Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aprendizaje por refuerzo para agentes competitivos en el juego del Risk

Autor: Alberto Vicente del Egido

Tutor: David Fernandez Barrero

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aprendizaje por refuerzo para agentes competitivos en el juego del Risk

Autor: Alberto Vicente del Egido

Tutor: David Fernandez Barrero

Tribunal:

Presidente: Julia Maria Clemente Parraga

Vocal 1º: Concepcion Batanero Ochaíta

Vocal 2º: David Fernández Barrero

Fecha de depósito: 14 de Septiembre de 2022

A todos los que han creído en mi desde el principio...

"El cambio es siempre el resultado final de todo verdadero aprendizaje"
Leo Buscaglia

Agradecimientos

Last but not least, I want to thank me.

Snoop Dogg

Este proyecto es la culminación de una etapa muy importante de mi vida, de años de interés, estudio, errores y aciertos, un cierre ideal que no habría sido posible sin mis padres que siempre apostaron por mi, sin la fe que mi hermana ha depositado sobre mi desde que tengo memoria, sin las tardes en Discord que hemos pasado los chavales enseñándonos algoritmos que ninguno sabíamos como funcionaban, sin el Dr. Calvo y su inconmensurable conocimiento de las redes neuronales, sin mis Templarios intentando ganar a mi agente a las Tres en Raya, sin Elena intentando entender(sin ningún tipo de éxito) los tres capítulos que le obligué a leer y sin Paula, la persona que mas ha sufrido los desajustes que ha ocasionado este proyecto en mi vida.

Por todo lo mencionado y todo lo que no hay tiempo de contar, este proyecto va dedicado a ellos, a todos los que han creído en mi desde el principio.

Gracias a todos.

Resumen

En este TFG se plantea la posibilidad de diseñar de cero un agente inteligente que mediante aprendizaje reforzado profundo aprenda a jugar de manera eficiente al juego de mesa de estrategia militar Risk. En primer lugar se diseñara un agente que mediante Q learning aprenda a jugar al Tres en Raya, a continuación uno que resuelva la misma tarea mediante Deep Q learning y mas adelante se adaptará este agente para que aprenda el Risk, tanto una versión reducida de este, como el original.

Palabras clave: Machine Learning, Deep Learning, Reinforcement Learning, Q Learning, Deep Q Learning, Risk, Board Games, Tic Tac Toe.

Abstract

In this TFG we propose the possibility of designing from scratch an intelligent agent that by deep reinforcement learning learns to play efficiently the military strategy board game Risk. Firstly, a Q Learning agent will be designed to learn how to play Tic-Tac-Toe, then one that solves the same task by Deep Q learning, and finally this agent will be adapted to learn Risk, a reduced version of it and the original one.

Keywords: Machine Learning, Deep Learning, Reinforcement Learning, Q Learning, Deep Q Learning, Risk, Board Games, Tic Tac Toe.

Índice general

Resumen **ix**

Abstract **xi**

Índice general **xiii**

Índice de figuras **xvii**

Índice de tablas **xix**

Índice de listados de código fuente **xxi**

Índice de algoritmos **xxiii**

Lista de acrónimos **xxiii**

Lista de símbolos **xxiii**

1 Introducción **1**

1.1 Presentación 1

1.2 Risk: El arte de la guerra 1

1.2.1 Juegos de estrategia 1

1.2.2 Historia 1

1.2.3 Reglas y Dinámica de Juego 2

1.2.3.1 Elementos de Juego 2

1.2.3.2 Preparación 2

1.2.3.3 Turno 3

1.2.3.4 Victoria 4

1.2.4 Reglas para 2 jugadores 4

1.3 IA y Risk: Motivaciones 4

1.4 Objetivos 4

1.5 Descripción del trabajo 5

2 Marco Teórico **7**

2.1 Inteligencia Artificial . . . 7

2.2 Redes neuronales: Una revolución en dos tiempos . . . 8

2.2.1 El Perceptrón: La primera red neuronal . . . 8

2.2.1.1 Ejecución Algoritmo Perceptrón . . . 9

2.2.1.2 Demasiado simple . . . 10

2.2.2 Redes neuronales profundas . . . 10

2.3 Aprendizaje por Refuerzo: Una aproximación mas humana . . . 12

2.3.1 Procesos de Decisión de Markov . . . 13

2.3.1.1 Elementos implicados . . . 13

2.3.1.2 Caso de ejemplo . . . 14

2.3.1.3 Recompensas de Markov . . . 14

2.3.1.4 Ecuación de Bellman: Distancia a la recompensa . . . 15

2.3.2 Q Learning . . . 15

2.3.3 Deep Q Learning . . . 18

2.4 Agentes de Risk . . . 20

3 Desarrollo y entrenamiento de los agentes **23**

3.1 Introducción . . . 23

3.2 Implementación del Agente Q-Learning . . . 23

3.2.0.1 Hiperparametros y Q inicial . . . 24

3.2.0.2 Variables de instancia . . . 24

3.2.0.3 Política Aleatoria . . . 24

3.2.0.4 Política Probabilística . . . 25

3.2.0.5 Política Voraz . . . 25

3.2.0.6 Actualización de la Q-Table . . . 26

3.2.0.7 Serializado y Carga . . . 29

3.3 Implementación del Agente Deep Q-Learning . . . 30

3.3.0.1 Hiperparametros . . . 30

3.3.0.2 Variables de instancia . . . 30

3.3.0.3 Política Aleatoria . . . 31

3.3.0.4 Política Probabilística . . . 31

3.3.0.5 Política Voraz . . . 33

3.3.0.6 Entrenamiento de las Redes Q Profundas (DQN) . . . 33

3.3.0.7 Serializado y Carga . . . 37

3.4 Resolución Tres en Raya . . . 37

3.4.1 Configuración del entorno . . . 37

3.4.2 Configuración del agente en el entorno . . . 38
3.4.2.1 Diseño de las Redes . . . 38
3.4.2.2 Limitaciones impuestas a los agentes . . . 39
3.4.3 Diseño del experimento . . . 40
3.5 Resolución Risk Simple . . . 40
3.5.1 Configuración del entorno . . . 40
3.5.2 Configuración del agente en el entorno . . . 41
3.5.2.1 Diseño de las Redes . . . 41
3.5.3 Diseño del experimento . . . 43
3.6 Resolución Risk . . . 44
3.6.1 Configuración del entorno . . . 44
3.6.2 Configuración del agente en el entorno . . . 44
3.6.2.1 Diseño de las Redes . . . 44
3.6.3 Diseño del experimento . . . 44

4 Resultados **47**
4.1 Introducción . . . 47
4.2 Tres en raya Q Learning . . . 47
4.3 Tres en raya Deep Q Learning . . . 49
4.3.1 Q Learning vs Deep Q Learning . . . 53
4.3.2 Diseño y programación desde cero de los agentes . . . 53
4.4 Risk Simple Deep Q Learning . . . 54
4.5 Risk Completo . . . 58

5 Conclusiones y lÃneas futuras **61**

Bibliografía **63**

Índice de figuras

1.1 Tablero del Risk 2
1.2 Tipos de tropa 3

2.1 Perceptrón Simple 9
2.2 Feed Forward 10
2.3 Convolucional 11
2.4 Long Short Term Memory 11
2.5 Generativa Adversa 12
2.6 Relación Agente Entorno 13
2.7 Grafo MDP 14
2.8 Tablero Q Learning 17
2.9 Espacio de salida 19
2.10 Arquitectura Agente Deep Q Learning 20

3.1 Tablero Tres en Raya 38
3.2 Arquitectura de red 39
3.3 Monitorización entrenamiento 40
3.4 Tablero Risk simple 41
3.5 Arquitectura de red tropas 42
3.6 Arquitectura de red ataque 43
3.7 Tablero Risk 44
3.8 Arquitectura de red tropas 45
3.9 Arquitectura de red ataque 45

4.1 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 1 48
4.2 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 2 49
4.3 Arquitectura de red del agente 1 50
4.4 Arquitectura de red del agente 2 51
4.5 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 1. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O 52

4.6 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 2. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O 52

4.7 Recompensa media por partida en cada *checkpoint* de entrenamiento enfrentando dos agentes entrenados juntos. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O 53

4.8 Partida completa entre dos agentes 53

4.9 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente con 16 neuronas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 54

4.10 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente con 24 neuronas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 54

4.11 Arquitectura de red tropas 55

4.12 Arquitectura de red ataque 55

4.13 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de tropas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 56

4.14 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de ataque. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 56

4.15 Captura del agente aprovechando el error de programación 57

4.16 Arquitectrua de red de tropas 58

4.17 Arquitectrua de red de ataque 59

4.18 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de tropas del Risk completo. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 . 60

4.19 Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de ataque del Risk completo. Donde la linea naranja representa el jugador 1 y la azul el jugador 2 . 60

Índice de tablas

1.1 Bonificación por territorio 3
1.2 Bonificación por cartas 3
1.3 Bonificación por continente 3

4.1 Factor de Descuento 48
4.2 Factor de Exploración 48
4.3 Hiperparámetros fijos copiados del agente de Q Learning con mejor rendimiento 49

Índice de listados de código fuente

Índice de algoritmos

Capítulo 1

Introducción

Robots are not going to replace humans, they are going to make their jobs much more humane. Difficult, demeaning, demanding, dangerous, dull – these are the jobs robots will be taking

Sabine Hauert, Co-founder of Robohub.org

1.1 Presentación

En este trabajo se pretende aprender la teoría detrás del Aprendizaje Reforzado, y mas en concreto Aprendizaje Reforzado Profundo, para crear un agente que aprenda a jugar al Risk mediante este.

