simulación: Utilizando la *política de despliegue*, se lanza una simulación desde el nodo seleccionado (o, en caso de haber expandido, desde uno de sus hijos), que termina al pasar un cierto tiempo o al llegar a un estado terminal, y se le da un valor al resultado de esa simulación.
4. Actualización: Con el valor obtenido en la simulación, se actualizan todos los nodos que han llevado hasta la hoja de la que ha partido la simulación.

Hay dos conceptos clave que faltan por definir: la política del árbol y la política de despliegue. La política del árbol dicta cómo seleccionar el nodo hoja que se va a expandir o desde el que se va a simular. La política de despliegue indica cómo simular las acciones de nuestro agente y los demás agentes que interactúan con el entorno.

Si observamos la figura 3, la política del árbol se aplicaría de manera recursiva en la fase de selección para elegir cuál es la hoja que expandimos. Desde el hijo, fruto de la expansión de la hoja seleccionada, se aplica la política de despliegue en la fase de simulación para definir nuestro comportamiento, y el de los otros agentes, a lo largo de ella.

Cuando termina el bucle, el agente selecciona una acción a tomar en base al árbol construido, la ejecuta, y mantiene en su memoria el subárbol correspondiente a dicha acción, descartando todo lo demás.

Esta explicación de los árboles de búsqueda de Montecarlo es muy genérica ya que el algoritmo está pensado para diversas aplicaciones. En el apartado de implementación veremos el árbol concreto que vamos a usar para nuestro agente.

5. Implementación

Con toda la teoría presentada, nuestro objetivo es crear un agente que juegue al *backgammon*. Para ello, utilizaremos un algoritmo de aprendizaje por refuerzo que combina los árboles de búsqueda de Montecarlo con las redes neuronales.

Vamos a comenzar haciendo una implementación para el tres en raya y luego extrapolaremos lo aprendido al *backgammon*. Ambos juegos son 1 contra 1 por turnos, con información completa y la única diferencia sustancial es que el *backgammon* tiene probabilidades asociadas así que habrá que hacer ligeras modificaciones para tenerlo en cuenta.

La idea principal es que vamos a tener una red neuronal que asigne un valor a los estados (el tablero y de quién es el turno), dependiendo de lo buenos o malos que sean para un jugador. Una red que funciona correctamente da valores cercanos a 0 si el estado llevará a un empate, y más positivos o negativos en función de si un jugador o el otro tienen más ventaja respectivamente. En nuestra implementación, denotaremos 1 como victoria de un jugador, -1 del otro y 0 empate.

El árbol de búsqueda de Montecarlo va a usarse para explorar las posibles acciones a tomar en un estado específico, ayudándose de la red neuronal. Durante el entrenamiento, tanto la red neuronal como el árbol de búsqueda de Montecarlo serán compartidos entre ambos jugadores.

Solo se van a mostrar algunas funciones clave del código, la implementación completa puede encontrarse en [Arp].

Entrenaremos la red neuronal con los conocimientos adquiridos en cada partida. Es la red la que persiste y mejora a lo largo del aprendizaje, ya que para cada partida se inicializa un árbol de búsqueda de Montecarlo nuevo. Veamos ahora la estructura del árbol que vamos a usar para la implementación.

5.1. Estructura del árbol de búsqueda de Montecarlo

Cada nodo del árbol es un estado del entorno (tablero y turno), y sus hijos son posibles estados a los que se llega a través de jugar un turno. Una arista conecta a cada hijo con el nodo, representando la transición de estados que provoca jugar un posible turno.

Dado que dimos una definición muy general del concepto de árbol de búsqueda de Montecarlo, hay cuatro cosas que tenemos que especificar para nuestra implementación: la información en cada arista, la política del árbol, la política de despliegue y la manera de seleccionar la acción a tomar al final del bucle.

Como los dos jugadores comparten el árbol de búsqueda para todas esas funciones hay que tener en cuenta de quién es el turno, ya que uno buscará tomar acciones que lleven a valores positivos y el otro a negativos. Denotamos con $x \in \{1, -1\}$ de quién es el turno.

5.1.1. Aristas

Queremos que cada arista guarde información que nos permita entender cómo de buena es la acción que lleva del estado padre al hijo. Para ello, la información que hay en cada arista es:

- N: número de simulaciones en las que en la etapa de selección se ha pasado por esta arista,
- Q: valor medio de todas esas simulaciones,

- P: probabilidad que asigna la red neuronal a la acción que lleva del padre al hijo (las P de los hijos de un nodo siempre suman 1)

Cuando se expande un nodo, N y Q de todas las aristas se inicializan a 0, y cuando ocurre una simulación desde un nodo hoja al que se ha llegado (en la etapa de selección) a través de una arista, se actualizan en consecuencia en dicha arista (esto ocurre en la fase "actualización").

Queremos que P exprese cómo de buena es una acción para la red neuronal en forma de probabilidad (para que sea oportuno comparar las acciones entre ellas), pero nuestra red neuronal devuelve valores de las posiciones, no probabilidades. Para pasar dichos valores a probabilidades utilizamos la función *softmax*. Sean $v_1, v_2, \ldots, v_k$ los valores que da la red neuronal a cada uno de los hijos y $x \in \{1, -1\}$ de quién es el turno, establecemos el P de cada arista i-ésima que lleva al hijo i-ésimo utilizando la siguiente formula:

$$P = \frac{e^{x \cdot v_i}}{\sum_{j=1}^{k} e^{x \cdot v_j}} \tag{15}$$

5.1.2. Política del árbol

La política del árbol que vamos a utilizar se basa en el método de selección UCB (*Upper Confidence Bound*) para el problema del bandido de n brazos. Este es un problema clásico del aprendizaje por refuerzo, en el que a un agente se le presentan n máquinas tragaperras y se le dan k tiradas. Las máquinas tragaperras dan recompensas de una manera probabilística desconocida por el agente.