1.2 Risk: El arte de la guerra

1.2.1 Juegos de estrategia

Los juegos de estrategia se diseñan para que el factor inteligencia, las habilidades de planificación y despliegue impulsen al jugador que las posee y utiliza correctamente hacia la victoria. Como cabía esperar la gran mayoría de estos juegos son de guerra, pues es una temática que encaja a la perfección con las mecánicas de este género entre las que destacan la gestión de recursos, la planificación de los turnos y sencillos sistemas de reglas para resolver los combates.

1.2.2 Historia

El juego de mesa de estrategia Risk aparece en el año 1950, creado por el director de cine frances Albert Lamorisse[1]. Este juego se lanza al mercado en 1957 con el nombre La Conquête Du Monde[2], sin embargo no tuvo un gran recibimiento entre el publico, ya sea por que necesitaba abundantes modificaciones, pues el juego dependía en gran parte del azar, o porque aun se estaban apagando las ultimas ascuas de la Segunda Guerra Mundial y el nombre no era muy apropiado en aquel contexto.

[1] Ganador de varios premios internacionales por sus cortometrajes, su trabajo mas conocido se Le Ballon Rouge
[2] La Conquista del Mundo

Años después, en 1959 una empresa de juegos de mesa americana llamada Parker Brothers, tras comprar los derechos de La Conquête Du Monde y modificar las reglas y dinámica de este, lanzaría al mercado en Estados Unidos el Risk tal y como lo conocemos hoy en dia.

1.2.3 Reglas y Dinámica de Juego

Este juego no tiene unas premisas muy ambiciosas ni complejas, de hecho su sencillez y accesibilidad es uno de sus puntos fuertes. A grandes rasgos, solo debes colocar tus tropas en un territorio para controlarlo y el jugador que controle todos los territorios gana la partida y un turno consiste en colocar tropas, atacar y defender territorios.

1.2.3.1 Elementos de Juego

- Tablero

  Dividido en 42 territorios, conectados entre si por una frontera colindante o por una vía marítima. Estos se agrupan en seis continentes, Europa, Asia, Oceanía, América del Norte, América del Sur y África.

Figura 1.1

- Dados

  Hay tres dados de ataque y dos de defensa.

- Cartas

  Las cartas de territorio muestran uno de estos y una o dos estrellas y la carta de alto el fuego determina el final de la partida.

- Tropas

  Las tropas básicas son los soldados de infantería, cinco de estos son intercambiables por un peón de Caballería y dos caballos por un peón de Artillería.

1.2.3.2 Preparación

1. Barajar y repartir las cartas de territorio entre los jugadores.

Figura 1.2

2. Por cada carta de 1 estrella el jugador que la tiene coloca una tropa en el territorio de esta, por cada carta de 2 estrellas coloca 2 tropas.

3. Se recogen todas las cartas, se barajan de nuevo y se dejan cerca del tablero.

1.2.3.3 Turno

1. Colocar tropas

   Al principio del turno se recibe un numero fijo de 3 tropas para colocar en los territorios ocupados, además de posibles tropas de bonificación por: Territorios, cartas o continentes

Tabla 1.1: Bonificación por territorio

Territorios	12-14	15-17	18-20	21-23	24-26	27-29	30-32	33-35	36-39	40-42
Tropas	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10

Tabla 1.2: Bonificación por cartas

Estrellas	2	3	4	5	6	7	8	9	10
Tropas	+2	+4	+7	+10	+13	+17	+21	+25	+30

Tabla 1.3: Bonificación por continente

Continentes	Europa	Asia	Oceanía	América del Norte	América del Sur	África
Tropas	+5	+7	+2	+5	+2	+3

2. Ataque

   Una vez se han colocado las tropas se puede decidir si atacar o no. En caso afirmativo se elige un territorio ocupado desde el que efectuar el ataque, uno conectado a este al que dirigir el ataque y el numero de tropas a usar (Siendo estas siempre menor que el total de tropas del territorio desde el que se ataca).

   El propietario del territorio atacado puede defenderse con 1 o 2 tropas. Ambos jugadores lanzan sus dados (Hasta 3 según las tropas usadas para el atacante y hasta 2 para el defensor)

   El combate se resuelve comparando los dados de ataque y defensa de mayor a menor por parejas, el perdedor de cada comparación pierde una tropa y en caso de empate la pierde el atacante. Cuando el territorio atacado pierde todas sus tropas, este es transferido al atacante.

   Tanto si se gana o pierde la batalla se puede atacar de nuevo si se desea, tantas veces como se quiera.

   Un jugador que pierde todos sus territorios queda eliminado.

3. Fortificar territorios

   Una vez terminados los ataques, se pueden mover tropas entre territorios conectados y si se ha conquistado algún territorio en este turno se recibe una carta.

1.2.3.4 Victoria

Gana el ultimo jugador sin eliminar.

1.2.4 Reglas para 2 jugadores

Cuando solo hay dos jugadores, se usa un tercer ejercito neutral que no coloca tropas, no ataca y no fortifica, es decir, solo puede defenderse. El resto de las normas y reglas son exactamente las mismas. Esta es la modalidad de juego que se usará en este proyecto.

1.3 IA y Risk: Motivaciones

El primer juego de mesa data de hace 5.000 años en la antigua Roma, donde los romanos pasaban parte de su tiempo de ocio lanzando una y otra vez los dados. Desde entonces al igual que los humanos, los juegos de mesa han evolucionado notablemente en cuanto a complejidad, por lo que la resolución de estos es un reto muy interesante.

Se han diseñado numerosos algoritmos de carácter general para resolver muchos de estos juegos como: el póker, las damas, scrabble, monopoly, el ajedrez e incluso go.

El primer gran avance en este ámbito fue en 1996, cuando una Inteligencia Artificial ganó al en aquel entonces campeón del mundo de ajedrez y uno de los ajedrecistas mas famosos de la historia Garry Kasparov. Desde entonces la mejoría de estos algoritmos y los logros que estos han conseguido ha alcanzado incluso al GO, el considerado juego de mesa más difícil del mundo.

La parte mas interesante de la resolución de juegos de mesa mediante la IA es que permite demostrar de forma muy accesible las capacidades de aprendizaje que esta posee. No hace falta ser un gran entendido de la materia para ver lo significativo que es que un ordenador aprenda a jugar al ajedrez mejor que el mejor ajedrecista de la historia y resulta realmente estimulante e interesante ver las nuevas estrategias de juego que se descubren gracias a esto.

Por ello, en este proyecto se quiere diseñar y desarrollar un agente inteligente que resuelva el Risk, que es un juego muy interesante, divertido y que en la sencillez de sus reglas esconde una gran variedad y complejidad de jugadas, estrategias y enfoques posibles

1.4 Objetivos

Para este trabajo de fin de grado el objetivo principal es resolver el juego del Risk mediante aprendizaje reforzado profundo. Este es un objetivo demasiado complejo como para tomarlo como una única meta, pues antes de pretender resolver un problema tan complejo, primero debemos investigar acerca de las posibles aproximaciones que pueden usarse en este contexto, y tras decidirnos por una o varias estas deben ser implementadas y probadas con un problema relacionado más sencillo para verificar su desempeño, para posteriormente enfrentarse al Risk

Por ello dividimos el objetivo principal en los siguientes objetivos específicos:

- Realizar un estudio del estado del arte del Aprendizaje Reforzado, Q learning y Deep Q learning en entornos multiagente por turnos
- Implementar un algoritmo de Q learning
- Implementar un algoritmo de Deep Q learning
- Resolver un entorno multiagente por turnos más simple que el Risk
- Resolver una versión simplificada del Risk
- Resolver el Risk

1.5 Descripción del trabajo

Una de las cualidades más interesantes de este trabajo es que partimos prácticamente de cero, es decir, se carece de experiencia previa en aprendizaje reforzado, por lo que antes de sumergirse de lleno en la ejecución de este proyecto se realizara una investigación para comprender las cualidades, fortalezas, debilidades y entresijos del aprendizaje reforzado mediante Q learning y Q learning profundo.

Una vez asimilada la parte teórica e implementados agentes con la capacidad de aprender mediante las dos variantes del aprendizaje reforzado estudiadas, se probará la efectividad y estudiará el rendimiento de estos en un juego más sencillo que el Risk, el Tic Tac Toe o tres en raya, pues empezar directamente con un problema de las dimensiones del Risk es precipitado y tras resolver este sencillo juego, se pasará por el mismo proceso con una o varias versiones simplificadas del Risk, como por ejemplo reducir el número de territorios, continentes o jugadores, para en último lugar, proceder a resolver el Risk completo.

Como es de esperar, en cada uno de los juegos resueltos, se documentará, justificara y extraerán conclusiones del proceso.

Capítulo 2

Marco Teórico

No es lo inteligente que eres lo que importa, lo que realmente cuenta es cómo es tu inteligencia.

Howard Gardner

2.1 Inteligencia Artificial

Definir que es la Inteligencia Artificial es en principio una tarea simple, se puede decir que es una disciplina que intenta replicar y desarrollar inteligencia a través de computación o se puede ver como el conjunto y combinación de algoritmos que buscan crear maquinas con inteligencia humana.

El problema de estas definiciones es que nos llevan una pregunta mas compleja, ¿que es esa inteligencia que la IA pretende imitar?

Entre las definiciones que la Real Academia Española de la Lengua (RAE) propone para *inteligencia* [1] se encuentran:

- Capacidad de entender o comprender.
- Capacidad de resolver problemas.
- Conocimiento, comprensión o acto de entender.

Estas definiciones son demasiado elementales, muy simples, por lo que tenemos que ir mas allá para comprender bien este concepto. Desde un punto de vista biológico, la inteligencia es la capacidad que nos permite adaptarnos y solventar nuevas situaciones para sobrevivir. Sin embargo esta capacidad innata del ser humano es sensible a la educación, la experiencia, las emociones y el entorno social del individuos, son muchos los factores que la potencian o debilitan y por ello todos deben tomarse en cuenta.

Cuanto mas profundizamos en nuestra búsqueda de una definición, mas densa se vuelve, sin embargo hemos pasado por alto un detalle, solo los humanos poseen estas habilidades, la inteligencia es una cualidad única de nuestra especie, es la capacidad de adquirir conocimientos y el uso que hacemos de ellos, y sabiendo esto podemos definir la Inteligencia Artificial (IA) como la disciplina que busca imitar esta forma de adquirir y usar conocimientos tan humana.

No se puede hablar de la IA sin hablar de Alan Mathison Turing, nacido en Londres en 1912, que fue un matemático, informático teórico y criptógrafo que se quitó la vida en 1954 tras ser castrado químicamente por orden judicial debido a su homosexualidad. Turing, en su artículo *Computing machinery and intelligence* [2], fue el primero en preguntarse si las maquinas podían pensar. Este artículo pasa a la historia porque en el se propone el famoso test de Turing [1], una prueba que muestra la capacidad de un ordenador para exhibir comportamiento inteligente considerado humano, o como ya hemos dicho antes imitar el comportamiento humano.

Este test implica una maquina A, una persona B y un interrogador C que tiene que descubrir cual de los otros dos implicados es una máquina. Para esto puede hacer cualquier pregunta a estos y recibirá una respuesta escrita a máquina. La hipótesis de Turing es que si la maquina consigue engañar al interrogador, se puede considerar inteligente. Sin embargo esta prueba no puede medir la inteligencia de una maquina, solo su capacidad para imitar la forma de conversar de los humanos.

2.2 Redes neuronales: Una revolución en dos tiempos

Las redes neuronales son modelos matemáticos y computacionales del funcionamiento del sistema nervioso. Un modelo sencillo que imita la forma en la que el cerebro humano procesa la información, mediante un elevado numero de unidades de procesamiento individuales conectadas, las neuronas.

Pero antes de definir en profundidad las redes neuronales, debemos empezar por sus orígenes, el Perceptrón.

2.2.1 El Perceptrón: La primera red neuronal

El Perceptrón simple es la primera red neuronal de la historia. Este algoritmo ve la luz en 1965 [3], diseñado por el psicólogo Frank Rosenblatt , como un clasificador binario o discriminador lineal. Es decir, este Perceptrón aprendia a clasificiar o discriminar datos a partir de un entrenamiento.

El Perceptrón esta formado por capas de neuronas, en este caso dos, la capa de entrada y la de salida. Cada neurona es una unidad de procesamiento que realiza la siguiente operación: $f(x) = fun(wx + b)$, donde a w se le llama peso , a b bias y fun es una función de activación lineal como hardlimit o relu. La capa de entrada esta formada por n neuronas cuyas x son las variables de entrada del modelo y la capa de salida esta formada por m neuronas cuyas x son la suma de todas las $f(x)$ de la capa anterior.

Durante el proceso de entrenamiento se tienen datos de entrada (X) y las salidas esperadas de cada uno de estos (Y), los cuales son utilizados para entrenar el modelo siguiendo estos pasos:

1. Se introduce el dato X_i y se calcula la salida O_i
2. Se calcula el error usando la salida generada O_i y la esperada Y_i con la formula: $e_i = Y_i - O_i$
3. Se propaga el error a las neuronas y se modifican los pesos en función de este con la formula: $w_{i+1} = w_i + e_i X_i$

[1] Inicialmente y mas acertadamente conocido como Juego de la Imitación

Veamos un ejemplo con los siguientes elementos:

- Perceptrón Simple con dos neuronas de entrada, una de salida y función de activación $f(x) = x$.
  $w = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ $b = 0$

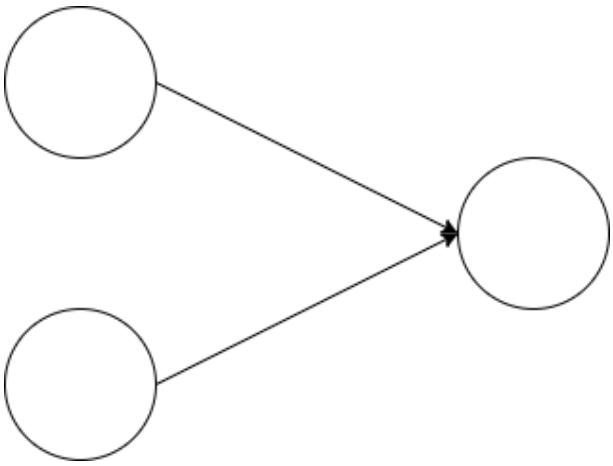

Figura 2.1: Perceptrón Simple

- Datos de entrada X $\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$
- Datos de salida Y $\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$

2.2.1.1 Ejecución Algoritmo Perceptrón

- Primera iteración:
  1. Salida: $O_0 = w^T X_0 + b_0 = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0 = 2$
  2. Error: $e_0 = Y_0 - O_0 = 1 - 2 = -1$
  3. Propagación y reajuste de pesos: $w_1 = w_0 + e_0 X_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
  4. Propagación y reajuste de bias: $b_1 = b_0 + e_0 = 0 - 1 = -1$
- Segunda iteración:
  1. Salida: $O_1 = w^T X_1 + b_1 = \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} - 1 = -1$
  2. Error: $e_1 = Y_1 - O_1 = 0 - (-1) = 1$
  3. Propagación y reajuste de pesos: $w_2 = w_1 + e_1 X_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
  4. Propagación y reajuste de bias: $b_2 = b_1 + e_1 = -1 + 1 = 0$
- Enésima iteración:...

  Se pueden ejecutar tantas iteraciones como sea necesario hasta que el modelo clasifique correctamente

2.2.1.2 Demasiado simple

Tras la implementación de este algoritmo en 1957, se vio que la capacidad de aprendizaje de este modelo era prácticamente nula, pues no puede reconocer patrones trivialmente complejos debido a que solo puede clasificar datos con una distribución lineal.

No sería hasta 1969, cuando Minsky y Papert demuestran que el Perceptrón simple no puede clasificar problemas no lineales (como la puerta lógica XOR) [4], que se comienza a buscar la manera de aumentar la capacidad de aprendizaje de este modelo, sugiriendo que la concatenación de Perceptrones simples, el Perceptrón multicapa, si podria resolver problemas de clasificación no lineal. Sin embargo no se conocían maneras de propagar los errores a las capas ocultas de este y aunque en 1986 Rumelhart y otros autores presentan la Regla Delta Generalizada"[5] que permite esta propagación a la capa oculta y el uso de funciones de activación no lineales, la tecnología no permitiría hasta años mas tarde que estos imitadores, clasificadores o aproximadores universales viesen mundo mas allá del teórico.

2.2.2 Redes neuronales profundas

El ordenador mas potente hasta la fecha cuando se teoriza por primera vez sobre las redes neuronales ejecutaba hasta 40.000 instrucciones por segundo y hoy en día un Intel Core i7, un procesador de gama media domestico, ejecuta hasta 100.000 Millones de Operaciones por Segundo (MIPS).

La capacidad de computo ha avanzado muy rápido en las ultimas décadas, permitiendo que a principios del siglo XXI se entrenasen las primeras redes neuronales profundas. Redes con arquitecturas cada vez mas complejas y profundas, un nuevo campo completamente abierto de la computación donde desde entonces se han logrado importantes avances.

Como es de esperar, la palabra red neuronal profunda es un hiperónimo , es decir, desde sus inicios se han diseñado muchos tipos distintos de redes neuronales entre las que vamos a destacar:

- Red Neuronal Feed Forward (NN)

  Estas redes obtienen su nombre de su mecánica principal, las salidas de una capa se usan como entradas de la siguiente, es decir, la red se alimenta hacia delante, generalmente son *fully conected Feed Forward Networks*, es decir, cada salida de una capa esta conectada a todas las entradas de la siguiente.

  Estas redes son la base de todas las demás, es el punto de partida, y la mayor parte de las veces se acoplan a la salida de los otros tipos de arquitectura.