El agente ha de equilibrar exploración y explotación (explicado en la sección 2.3) para maximizar las recompensas a lo largo de las k tiradas. Una de las mejores maneras de equilibrar conocidas es el método UCB, del cual voy a utilizar una variación. Para profundizar más en este problema, ver la sección 2 de [SB20]. Una variante del UCB también fue utilizada en *AlphaZero*, uno de los algoritmos desarrollados por *DeepMind*. Para más información sobre *AlphaZero* ver [Sil+16].

Sean $N_1, N_2, \ldots, N_k$, $Q_1, Q_2, \ldots, Q_k$, $P_1, P_2, \ldots, P_k$ los N, Q y P de las k aristas que salen del nodo en el que estamos, elegimos la acción i de la siguiente manera:

$$i = argmax(x \cdot Q_i + c \cdot P_i \sqrt{\frac{\ln((\sum_{j=1}^{k} N_j) + 1)}{N_i + 1}}) \tag{16}$$

donde $c > 0$ es una constante que indica cuánto exploramos. Esta fórmula es igual que la de UCB salvo por dos cambios. El primero es multiplicar la raíz por P_i, esto se hace para dar una mayor importancia a las acciones que la red neuronal considera mejores. El segundo es el $+1$ arriba y abajo de la fracción, la razón por la que hacemos este cambio es que cuando dos o más hijos de un nodo se han simulado 0 veces queremos seleccionar el mejor según la red neuronal.

5.1.3. Política de despliegue

Queremos que la política de despliegue sea computacionalmente barata, ya que hacer simulaciones rápidas nos va a permitir efectuar muchas simulaciones y poder jugar muchos turnos en estas, lo cual aumenta considerablemente la habilidad del agente.

Por tanto, la política de despliegue va a estar basada simplemente en los valores que nos de la red neuronal pasados por *softmax*, es decir, los P.

5.1.4. Selección de acción

Por último, para la selección de la acción real a tomar al terminar el bucle (desde la raíz del árbol), volvemos a usar *softmax* para seleccionar la acción i con probabilidad:

$$p_i = \frac{e^{x \cdot Q_i}}{\sum_{j=1}^{k} e^{x \cdot Q_j}} \tag{17}$$

Nótese que los valores Q_i se han actualizado con la información de todas las simulaciones.

5.1.5. Código

Para jugar una partida completa, se usa el siguiente código:

```
def playGame() -> tuple[torch.Tensor, int]:
    playerStart = random.choice([1,-1])
    states = []
    mt = MonteTree(None, TicTacToeBoard(playerStart))
    while not mt.board.isOver():
        mt = mt.playTurn()
        states.append(mt.board.board + [mt.board.player])
    return torch.tensor(states), mt.board.winner()
```

Donde `playerStart` indica qué jugador empieza y se elige aleatoriamente, lo cual puede ser relevante ya que si se entrenase la red neuronal con el mismo jugador comenzando siempre esto podría conllevar resultados no deseados. Por ejemplo, en el tres en raya la red podría dar menos importancia al valor de entrada que determina de quién es el turno, ya que, si empieza siempre el mismo jugador, ese valor está determinado inequívocamente por el estado del tablero.

Continuando con el análisis del código, `states` va a almacenar todos los estados por los que pasa la partida y `mt` es la instancia del árbol de búsqueda de Montecarlo. Una partida consiste en ejecutar un bucle que usa `mt` para jugar turnos hasta que la partida termina. El método devuelve `states` en forma de tensor y quién ha sido el ganador de la partida, datos que se usarán para entrenar la red neuronal.

El código de jugar un turno es:

```
class MonteTree:
    def playTurn(self) -> "MonteTree":
        for i in range(MonteTree.simulationsPerTurn):
            selected = self.nodeSelection()
            selected.expansion()
            if len(selected.children) != 0:
                selected = selected.nodeSelection()
            simVal = selected.simulation()
            selected.backup(simVal)
        return self.actionSelection()
```

Observamos que para jugar un turno en la partida real se simulan `MonteTree.simulationsPerTurn` partidas. En el bucle se ejecutan las 4 fases del árbol de búsqueda de Montecarlo especificadas en la sección 4.2 y al terminar este se selecciona una acción que es jugada en la partida real.

5.2. Aprendizaje de la red neuronal

La idea general para entrenar la red neuronal es la siguiente: dado que los movimientos que hemos jugado a lo largo de la partida han dependido en gran medida de ella, queremos reforzar esos movimientos si han llevado a la victoria y penalizarlos si han llevado a una derrota. Además, queremos reforzar/penalizar más un movimiento cuanto más cercano esté al final de la partida.

En cada movimiento i que jugamos guardamos el valor v_i que la red neuronal le da. La función de pérdida se parece a la definida en la sección 3.3.1, pero ajustando para reforzar más los movimientos más tardíos. Sea z el resultado final de la partida (1 en caso de victoria, 0 empate, -1 derrota), i el movimiento en el que se ha obtenido v_i y t el número de movimientos total:

$$l = c^{t-i}(v_i - z)^2 \tag{18}$$

donde $0 < c \leq 1$ es una constante que cuanto más grande más importancia da a los primeros movimientos de la partida. La red neuronal se va a entrenar al final de cada partida con todos los movimientos a la vez.

El código de la función pérdida es:

```
1 def loss_fn(preds: Tensor, result: int) -> Tensor:
2     l = len(preds)
3     realT = torch.tensor([result] * l)
4     importancePowers = [TicTacToeNN.importance ** i for i in range(l)]
5     importancePowers.reverse()
6     importanceT = torch.tensor(importancePowers)
7     errors = (preds - realT) ** 2 * importanceT
8     return errors.mean()
```

Aquí `preds` es un tensor de los valores de todos los movimientos de en una partida, `result` es el resultado de la partida y `TicTacToeNN.importance` es la constante c.

5.3. Tres en raya

Vamos a comenzar con un juego muy sencillo, el tres en raya, para después extrapolar el conocimiento adquirido al *backgammon*.

5.3.1. Reglas

El tres en raya es un juego de dos jugadores. Sobre un tablero 3x3, los jugadores toman turnos colocando una ficha en una casilla por turno. El juego termina por dos razones, o bien tres de las fichas de uno de los jugadores forman una línea recta (horizontal, vertical o diagonal), en cuyo caso dicho jugador gana, o bien el tablero se llena, en cuyo caso los jugadores empatan.

TABLERO

0.5736	0.2790	0.5403
0.4037	0.7111	0.2733
0.4696	0.3405	0.5313

TURNO

0.1369

Figura 4: Resultados tres en raya monocapa

5.3.2. Implementación con red monocapa

Dado que el tres en raya es un juego bastante sencillo, comenzaremos con una red de una sola capa, que tiene 10 valores de entrada: los 9 valores del tablero (valores $1,-1,0$; si la casilla tiene ficha de un jugador, del otro jugador o está vacía, respectivamente) y de quién es el turno (valor $1,-1$; si es el turno de un jugador o del otro, respectivamente), y un valor de salida: el valor de la posición.

Como queremos que los valores de salida de la red neuronal estén entre 1 y -1, pondremos un $tanh$ después de computar $x\,W^T$. Dado que se asocia un peso a cada casilla, no esperamos que la red neuronal sea capaz de interpretar patrones complejos. Las hipótesis son que al terminar el entrenamiento:

1. Todos los pesos serán positivos, ya que nunca es malo para un jugador tener una ficha suya en una casilla o que sea su turno.
2. Los pesos de las esquinas serán similares entre sí y los pesos de las aristas también, por simetría.
3. El peso de la casilla central será mayor que el de todas las demás, ya que en el tres en raya el centro es muy importante.

Tras jugar 2500 partidas, en las cuales para cada movimiento se han simulado 1000 posibles partidas, los pesos de la red neuronal han quedado tal y como vemos en la figura 4, cumpliendo las hipótesis.

No vamos a profundizar mucho más en los resultados de esta implementación, ya que sabemos que las redes neuronales de una sola capa están muy limitadas, pero hay un ajuste importante que hemos de hacer para las siguientes fases.

Viendo cómo juega el agente, he observado que el árbol de búsqueda de Montecarlo calcula correctamente los valores de las posiciones a las que puede mover. Sin embargo, por la manera en la que hemos definido la selección de acción (sección 5.1.4), muchas veces no efectúa el mejor movimiento; para ver por qué, veamos un ejemplo:

Digamos que es el turno del jugador 1 y el estado del tablero permite dos movimientos distintos. Tras hacer las pertinentes simulaciones, cuando se va a seleccionar la acción a tomar se tiene $Q_1 = 1$, $Q_2 = -1$, es decir, el primer movimiento siempre lleva a ganar y el segundo siempre a perder. Cuando computamos las probabilidades de seleccionar cada acción queda:

$$p_1 = \frac{e^{1\cdot 1}}{e^{1\cdot 1} + e^{1\cdot -1}} = 0{,}88 \quad (19)$$

$$p_2 = \frac{e^{1\cdot -1}}{e^{1\cdot 1} + e^{1\cdot -1}} = 0{,}12 \quad (20)$$

Podemos ver que, en esta situación, el agente tomará una decisión que lleva a una derrota certera un 12 % de las veces, aunque pueda ganar.

No seleccionar siempre la mejor acción puede ser bueno para la exploración, pero a la hora de jugar partidas reales daña mucho la capacidad del agente. Por ello, vamos a añadir flexibilidad a la fórmula de la siguiente manera:

$$p_i = \frac{e^{\mu\cdot x\cdot Q_i}}{\sum_{j=1}^{k} e^{\mu\cdot x\cdot Q_j}} \quad (21)$$

cuanto más grande hagamos μ más probabilidad tendrá el agente de efectuar el mejor movimiento.

5.3.3. Implementación con red multicapa

Para esta implementación añadimos 4 capas intermedias de 6 neuronas cada una y mantenemos la estructura de la capa de entrada y de salida iguales. Utilizaremos la función de activación *tanh*, definida en la sección 3.4.4. Para comprobar los resultados, veremos con cuánta distancia es capaz de ver movimientos ganadores o evitar derrotas el agente y cómo se comporta en partidas reales contra sí mismo y contra mí.

Para el entrenamiento vamos a establecer el parámetro $\mu = 1$ para permitir mucha exploración de movimientos no óptimos; pero cuando queramos comprobar si el agente se ha entrenado correctamente haremos $\mu \approx \infty$ para asegurar que escoge siempre la mejor acción.