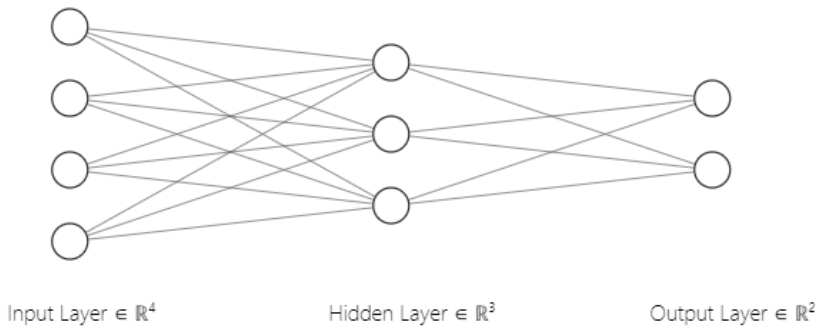

Figura 2.2: Feed Forward

- Red Neuronal Convolucional (CNN)

  Estas redes se inspiran en el funcionamiento de la visión humana, lo que hace a esta arquitectura muy útil en problemas donde los datos de entrada presentan una distribución espacial, especialmente en las imágenes. Estas redes se construyen con dos grupos de capas, un primer grupo que extrae características(features) de los datos mediante convolución y un segundo grupo que realiza la tarea de clasificación o regresión, es decir, como ya hemos comentado antes el segundo grupo es una NN

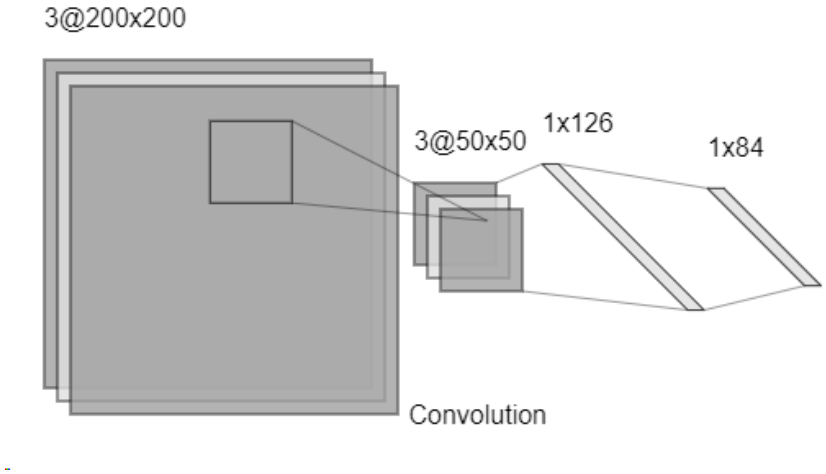

Figura 2.3: Convolucional

- Red Neuronal Recurrente (RNN)

  Estas redes permiten la conexión de la salida de una capa a una capa anterior, lo que crea bucles llamados celdas de memoria y las convierte en candidatos ideales para el procesamiento de series temporales, en especial la arquitectura Long Short Term Memory (LSTM), que contiene compuertas que administran como la información fluye en las celdas, la puerta de entrada controla que información pasa a la memoria y la de olvido cuanto tiempo se almacena esta.

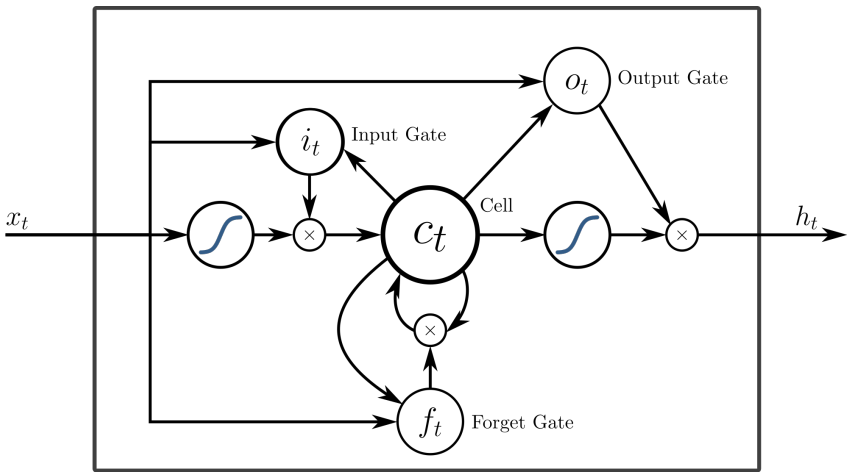

Figura 2.4: Long Short Term Memory

- Redes Generativas Adversas (GAN)

  Estas redes son capaces de aprender a generar datos similares a los que reciben como entrenamiento. El quid de esta arquitectura es la existencia de dos redes compitiendo entre si, una generadora, que toma datos basura como entrada y genera muestras y la discriminadora, que recibe muestras del generador y del conjunto de entrenamiento y deberá ser capaz de diferenciar entre las dos fuentes. De esta manera el generador intenta producir muestras cada vez mas indistinguibles del set de datos para engañar a la red discriminadora.

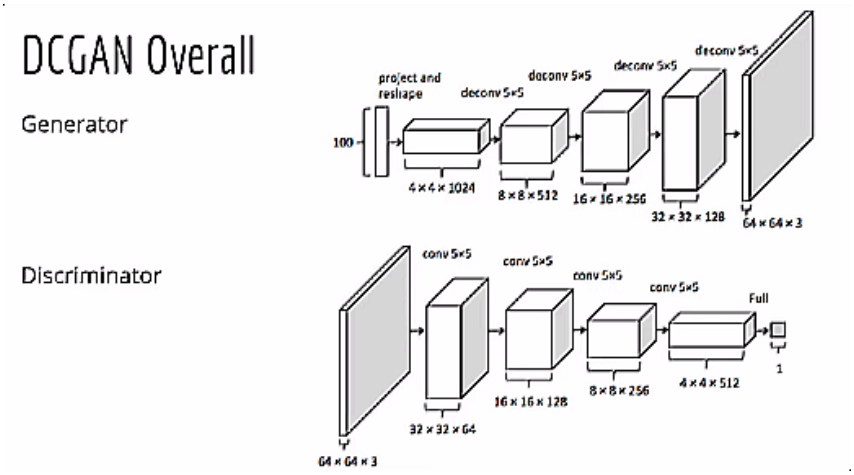

Figura 2.5: Generativa Adversa

2.3 Aprendizaje por Refuerzo: Una aproximación mas humana

Juegos de cartas, la brisca, el cinquillo, la escoba, la pocha, mentiroso . . . , clásicos pasatiempos que todos podemos disfrutar pues son sencillos y cuentan con una curva de aprendizaje mínima debido a como se aprende a jugar a estos. Cuando vas a aprender a jugar a un nuevo juego de cartas, la persona que va a enseñarte te explica vagamente las normas y suele terminar con la siguiente frase: *"Jugamos y ya lo iras pillando"*, en ese momento tu comienzas a jugar de una manera pseudoaleatoria mientras que el otro jugador mas experimentado te gana sin esfuerzo partida tras partida. Pero tras suficientes intentos, siguiendo tu estrategia al azar consigues ganar una partida y piensas *"Vale así se juega, te vas a enterar"*, pero tu amigo al ver que mejoras adapta sus estrategias a la que has descubierto, obligándote a seguir buscando una mejora hasta que le alcances.

Esta forma de aprender es la que imita el Aprendizaje por Refuerzo o Reinforcement Learning (RL), una rama de la Inteligencia Artificial que busca el aprendizaje por medio de la experiencia y el refuerzo positivo o negativo.

Este nuevo enfoque que ofrece el RL consta de dos componentes principales:

- Agente: El modelo que queremos entrenar para que aprenda a tomar decisiones.
- Entorno: Ambiente con el que el agente interactúa , extrayendo información de este para tomar decisiones, las cuales afectaran directamente al mismo.

Con estas simples pero efectivas definiciones, podemos notar la simbiótica relación que existe entre el agente y el entorno, una relación basada en los siguientes elementos:

- Acción: Una de las posibles decisiones que puede tomar y ejecutar el agente.
- Estado: Recoge la información necesaria para conocer cómo están los diversos elementos que conforman el entorno
- Recompensa: A raíz de cada acción, se producirá un cambio de estado y se puede otorgar en función de este una recompensa al agente implicado.

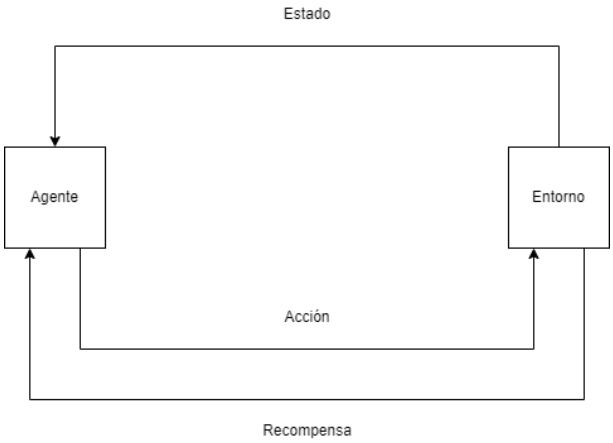

Figura 2.6: Relación Agente Entorno

La dinámica de esta relación puede resumirse en esta frase: *Çada acción tiene su reacción"*, el entorno condiciona al agente para tomar decisiones, las cuales a su vez modifican el entorno, dinámica que se representa matemáticamente como Procesos de Decisión de Markov (MDP).

2.3.1 Procesos de Decisión de Markov

Los MDP son una forma matemática de definir los problemas que se quieren resolver con RL, es decir, describen formalmente el entorno donde se desarrolla y actúa el agente.

2.3.1.1 Elementos implicados

Los MDP cumplen con la propiedad de Markov, que afirma que el futuro es independiente del pasado dado el presente, es decir, conocer el estado actual es suficiente para saber que reacción causaría cierta acción en el entorno. También se puede ver como que S_t contiene toda la información necesaria de los estados pasados.

Sin embargo todos estos estados tienen que registrarse, declararse de alguna manera. Por ello para determinar los posibles estados en los que puede encontrarse el entorno, se usa la matriz de transición de estados, la cual muestra la probabilidad de transición de un estado S a un estado S'

Conociendo esto, podemos definir y representar un MDP como una secuencia de estados que posee la propiedad de markov y como una tupla (S, P), donde S es la lista de estados a la que puede pertenecer y P la matriz de transición de estos.

Veamos un ejemplo para entenderlo mejor.