A lo largo de todos los ejemplos, el agente será el jugador 1, es decir, el que coloca las X. Para cada caso vamos a mostrar la Q de cada casilla, ya que en eso se basa la decisión de que acción tomar. Las casillas están numeradas del 1 al 9, de izquierda a derecha y de arriba a abajo.

Se ha entrenado al agente en 5000 partidas, en las cuales para cada movimiento se han simulado 500 posibles partidas. Durante las pruebas en cada movimiento haremos 6000 simulaciones. Los resultados son los siguientes:

Victoria en un movimiento:

Estos son los casos más simples, en los que si el agente coloca la ficha en una casilla gana inmediatamente. Vamos a probar al agente con los casos de la figura 5.

Caso 1:
$Q_4 = 0{,}82$ $Q_6 = 0{,}97$ $Q_7 = 1$ $Q_8 = 0{,}35$ $Q_9 = 0{,}82$

Caso 2:
$Q_6 = -0{,}74$ $Q_7 = 0{,}19$ $Q_8 = 1$

Caso 3:
$Q_4 = 0{,}87$ $Q_5 = 1$ $Q_6 = -0{,}29$ $Q_7 = 0{,}98$ $Q_8 = 0{,}90$

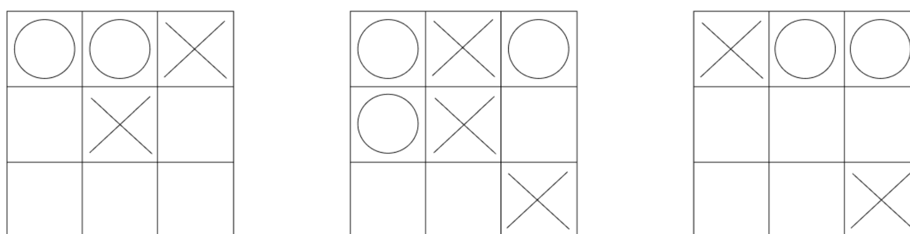

Figura 5: Casos de victoria en un movimiento

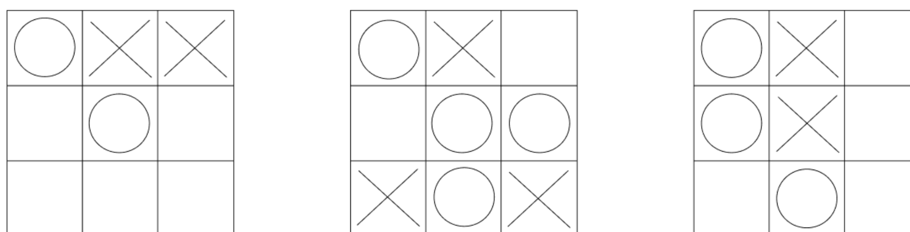

Figura 6: Casos de derrota en un movimiento

En los tres casos vemos que el agente ha posicionado como mejor la acción que gana y como $\mu \approx \infty$ esa es la acción que siempre va a tomar. También es interesante observar que en el caso 1 da una muy buena puntuación a poner la ficha en la sexta casilla, ya que esto también le llevará a una victoria si juega de manera óptima, sin importar lo que juegue el oponente. Esto también ocurre en la casillas 4, 7 y 8 del caso 3.

Derrota a un movimiento

Estos casos son ligeramente mas complejos: el agente ha de poner la ficha en una casilla para evitar que el oponente gane en el siguiente turno. Vamos a probarlo con los casos de la figura 6.

Caso 1:
$Q_4 = -0{,}67 \qquad Q_6 = -0{,}71 \qquad Q_7 = -0{,}77 \qquad Q_8 = -0{,}94 \qquad Q_9 = -0{,}01$

Caso 2:
$Q_3 = -1 \qquad Q_4 = 0$

Caso 3:
$Q_3 = -0{,}74 \qquad Q_6 = -0{,}73 \qquad Q_7 = 0{,}03 \qquad Q_9 = -0{,}78$

De nuevo, en los tres casos el agente ha colocado la ficha en la mejor posición. Una observación importante sobre la que volveremos más adelante es que el agente no da un valor tan negativo como el que debería a los movimientos que no evitan la derrota. En cualquiera de esos movimientos si el oponente juega de forma óptima siempre perderemos, por tanto esos movimientos deberían tener un valor más cercano a -1.

Victoria en dos movimientos

Para estos casos, el agente ha de efectuar una acción que lleve a la victoria (sin importar lo que juegue el oponente) en el siguiente movimiento. Vamos a probarlo con los casos de la figura 7.