2.3.1.2 Caso de ejemplo

Vamos a usar el caso de una tarde típica de un niño en verano, comenzando desde que se sienta en el sofá tras comer. Teniendo como posibles estados estar en el sofá, merendar, jugar al fútbol, al ordenador, ir a la piscina, echarse la siesta y ducharse. Teniendo además la matriz de transición podemos definir este MDP como:

$$S = \Big\{ Merendar \quad Fútbol \quad Ordenador \quad Piscina \quad Siesta \quad Sofa \quad Ducha \Big\}$$

$$P = \begin{bmatrix} & Merendar & Fútbol & Ordenador & Piscina & Siesta & Sofa & Ducha \\ Merendar & * & 0{,}2 & 0{,}1 & 0{,}3 & 0{,}3 & 0{,}1 & 0 \\ Fútbol & 0 & * & 0{,}1 & 0{,}7 & 0 & 0{,}2 & 0 \\ Ordenador & 0{,}5 & 0 & * & 0 & 0{,}5 & 0 & 0 \\ Piscina & 0 & 0 & 0 & * & 0 & 0 & 1 \\ Siesta & 0{,}8 & 0{,}1 & 0{,}1 & 0 & * & 0 & 0 \\ Sofa & 0{,}6 & 0{,}2 & 0{,}1 & 0 & 0{,}1 & * & 0 \\ Ducha & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

Otro de los atractivos del uso de MDP es que se pueden representar como grafos.

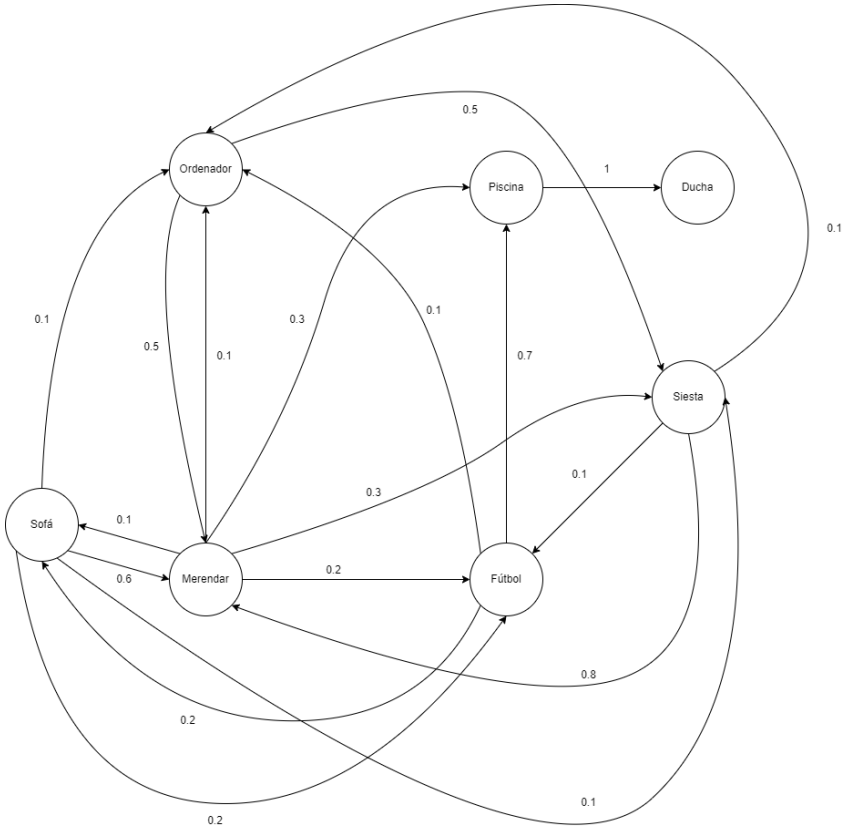

Figura 2.7: Grafo MDP

2.3.1.3 Recompensas de Markov

Una vez tenemos un entorno definido cuya dinámica esta clara y acotada el agente ya puede interactuar con este, pero para que además el agente pueda aprender necesita metas, motivaciones o recompensas. Es decir, un MDP con recompensas representa un problema en el que se juzga como de favorable o positivo

es transicionar a un estado u otro, lo cual permite al agente determinar que acciones tomar en función de las puntuaciones de estas.

Por ello, a la representación anterior de los MDP debemos añadir la lista de recompensas R, la cual guarda las recompensas de las transiciones entre estados, quedando de la siguiente manera: (S, P, R)

2.3.1.4 Ecuación de Bellman: Distancia a la recompensa

Al tener un sistema de puntuación para las acciones, aparece una regla muy sencilla para tomar decisiones: Ejecutar la acción con mayor recompensa en cada paso. Sin embargo, tener en cuenta solo las recompensas a corto plazo no es siempre favorable, pues por ejemplo en el ajedrez, a veces se debe sacrificar una pieza para tender una trampa a tu adversario y ganar ventaja a largo plazo. Para ello, se introducen dos nuevos conceptos, el valor de retorno G y el factor de descuento γ.

El valor de retorno es la suma ponderada de las recompensas de un camino concreto del grafo del MDP, es decir, de una secuencia definida de acciones donde el factor de descuento determina el peso que se le da a las recompensas futuras sobre la inmediata, siendo $\gamma = 1$ la misma importancia a ambas y $\gamma = 0$ solo al corto plazo.

$G = R_{t_{+1}} + \gamma(R_{t_{+2}} + R_{t_{+3}} ... + R_{t_{+n}})$

2.3.2 Q Learning

Cuando se conocen todas las recompensas asociadas a las acciones de un MDP es muy facil encontrar los caminos óptimos, es decir, los que dan mayor recompensa. Por ejemplo teniendo esta tabla de recompensas:

$$S = \begin{Bmatrix} & S_1 & S_2 & S_3 & S_4 \\ S_1 & * & 1 & -10 & 10 \\ S_2 & 3 & * & 1 & -10 \\ S_3 & 6 & 1 & * & 1 \\ S_4 & 0 & 100 & 2 & * \end{Bmatrix}$$

Con un factor de descuento 1 y empezando en S_1, obviamente el camino con mayor recompensa es $\begin{Bmatrix} S_1 & S_4 \end{Bmatrix}$ cuyo valor de retorno es

$$G = R_{t_{+1}} + \gamma R_{t_{+2}} = 10 + 1 * 100 = 110$$

.

Sin embargo este es un caso ideal, en un problema que requiere el uso de RL para su resolución, no se conoce esta tabla de recompensas también llamada Q-Table, lo cual soluciona el algoritmo Q Learning, el cual consiste en que el agente deberá explorar por su cuenta el entorno e ir generando esta tabla mediante dos sencillas reglas:

- Si una acción desde un estado determinado causa un resultado desfavorable, es decir, una recompensa negativa, el agente deberá aprender a evitarla, por otra parte si la acción genera un resultado positivo, el agente deberá llevarla a cabo cada vez que se encuentre en ese estado.
- Respecto a los estados, cuando las acciones disponibles en uno de estos solo aporten recompensas negativas, el agente deberá aprender a evitar las acciones que llevan a estos, y si un estado tiene acciones muy positivas el agente deberá priorizar acciones que lo lleven hasta este.

Teniendo claras estas dos premisas, podemos determinar el valor de retorno o valor Q de una acción a desde un determinado estado s como:

$$Q'_{(s_t,a_t)} = r_{(s_t,a_t)} + \gamma max_{a_{t+1}} Q_{(s_{t+1},a_{t+1})}$$

Es decir, el valor Q para un par estado acción es la suma de la recompensa inmediata recibida por esa acción mas el mejor valor Q que se puede obtener del estado al que nos lleva esta acción descontado por el factor de descuento. Para aproximar estos cálculos, los valores de la Q-Table se establecen a un valor fijo que puede ser o no aleatorio, el agente, al interactuar con el entorno va tomando pares estado acción y anotando las recompensas recibidas y así calcula el valor Q de estos usando la ecuación ya mencionada para guardarlo en la tabla.

Para hacer que la actualización de los valores Q sea mas gradual se incorpora el factor de aprendizaje α que genera la siguiente variante de la ecuación Q:

$$Q'_{(s_t,a_t)} = (1-\alpha)Q_{(s_t,a_t)} + \alpha(r_{(s_t,a_t)} + \gamma max_{a_{t+1}} Q_{(s_{t+1},a_{t+1})})$$

De esta manera el nuevo valor Q es una suma ponderada del valor Q actual y el valor Q aprendido, cuyos pesos dependen del factor de aprendizaje.

Una particularidad importante de este algoritmo, es que solo se aprenden los valores Q de las acciones que se ejecutan sobre los estados transitados, es decir, no se obtiene ningún tipo de información de estados u acciones no transitados, por ello es conveniente que al principio del aprendizaje el agente seleccione acciones al azar en busca de estados desconocidos (Exploración) y que al final del aprendizaje solo siga los caminos con mayor recompensa (Explotación). Para poder cumplir este equilibrio, la acciones se seleccionan en función a una distribución de probabilidad que se ve afectada por tres factores:

- Iteración actual: En las iteraciones iniciales el agente jugara de manera aleatoria y a medida que aumente este parámetro se volverá mas voraz.
- Valor Q relativo: Las mejores acciones deben tener mas probabilidades de ser seleccionadas y las peores deben acercarse lo máximo posible a 0.
- Constante de explotación E: Determina como de sensible es la distribución a los dos factores anteriores.

Sabiendo esto, definimos la probabilidad de tomar una acción a como la exponencial de el valor Q multiplicado por la iteración t y la constante de explotación dividido entre el sumatorio de lo mismo de cada acción posible.

$$P(a_t) = \frac{e^{E \cdot t \cdot Q(s_t,a_t)}}{\sum_{a \in A_{s_t}} e^{E \cdot t \cdot Q(s_t,a)}}$$

Para ejemplificar el funcionamiento del Q Learning, vamos a utilizar un sencillo tablero lineal, con casillas con recompensas y penalizaciones y en el que el agente solo tiene dos acciones, moverse a la derecha o moverse a la izquierda, con las que debe intentar llegar a la ultima casilla del tablero, la de la derecha.

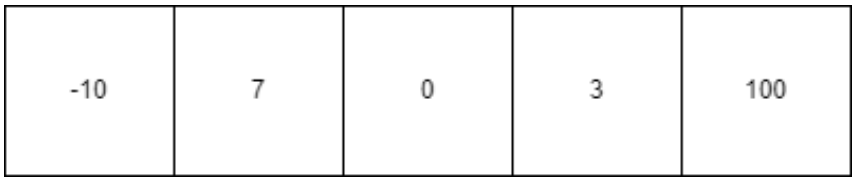

Figura 2.8: Tablero Q Learning

El calculo de la recompensa para un acción es sencillo, a la recompensa asociada a la casilla o estado al que se mueve el agente se le resta el numero de pasos que este ha dado, es decir, si es el primer paso y se mueve a la casilla 1, la recompensa será $7 - 1 = 6$.

Teniendo en cuenta que las casillas se numeran del 0 al 4 de izquierda a derecha tenemos la Q-Table inicial:

$$Q = \begin{Bmatrix} & S_0 & S_1 & S_2 & S_3 & S_4 \\ S_0 & * & 0 & 0 & 0 & 0 \\ S_1 & 0 & * & 0 & 0 & 0 \\ S_2 & 0 & 0 & * & 0 & 0 \\ S_3 & 0 & 0 & 0 & * & 0 \\ S_4 & 0 & 0 & 0 & 0 & * \end{Bmatrix}$$

El agente comienza cada iteración o partida desde una casilla aleatoria entre la 1 y la 3, con un factor de aprendizaje y constante de explotación 1.

- Iteración 1: Casilla de inicio 2
  - Acciones posibles: Derecha (0) e izquierda (1)
  - Valores Q: 0 y 0
  - Probabilidades: 0.5 y 0.5

  El agente selecciona la segunda acción, ir a la derecha, por lo que tras conocer la recompensa $3 - 1 = 2$ procede a calcular el nuevo valor Q para introducirlo en la tabla:

$$Q'_{(s_2,a_0)} = (1-1) \cdot 0 + 1 \cdot (2 + 1 \cdot 0) = 0 + 2 = 2$$

$$Q = \begin{Bmatrix} & S_0 & S_1 & S_2 & S_3 & S_4 \\ S_0 & * & 0 & 0 & 0 & 0 \\ S_1 & 0 & * & 0 & 0 & 0 \\ S_2 & 0 & 0 & * & 2 & 0 \\ S_3 & 0 & 0 & 0 & * & 0 \\ S_4 & 0 & 0 & 0 & 0 & * \end{Bmatrix}$$

  En la casilla 3, se encuentra con el siguiente caso:

  - Acciones posibles: Derecha (0) e izquierda (1)
  - Valores Q: 0 y 0
  - Probabilidades: 0.5 y 0.5

Seleccionando de nuevo al azar, el agente decide ir a la derecha, recibiendo la recompensa $100-2=98$ y guardando el nuevo valor Q en la tabla:

$$Q'_{(s_2,a_0)} = (1-1)\cdot 0 + 1\cdot(98+1\cdot 0) = 0+98 = 98$$

$$Q = \begin{Bmatrix} & S_0 & S_1 & S_2 & S_3 & S_4 \\ S_0 & * & 0 & 0 & 0 & 0 \\ S_1 & 0 & * & 0 & 0 & 0 \\ S_2 & 0 & 0 & * & 2 & 0 \\ S_3 & 0 & 0 & 0 & * & 98 \\ S_4 & 0 & 0 & 0 & 0 & * \end{Bmatrix}$$

Y asi finaliza la primera iteración, el resto transcurrirán de forma similar, exceptuando que las Qs ya no serán todas 0 por lo que la distribución de probabilidad dejara de ser uniforme y que el valor Q máximo del estado al que se transiciona no sera nulo tampoco.

2.3.3 Deep Q Learning

Completar la Q-Table del laberinto lineal del ejemplo anterior es relativamente sencillo, con unas pocas partidas el agente podría recorrer todos los estados posibles y así tener toda la información necesaria para obtener un desempeño perfecto. Si el problema a resolver fuese el mismo que el anterior pero con un tablero de 10 casillas el agente tendría que aprender hasta 10 valores Q, si el problema a resolver fuese aprender a jugar al juego de mesa tres en raya, el agente debería aprender hasta 9! Qs. No es complicado deducir viendo esto que la eficiencia del Q Learning decae a medida que aumenta el tamaño del espacio de estados a explorar, pues aunque una de las ventajas de este método de aprendizaje es la exploración inteligente (no necesita explorar todo el espacio de estados porque cierra caminos que no parecen útiles relativamente temprano) si hay demasiados estados, el aprendizaje se vuelve superficial con este método si no se explora lo suficiente, pero demasiada exploración provoca tiempos de aprendizaje y requerimientos de memoria colosales.

Para solucionar el problema de dimensionalidad del Q Learning, en 2013 Deep Mind publica un paper titulado *Playing Atari with Deep Reinforcement Learning* [6], sacando a la luz esta variante del aprendizaje reforzado que incluye DQN, las cuales aproximan la función generadora de la Q-Table, permitiendo así extraer información de estados ya transitados y usarla en los desconocidos gracias a su facilidad para generalizar.

Para entrenar una NN en un problema de aproximación de funciones necesitamos datos etiquetados, lo cual es el principal inconveniente a la hora de usar estas redes en el aprendizaje reforzado, ya que los valores no se conocen a priori y es un trabajo a ciegas. Para derribar este muro que separa estos dos algoritmos de aprendizaje, se reinventa el aprendizaje supervisado, de manera que la red en vez de aprender los valores exactos que se esperan según cierto dato de entrada, aprende una distribución de valores relativa que sin ser exactamente la original, mantiene la dinámica de esta. Puede compararse con transponer una partitura musical, cambia la tonalidad pero la canción es la misma.

Un agente de Deep Q Learning se compone de tres elementos esenciales:

1. Value Network: Encargada de aproximar el valor Q de todas las acciones dado un estado como input
2. Target Network: Red con la misma topología que la anterior pero con sus propios pesos, encargada de aproximar el valor de la recompensa a largo plazo $max_{a_{t+1}} Q_{(s_{t+1},a_{t+1})}$
3. Experience Replay: Acciones y sus recompensas asociadas de partidas jugadas anteriormente que se samplean aleatoriamente para el entrenamiento de las redes

Debido al uso de estas DQN espacio de salida cambia del aprendizaje reforzado al reforzado profundo, ya que la Q-Table tiene como entrada un estado y un acción a realizar desde este y las DQN tienen como entrada un estado y como salida un valor Q para cada acción.

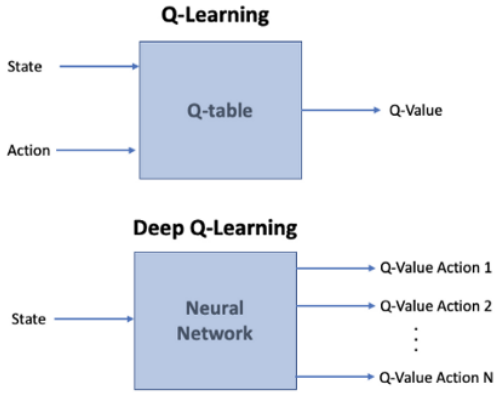

Figura 2.9: Espacio de salida

A continuación profundizaremos en el punto mas interesante de este algoritmo, el entrenamiento de las DQN. Para generar datos y usar así un aprendizaje supervisado, se utiliza la ya mencionada ecuación de Bellman:

$$Q'_{(s_t,a_t)} = (1-\alpha)Q_{(s_t,a_t)} + \alpha(r_{(s_t,a_t)} + \gamma max_{a_{t+1}} Q_{(s_{t+1},a_{t+1})})$$

Sustituyendo los valores antes obtenidos de la tabla por los obtenidos de las DQN de la siguiente manera:

1. $Q_{(s_t,a_t)}$ se sustituye por el valor Q para dicha acción obtenido de la Value Network
2. $max_{a_{t+1}} Q_{(s_{t+1},a_{t+1})}$ se sustituye por el máximo valor de los generados por la Target Network

Teniendo asi la ecuación de Bellman para Deep Q Learning:

$$Q'_{(s_t,a_t)} = (1-\alpha)(ValueNetwork)_{a_t} + \alpha(r_{(s_t,a_t)} + \gamma(TargetNetwork)Q_{(s_{t+1},a_{t+1})})$$

De esta manera, obtenemos el nuevo valor Q para dicha acción que se usa para entrenar la Value Network, cuyos pesos son copiados a la Target Network cada N iteraciones la cual no se entrena nunca.