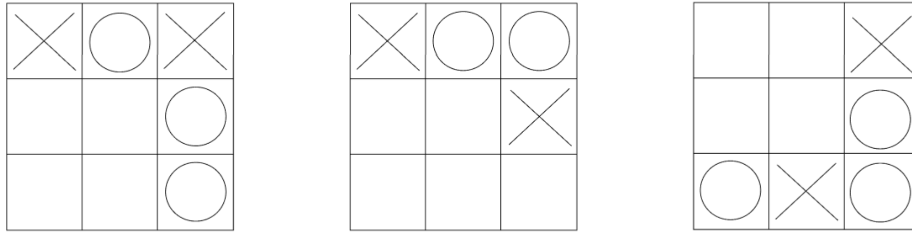

Figura 7: Casos de victoria en dos movimientos

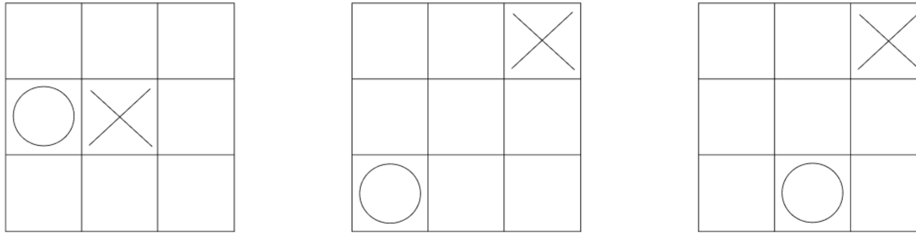

Figura 8: Casos de victoria en tres movimientos

Caso 1:
$Q_4 = -0{,}07$ $Q_5 = 0{,}07$ $Q_7 = 0{,}98$ $Q_8 = -0{,}32$

Caso 2:
$Q_4 = 0{,}93$ $Q_5 = 0{,}93$ $Q_7 = 0{,}79$ $Q_8 = 0{,}70$ $Q_9 = -0{,}44$

Caso 3:
$Q_1 = 0{,}31$ $Q_2 = 0{,}98$ $Q_4 = -0{,}21$ $Q_5 = 0{,}28$

El agente ha tomado la mejor acción en todos los casos. El caso 2 es interesante, porque hay dos posibles casillas que llevan a la victoria en el siguiente movimiento (la 4 y la 5) y el agente le ha dado un valor muy similar a ambas, como era de esperar.

Victoria en tres movimientos

En estos casos el agente ha de poner la ficha en una casilla que le lleve a ganar en tres movimientos. Probamos con los casos de la figura 8.

Caso 1:
$Q_1 = 0{,}72$ $Q_2 = 0{,}67$ $Q_3 = 0{,}29$ $Q_6 = 0{,}42$ $Q_7 = 0{,}85$ $Q_8 = 0{,}67$ $Q_9 = 0{,}65$

Caso 2:
$Q_1 = 0{,}17$ $Q_2 = 0{,}27$ $Q_4 = 0{,}31$ $Q_5 = 0{,}23$ $Q_6 = 0{,}19$ $Q_8 = -0{,}23$ $Q_9 = 0{,}87$

Caso 3:
$Q_1 = 0{,}44$ $Q_2 = 0{,}35$ $Q_4 = 0{,}00$ $Q_5 = 0{,}89$ $Q_6 = 0{,}13$ $Q_7 = 0{,}19$ $Q_9 = 0{,}77$

El agente a colocado la ficha en uno de los sitios correctos en los tres casos. Sin embargo, cabe notar que en el caso 1 todas las casillas salvo la 6 llevan a la victoria asegurada pero el agente ha dado menos puntuación a la casilla 3 que a la 6. En el caso 2, tanto la casilla 1 como la 9 llevan a la victoria, pero el agente les asigna valores muy dispares. Ocurre algo parecido en el caso 3, donde las casillas 1, 5 y 9 conducen a ganar y son las mejores valoradas por el agente pero obtienen valores demasiado distantes.

Reflexionando sobre esto y uniéndolo con la observación de la parte “Derrota en un movimiento”, he concluido que hay dos causas que conducen a estos problemas. La primera es que todos estos son escenarios artificiales. El agente jugando contra sí mismo casi nunca llega a estas posiciones y por tanto la red neuronal no está entrenada para valorarlas correctamente.

Pero aunque la red neuronal se hubiese entrenado con partidas mucho más diversas, hay otro problema: la manera en la que escogemos los movimientos en las simulaciones. Tal y como se observó al final de la sección 5.3.2, utilizar la función *softmax* es positivo durante el entrenamiento, ya que favorece la exploración de líneas de juego muy distintas, pero puede dañar la capacidad del agente al jugar partidas reales. Los Q se calculan como resultado de las simulaciones y las acciones en estas se escogen en función de P, que proviene de una función *softmax*.

Esto tiene una influencia más dramática cuanto mayor es la distancia a la victoria/derrota asegurada, ya que en las simulaciones se jugarán muchas acciones no óptimas por parte de ambos jugadores. Por ello, para el resto del trabajo, no vamos a escoger la acción a tomar en las simulaciones basándonos en P sino que vamos incluir otro coeficiente en la función que calcula las probabilidades:

$$p_i = \frac{e^{\alpha \cdot x \cdot v_i}}{\sum_{j=1}^{k} e^{\alpha \cdot x \cdot v_j}} \tag{22}$$

donde p_i es la probabilidad de tomar la acción i en una simulación y v_i es el valor que le da la red neuronal al hijo i-ésimo. Este cambio es igual al realizado en la sección 5.3.2, pero esta vez afectando a la manera en la que se efectúan las simulaciones.

Cabe notar que, al contrario que cuando introdujimos el coeficiente μ, no queremos $\alpha \approx \infty$ en las partidas reales, ya que queremos que se exploren partes del árbol y simulaciones que pueden no ser las mejores según la red neuronal. Para las partidas contra sí mismo y contra mí, vamos a establecer $\alpha = 4$, dado que tras algunas pruebas se ha visto que equilibra adecuadamente exploración y explotación.

Partidas

Para esta parte se ha reducido el número de simulaciones a 1000 por movimiento para que el agente juegue más rápido.