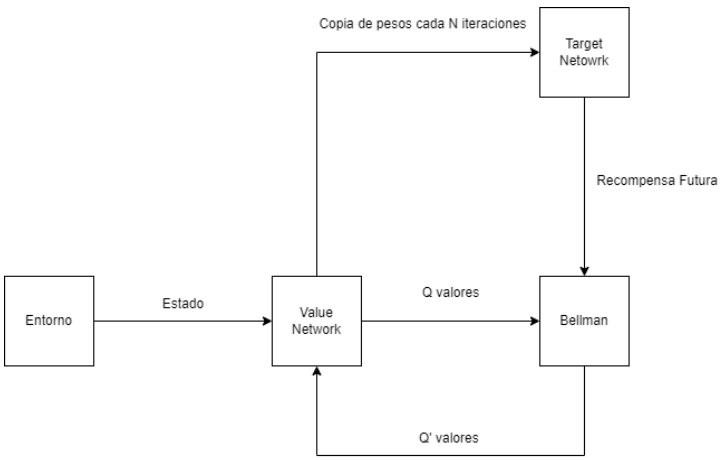

Figura 2.10: Arquitectura Agente Deep Q Learning

2.4 Agentes de Risk

Los juegos de mesa siempre han sido un buen entorno en el que desarrollar y probar algoritmos de aprendizaje, el ajedrez, backgammon, go... y aunque el Risk no es uno de estos, es uno de los juegos de mesa mas populares y conocidos por lo que se han explorado varios algoritmos para crear agentes que tengan un buen desempeño jugando a este.

Martin Sonesson publica en 2008 una entrada en su blog titulada *Creating an AI for Risk board game* [7], en la cual expone las claves del proceso de creación de un agente que juega al Risk con un algoritmo en el que convergen el clásico *Minimax* y los algoritmos evolutivos. El agente que Martin Sonesson propone una IA que en cada turno debe sopesar todas las posibles acciones a tomar y determinar cual es la mas favorable, teniendo en cuenta para realizar este calculo de favorabilidad la proximidad a otros jugadores, lo fuertes o débiles que son estos, la posibilidad de conseguir o arrebatar continentes... Todas estas condiciones tienen un peso asignado por el creador, que determina como de importantes son para calcular el valor asociado a una acción y son estos los que fluctuarán durante el proceso de aprendizaje, que es donde entra en juego el componente evolutivo, pues este consiste en que se genera un agente con pesos aleatorios que se enfrentará al original y en cada iteración el que gane mantendrá su configuración y el perdedor recibirá unos nuevos pesos aleatorios.

Con una aproximación muy similar que también hace uso de un entrenamiento evolutivo pero llevando mas allá el proceso de selección de acciones, el usuario de GitHub arman-aminian publicó hace dos años en su perfil el código de un agente de Risk titulado *Implementation of the Risk Game and Artificial Intelligence Agent Player*[9], en el cual describe el funcionamiento de un agente que hace uso del algoritmo *Minimax Alpha Beta con poda* para determinar que acciones tomar. Este algoritmo consiste en un árbol de decisión donde los nodos son estados y las ramas acciones posibles, en el cual no se profundizan todos los caminos y se desechan o podan algunos de ellos en función de una heurística.

Por último, el mas cercano al enfoque que vamos a explorar en este proyecto, Erik Blomqvist en su proyecto de fin de grado de Ingenieria Informatica [8] diseñó un agente que aprende a jugar al Risk mediante el uso del algoritmo AlphaZero, variante del famoso algoritmo de aprendizaje para el juego de mesa

Go AlphaGo. Este algoritmo esta a la cabeza del aprendizaje reforzado profundo, combinando las DQN con Policy Networks y arboles de decisión, llegando a niveles de aprendizaje hasta antes desconocidos.

Capítulo 3

Desarrollo y entrenamiento de los agentes

La ciencia se compone de errores, que a su vez, son los pasos hacia la verdad.

Julio Verne

3.1 Introducción

En este capítulo se incluirá la descripción del desarrollo del trabajo.

El capítulo se estructura en 4 apartados:

1. Implementación de los agentes de Q Learning y Deep Q Learning
2. Resolución del Tres en Raya con estos agentes
3. Resolución de una versión simplificada del Risk con el agente de Deep Q Learning
4. Resolución del Risk con el agente de Deep Q Learning

3.2 Implementación del Agente Q-Learning

Como hemos visto en 2.3.2 un agente de Q Learning debe cumplir las siguientes premisas:

1. Input: Estado actual
2. Output: Acción seleccionada
3. Q-Table propia
4. Hiperparametros: Factor de descuento, de exploración y learning rate
5. Valor inicial para la Q-Table

Para que cumpla todos estas premisas y su función de llenar la Q-Table y usarla para determinar que acción tomar en cada situación, hemos creado una clase llamada agente que diseccionaremos a continuación (Cabe destacar que el diseño de este agente esta enfocado a resolver el Tres en Raya).

3.2.0.1 Hiperparametros y Q inicial

Los hiperparametros y la Q inicial son variables de clase, es decir, no pertenecen a la instancia concreta, sino a la clase, asi pues todas las instancias de un agente comparten estos parámetros.

Listado 3.1: Variables de clase

```
descuento = 0.68 #Factor de descuento
alfa = 0.25 #Learning Rate
e = 0.15 #Factor de exploracion
qinicial = 6 #Valor Q inicial
```

3.2.0.2 Variables de instancia

El agente tiene cuatro variables de instancia:

- Un diccionario que representa la Q table no como una tabla, sino como pares estado valor, que otorgan un valor Q a cada estado, reduciendo de esta manera la cantidad de memoria necesaria.
- El jugador que representa, X u O (Input para instanciar)
- Una instancia del entorno o generador de partidas (Input para instanciar, deben comapartir el mismo todos los agentes que se entrenen juntos)
- La iteración actual, el numero de partidas jugadas

Listado 3.2: Variables de instancia

```
self.estados = dict() #Q-Table
self.player = player #X u O
self.juego = juego #Juego: clase con el entorno del Tic Tac Toe
self.iteracion = 0 #Iteracion actual
```

3.2.0.3 Política Aleatoria

Esta política toma el estado actual y selecciona aleatoriamente una acción entre las posibles.

Listado 3.3: Política Aleatoria

```
def politicaAleatoria(self):

        acciones, tableros = self.juego.getTableros(self.player)
        #Acciones posibles y estado actual

        return acciones[random.randint(0, len(acciones)-1)]
        #Seleccion aleatoria de accion
```

3.2.0.4 Política Probabilística

Esta función es el epicentro del agente, donde se toman las decisiones de juego durante el entrenamiento.

Primero se obtienen el estado actual y las acciones posibles desde este, para a continuación extraer del diccionario los valores Q de cada estado al que se llega con dichas acciones, teniendo en cuenta que los estados aun no transitados no están en el diccionario por lo que su valor Q sera el asignado en la variable de clase *qinicial*.

Una vez se tienen las Qs, usando la formula mencionada en 2.3.2 se calculan las probabilidades de realizar cada acción disponible.

Y para terminar mediante una selección aleatoria ponderada se toma una acción un otra.

Listado 3.4: Política Probabilistica

```
def politica(self):

        #Acciones posibles y estado actual

        acciones, tableros = self.juego.getTableros(self.player)

        # Sacamos las Qs de los tableros

        qs = []

        for tablero in tableros:
                if self.estados.get(tablero) == None:
                        qs.append(qinicial)
                else:
                        qs.append(self.estados.get(tablero))

        # Calculo de probabilidades

        qtotal = 0

        for q in qs:
                qtotal += math.exp(q*e*t)

        probabilidades = []

        for q in qs:
                probabilidades.append(math.exp(q*e*t)/qtotal)


        # Elegimos aleatoriamente la accion

        indice = choice(len(acciones), 1, p=probabilidades)


        return acciones[indice[0]]
```

3.2.0.5 Política Voraz

Esta política tras ver el estado actual y las acciones posibles y de la misma manera que el anterior conocer los valores Q de estas, selecciona la acción con mayor valor Q.

Listado 3.5: Política Voraz

```
def politicaVoraz(self):

        #Acciones posibles y estado actual

        acciones, tableros = self.juego.getTableros(self.player)

        # Sacamos las Qs de los tableros

        qs = []

        for tablero in tableros:
                if self.estados.get(tablero) == None:
                        qs.append(qinicial)
                else:
                        qs.append(self.estados.get(tablero))


        # Elegimos la accion con mayor q

        qmax = max(qs)

        indice = qs.index(qmax)

        return acciones[indice]
```

3.2.0.6 Actualización de la Q-Table

Esta es, junto a la política probabilística 3.2.0.4, el nucleo del agente de Q Learning, pues es en esta función donde adquiere el conocimiento.

Al terminar una partida, el agente recibe una lista ordenada de todos los estados que se han transitado en la iteración (incluidos los de su oponente), y el ganador de la partida, que puede ser X, O y empate. Teniendo en cuenta este ultimo dato, se define la recompensa asociada al estado final, 10 si ha ganado, 0 si ha perdido y 5 si ha empatado.

A continuación, para poder guardarlos en la Q-Table, el agente debe saber que estados de la lista han sido consecuencia de sus acciones y cuales no, y para ello se usa la función auxiliar setEstados, la cual tiene el siguiente funcionamiento:

- Determinar si el agente empezó la partida. Si lo hizo los estados que le pertenecen son los impares, sino los pares
- Recorrer la lista por los estados que le pertenecen al agente guardando en el diccionario con valor Q inicial los que se desconocían hasta ahora.
- Todos los estados que son consecuencia de las acciones del agente se guardan en una lista que es el output de la función

Una vez se ha conseguido la lista de estados propios del agente, esta se recorre de final a principio y se actualizan los valores Q haciendo uso de la ecuación del Bellman, excepto con el estado final, que su nueva Q es la recompensa asignada en el inicio de la función.

Para calcular la recompensa futura del resto de estados, se usa una función auxiliar llamada maxQ-Proxima que actúa de la siguiente manera:

- Obtiene los estados hijo del que recibe como input, es decir, los estados generados por el contrincante a partir del estado actual.
- Recorriendo la lista de estados hijo del actual, en cada uno se generan los estados hijos de este, es decir, la siguiente generación de estados que son consecuencia de una acción del agente.
- Entre esos estados nieto del actual, se obtienen sus Qs y se devuelve la de mayor valor.