El agente ha jugado contra sí mismo 100 partidas, en las cuales el primer movimiento es aleatorio (si no las partidas son todas muy parecidas) y ha empatado todas ellas. También ha jugado otras 100 partidas contra un oponente que juega movimientos aleatorios, en todas las partidas ha empezado el oponente, y nuestro agente ha ganado 93 partidas, empatado 7 y no ha perdido ninguna.

Yo he jugado 10 partidas contra el agente, empezando yo en todas las partidas y buscando líneas de juego extrañas para intentar batirlo pero me ha ganado una y todas las demás han acabado en empate.

Por último, ha jugado 200 partidas contra el modelo con la red monocapa, empezando 100 veces cada uno. De las 100 que ha comenzado ha ganado 3, empatando todas las otras, y de las 100 que ha comenzado el agente monocapa, el multicapa ha perdido 1, empatando todas las demás.

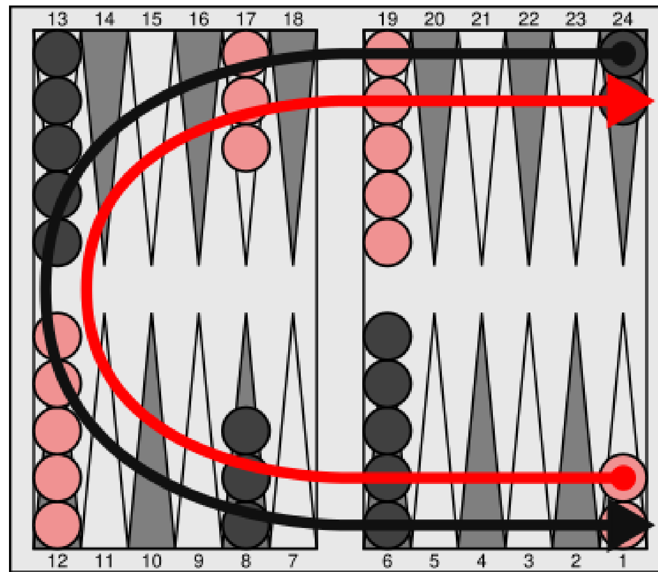

Figura 9: Tablero de *backgammon*, de [Wik]

5.4. *Backgammon*

5.4.1. Reglas

El *backgammon* es un juego de dos jugadores, que se juega en un tablero de 24 casillas. Cada jugador tiene 15 fichas, las de uno son de color rojo y las del otro de color negro. Para ganar, un jugador debe llevar todas sus fichas al final del tablero y sacarlas; los jugadores mueven en sentidos contrarios. Podemos ver un tablero con las casillas numeradas, las fichas, los sentidos en los que ambos jugadores mueven y la disposición al inicio de la partida en la figura 9.

En cada turno, un jugador tira dos dados y mueve dos veces, una por cada dado, una ficha el número casillas en la dirección indicada en la figura igual al número del dado. Esos dos movimientos pueden ser de la misma ficha. Hay un caso especial, en el que si ambos dados son el mismo número, el jugador obtiene cuatro movimientos con dicho número, en vez de dos. Para decidir quién empieza, al inicio de la partida cada jugador tira un dado y empieza el jugador con el número más alto, que juega su turno utilizando ambos dados.

Hay restricciones a la hora de mover una ficha. En la casilla de llegada de la ficha puede haber dos situaciones: que no haya fichas del oponente (es decir, hay fichas nuestras o la casilla esta vacía) o que haya una sola ficha del oponente (si hay más de una ficha del oponente en una casilla, la ficha del jugador no puede colocarse en esa casilla). En caso de haber una sola ficha del oponente, esa ficha se quita y se lleva al centro del tablero.

Si en el inicio de un turno un jugador tiene fichas de su color en el centro del tablero, debe quitarlas antes de seguir jugando. Para quitar una ficha del centro del tablero ha de usar sus movimientos, y pondrá la ficha en una de sus seis primeras casillas, dependiendo del número del dado. Es importante tener en cuenta que la restricción de movimiento de la que hemos hablado antes también se aplica al quitar las fichas del centro del tablero. En cada turno, los jugadores deben mover las fichas maximizando la cantidad de dados que usan.

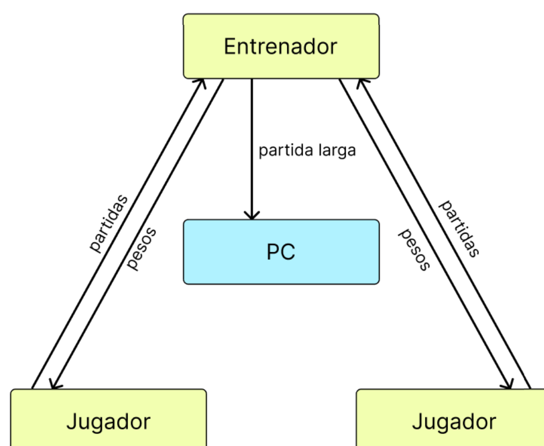

Figura 10: Infraestructura de implementación *backgammon*

Cuando todas las fichas de un jugador están en las 6 casillas más cercanas al final, el jugador puede empezar a sacar las fichas del tablero. Para hacerlo, debe mover las fichas la manera usual, pero como si hubiese una casilla más al final en la que las fichas acaban. Las fichas deben llegar a esa casilla de manera exacta para ser sacadas del tablero, salvo si la ficha que se esta introduciendo es la más alejada del final, entonces se puede sobrepasar (por ejemplo si la ficha más alejada del final del jugador rojo esta en la casilla 20, necesitaría un 5 para sacarla del tablero, pero con un 6 también puede sacarla).

El juego original es ligeramente más complejo: otorga a un jugador mayor puntuación si acaba mucho antes que el otro y permite apostar, pero para nuestro estudio vamos a utilizar esta simplificación.

5.4.2. Modificaciones en el árbol de búsqueda de Montecarlo

Dado que el *backgammon* es un juego probabilístico es necesario realizar algunas modificaciones sobre el árbol de búsqueda de Montecarlo planteado para el tres en raya. La estructura planteada se basa en la descrita en [VCU07].

En el árbol modificado hay dos tipos de nodo, con dados y sin dados. Los hijos de nodos sin dados son nodos con dados y al revés. Una arista de un nodo con dados a uno sin dados representa el turno de un jugador y se comporta en todos los sentidos igual que una arista del tres en raya. Una arista de un nodo sin dados a uno con representa simplemente una tirada de dados. Estas últimas no guardan ninguna información aparte de la tirada de dados, ya que para pasar de un nodo sin dados a uno con no hay que tomar ninguna decisión, solo hay que tirar dados.

Este cambio afecta sobretodo a la etapa de elección, para pasar por una arista de un nodo sin dados a uno con dados se tiran dos dados y se toma el nodo con dichos dados o se crear si no existe. El cambio no afecta ni a la etapa de simulación ni a la de expansión, y en la de actualización simplemente ignoramos las aristas de nodos sin dados a nodos con dados.

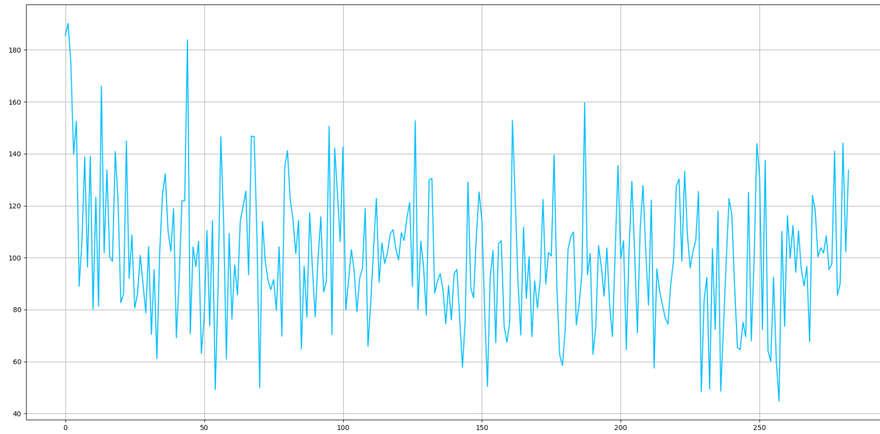

Figura 11: Primera red, evaluaciones de *gnubg*

5.4.3. Infraestructura de implementación

El *backgammon* es un juego mucho mas complejo que el tres en raya, se estima que hay unos 10^{20} estados posibles (ver [Tes95]), las partidas suelen durar entre 60 y 80 movimientos y el cálculo de los movimientos posibles en un estado es complejo. Por ello, tanto la cantidad de partidas necesarias para entrenar la red neuronal como la duración de dichas partidas es muy superior a las del tres en raya.

Para poder producir un número de partidas suficientemente grande, he montado una pequeña infraestructura en la nube, figura 10. Los nodos amarillos son instancias en la nube y el nodo azul es mi ordenador.

Los nodos jugadores juegan partidas en bucle, y mandan las partidas al nodo entrenador. El nodo entrenador entrena la red neuronal y devuelve los pesos actualizados a los nodos jugadores. Además, el nodo entrenador juega partidas más largas que las que juegan los nodos jugadores (número de simulaciones en el árbol y profundidad de estas más elevado), que son las que utilizamos para evaluar si el agente está mejorando.

Para evaluar cómo juega el agente, se analiza cada partida utilizando el motor *gnubg* ([GNU]), el cual proporciona una media de los errores cometidos en cada movimiento no forzado. A continuación se presentan los resultados de entrenar el agente con la misma estructura de árbol de búsqueda de Montecarlo pero distintas redes neuronales.

5.4.4. Resultados primera red

Esta red recibe 29 valores de entrada: el número de fichas en cada una de las 24 casillas del tablero (número positivo si son fichas de un jugador, negativo si son del otro), el número de fichas que cada jugador tiene en el centro del tablero, las que tienen fuera del tablero y de quién es el turno. También cuenta con tres capas intermedias con 64 neuronas cada una, y una neurona de salida. La función de activación utilizada es *tanh*.

Al probar esta red neuronal y dibujar un gráfico de las evaluaciones proporcionadas por *gnubg* sobre las partidas jugadas por el nodo entrenador, se ha observado una ligera mejora al principio seguida por una comportamiento muy lateral, como podemos observar en la figura 11.

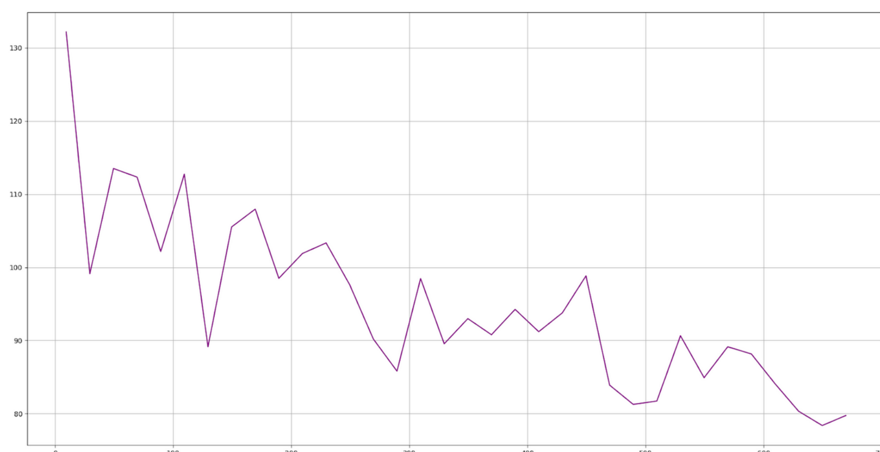

Figura 12: Segunda red, evaluaciones de *gnubg* (partidas agrupadas de 20 en 20)

Pese a ello, al probar a evaluar ciertas posiciones con la red hay un aprendizaje visible, ya que para posiciones muy claramente ganadoras les da un valor positivo, y a las muy claramente perdedoras uno negativo, por tanto he teorizado que la red neuronal es capaz de entender patrones evidentes, pero quizá sea demasiado pequeña para aprender estrategias complejas, lo cual es necesario para jugar al *backgammon*.

5.4.5. Resultados segunda red

Esta segunda red mantiene la estructura de entrada y salida de la primera, pero en vez de 3 capas intermedias tiene 5, la primera de 128 neuronas y las demás de 64.

Observando el gráfico de las evaluaciones proporcionadas por *gnubg* (figura 12) es evidente que esta red ha tenido más exito, ya que la tendencia decreciente del error cometido es clara. Estos resultados provienen de 3 días de entrenamiento, a lo largo de los cuales la red neuronal ha sido entrenada con unas 9000 partidas.

En el gráfico podemos observar que el error cometido esta entorno a 80. Yo que soy un jugador casual he jugado 10 partidas contra mí y he obtenido evaluaciones entre 70 y 110, por tanto el agente que hemos entrenado tiene un nivel similar al de un jugador casual.

6. Conclusión

Concluimos que utilizando árboles de búsqueda de Montecarlo y redes neuronales se puede conseguir un agente que juegue bien al *backgammon*.

Para mejorar la capacidad de este agente se puede usar una red neuronal más grande, entrenarlo con más partidas o incrementar el número de simulaciones y la profundidad de dichas simulaciones en el árbol de búsqueda de Montecarlo.

También podrían introducirse cambios cualitativos en la red neuronal que quizá mejorasen la capacidad del agente, como usar una estructura de entrada más parecida a la de *AlphaZero*, lo cual consistiría en introducir un vector para las fichas de cada jugador, en vez de un tablero conjunto como se hace ahora. Otro aspecto distinto de *AlphaZero* es que utiliza capas convolucionales bidimensionales, que no tienen sentido en el *backgammon*, pero podrían introducirse capas convolucionales unidimensionales.

Conclusion

We conclude that using Montecarlo tree search and neural networks, an agent which plays *backgammon* well can be obtained.

To improve the ability of the agent, a bigger neural network could be used. We could also train it with more games or increment the amount of simulations and their depth in the Monte Carlo tree search.

Qualitative changes could also be added into the neural network, which may improve the ability of the agent, such as using a input structure more similar to *AlphaZero*, which would involve introducing a vector for the pieces of each player, instead of a joint board. Another aspect which is different in *AlphaZero* is the use of bidimensional convolutional layers, which don't make sense in *backgammon*, however unidimensional convolutional layers could be added.

Referencias

[Tes95] Gerald Tesauro. «Temporal difference learning and TD-Gammon». En: *Commun. ACM* 38.3 (mar. de 1995), págs. 58-68. URL: https://doi.org/10.1145/203330.203343.

[VCU07] François Van Lishout, Guillaume Chaslot y Jos Uiterwijk. «Monte-Carlo tree search in backgammon». En: *Computer Games Workshop* (ene. de 2007).

[Sil+16] D. Silver, A. Huang, C. Maddison et al. «Mastering the game of Go with deep neural networks and tree search». En: *Nature 529* (2016), págs. 484-489. URL: https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf.

[GK20] Laura Graesser y Wah Loon Keng. *Foundations of Deep Reinforcement Learning*. Addison-Wesley, 2020.

[SB20] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning*. Westchester Publishing Services, 2020.

[Mur24] Kevin P. Murphy. *Reinforcement learning: An Overview*. 2024.

[Alb] Alberto Maurel. *Aprendizaje por refuerzo: Fundamentos teóricos del algoritmo AlphaZero e implementación (Trabajo de Fin de Grado)*.

[Arp] Miguel de Arpe Renard. *Código en github*. URL: https://github.com/migueldar/tfgInformatica.

[GNU] GNU programmers. *GNU backgammon*. URL: https://www.gnu.org/software/gnubg/.

[Tho] Tommy Thompson. *AI 101: Monte Carlo Tree Search*. URL: https://www.youtube.com/watch?v=lhFXKNyAOQA.

[Wik] Wikipedia contributors. *Backgammon — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Backgammon.

[Zha+] A. Zhang, Zachary C. Lipton, Mu Li et al. *Dive into deep learning*. URL: https://d2l.ai/index.html.