Para finalizar, se aumenta en uno el contador de iteraciones.

Listado 3.6: setEstados

```
def setEstados(self, pasos):

        propios = [] #Pasos del jugador para devolver

        #Se comprueba si el agente ha empezado la partida
        #para determinar que pasos de esta son acciones suyas
        empieza = False

        if pasos[1].count(self.player)==1: #Si ha hecho el primer movimiento
                empieza = True

        indice = 0

        for estado in pasos: #Por cada paso

                if empieza and (indice%2!=0): #Si empieza sus estados son los impares

                        if self.estados.get(estado) == None:

                                self.estados.setdefault(estado, qinicial)

                        propios.append(estado)

                elif (not empieza) and (indice%2==0) and (indice != 0):
                #Si no empieza sus estados son los pares pero no el tablero vacio

                        if self.estados.get(estado) == None:

                                self.estados.setdefault(estado, qinicial)

                        propios.append(estado)

                indice += 1
        return propios
```

Listado 3.7: maxQProxima

```
        def maxQProxima(self,estado):

                hijos = self.juego.getHijosContrario(estado, self.player)
                #Sacamos los hijos del estado, los que son estados del otro jugador

                nmax = 0

                for hijo in hijos: #por cada hijo

                        nietos = self.juego.getHijosPropios(hijo, self.player)
                        #Sacamos los nietos, estados del jugador propio



                        for nieto in nietos: #por cada nieto

                                qnieto = self.estados.get(nieto)
                                #Sacamos su q y si es mayor que nmax las intercambiamos

                                if qnieto != None:

                                        if qnieto > nmax:

                                                nmax = qnieto


                return nmax
```

Listado 3.8: Actualizar Q-Table

```
def actualizar(self, pasos, ganador):

        #Definimos recompensa del estado final
        recompensa = 0

        if ganador == self.player:

                recompensa = 10

        elif ganador == "-":

                recompensa = 5

        # guardamos los pasos
        propios = self.setEstados(pasos)

        primero = True

        indice = len(propios)-1

        while indice >= 0 :

                q = self.estados.get(propios[indice])
```

```
            if not primero:

                    #calculo nueva q

                    qfutura = recompensa + descuento * self.maxQProxima(propios[indice])

                    nuevaq = (1-alfa) * q + alfa*qfutura

                    #actualizamos q
                    recompensa = nuevaq
                    dic = {propios[indice]: nuevaq}
                    self.estados.update(dic)
            else:

                    #calculo nueva q

                    nuevaq = recompensa

                    #actualizamos q
                    dic = {propios[indice]: nuevaq}
                    self.estados.update(dic)

            indice = indice - 1
            primero = False

    self.iteracion += 1
```

3.2.0.7 Serializado y Carga

Para guardar y cargar los agentes se ha usado la librería de serializado de python llamada pickle, con la que se guardan la ficha con la que juega el agente, el diccionario con la Q-Table y la iteración actual.

Listado 3.9: Serializado

```
    def guardar(self, archivo):

            archivo = open(archivo+'.pickle', 'wb')
            array = [self.player, self.estados, self.iteracion]
            sys.setrecursionlimit(100000)
            pickle.dump(array, archivo)
            archivo.close()
```

Listado 3.10: Carga

```
    def cargar(self, archivostr):

            archivo = open(archivostr+'.pickle', 'rb')
            array = pickle.load(archivo)
            self.player = array[0]
            self.estados = array[1]
            self.iteracion = array[2]
            print("Agente " + archivostr + " con " + str(len(self.estados)) +
                    " estados y que juega con la ficha: " + self.player + " cargado")
```

3.3 Implementación del Agente Deep Q-Learning

Como decíamos en 2.3.3, aunque comparten puntos en común la gran diferencia entre un agente de Q Learning y uno de Deep Q Learning es que el segundo sustituye la Q-Table por dos DQNs, lo que supone importantes cambios respecto a su entrenamiento, es decir, al método de descubrir los Q valores.

Teniendo en cuenta esto las premisas principales de este agente son:

1. Input: Estado actual
2. Output: Acción seleccionada
3. Experience Replay: Memoria de partidas anteriores
4. Hiperparametros: Factor de descuento, de exploración y learning rate
5. DQNs

Aunque este agente es una modificación del anterior, hay cambios importantes en su programación que veremos a continuación.

3.3.0.1 Hiperparametros

Como en el agente anterior 3.2.0.1, estos son variables de clase.

Listado 3.11: Variables de clase

```
descuento = 0.68
alfa = 0.25
e = 0.3
```

3.3.0.2 Variables de instancia

En este agente, las variables de instancia cambian pues ya no se necesita la Q-Table, ahora se tiene las siguientes variables:

- La lista que guarda movimientos de partidas anteriores y un entero que representa su tamaño maximo.
- Lista con los movimientos de la partida actual
- El jugador que representa, X u O (Input para instanciar)
- Una instancia del entorno o generador de partidas (Input para instanciar, deben comapartir el mismo todos los agentes que se entrenen juntos)
- La iteración actual, el numero de partidas jugadas
- Las DQN

Listado 3.12: Variables de instancia

```
self.replay = [] #Partidas anteriores
self.partida = [] #Partida actual
self.player = player #X u O
self.juego = juego #Juego:  Clase con el entorno del Tic Tac Toe
self.iteracion = 0 #Iteracion actual
self.replaysize = 1000 #Longitud maxima replay

self.model = Sequential()
self.model.add(Flatten(input_shape= array.shape))
self.model.add(Dense(50))
self.model.add(Activation('relu'))
self.model.add(Dense(9))
self.model.add(Activation('linear'))
optimizer = Adam(learning_rate=0.01)
self.model.compile(loss='mse', optimizer=optimizer)

self.model2 = Sequential()
self.model2.add(Flatten(input_shape= array.shape))
self.model2.add(Dense(50))
self.model2.add(Activation('relu'))
self.model2.add(Dense(9))
self.model2.add(Activation('linear'))
self.model2.compile(loss='mse', optimizer=optimizer)

self.model2.set_weights(self.model.get_weights())
```

3.3.0.3 Política Aleatoria

Misma que el agente anterior 3.2.0.3

3.3.0.4 Política Probabilística

Al igual que en el anterior, esta función es el epicentro de la toma de decisiones del agente, pero el funcionamiento no es el mismo.

Si recordamos el funcionamiento del agente anterior, este recibia la lista de movimientos del entorno al acabar la partida, sin embargo ahora los estados y acciones se guardan en el agente en cuanto son tomadas estas acciones. Cada par estado acción se representa con una lista de longitud cuatro que representa:

- Estado actual
- Acción tomada
- Recompensa asociada
- Estado al que lleva tomar dicha acción. Cuando se añade un nuevo estado, esta posición de la lista se declara vacía, y en el siguiente turno el estado en el que se encuentre el entorno se guarda en esta posición, encadenando de esta manera el flujo de acciones.

Por ello, lo primero que se hace es (si no es el primer movimiento) guardar el estado actual en la posición de estado hijo del anterior.

A continuación se presenta a la red de valor el estado como input obteniendo así un valor Q para cada acción.

Una vez se tienen las Qs, usando la formula mencionada en 2.3.2 se calculan las probabilidades de realizar cada acción disponible.

Y para terminar mediante una selección aleatoria ponderada se toma una acción un otra.

Listado 3.13: Política Probabilistica

```
def politica(self):

        if len(self.replay) != 0 :

                index = len(self.replay) - 1

                self.replay[index][2] = self.juego.getReward(self.player)

        tablero = self.juego.getEstado()

        # Sacamos las Qs de los tableros
        qs = self.model.predict([[tablero]])

        qs = qs[0]


        # Calculo de probabilidades

        qtotal = 0

        for q in qs:
                qtotal += math.exp(q*e)

        probabilidades = []

        for q in qs:
                probabilidades.append(math.exp(q*e)/qtotal)

        accion = choice(len(qs), 1, p=probabilidades)

        accion = accion[0]

        row = accion // 3

        col = (accion % 3)

        reward = 0

        nuevo = [tablero,accion,reward,[]]

        self.partida.append(nuevo)

        return [row,col]
```

3.3.0.5 Política Voraz

Esta función es igual que la del anterior agente 3.2.0.5, excepto por el método de obtención de las Qs, que en este caso es usando la Value Net.

Listado 3.14: Política Voraz

```
def politicaVoraz(self):

        # Sacamos las Qs de los tableros
        tablero = self.juego.getEstado()
        qs = self.model.predict([[tablero]])

        #print(qs)

        q = np.argmax(qs)

        row = q // 3

        col = (q % 3)

        return [row,col]
```

3.3.0.6 Entrenamiento de las DQN

Esta es, junto a la política probabilística 3.2.0.4, el nucleo del agente de Q Learning, pues es en esta función donde adquiere el conocimiento.

Al terminar una partida, el agente tiene una lista ordenada de todos los estados que se han transitado en la iteración (incluidos los de su oponente), y recibe el ganador de la partida, que puede ser X, O, empate o error de cualquiera de los jugadores (Las DQN también pueden aprender a discriminar acciones *ilegales*). Teniendo en cuenta este ultimo dato, se define la recompensa asociada al estado final, 10 si ha ganado, -5 si ha perdido, 5 si ha empatado y -10 si ha realizado un movimiento erróneo.

A continuación, se guardan las acciones de la partida en la lista que guarda la experiencia anterior, se vacía la lista de la partida para usarse en la siguiente y se aumenta en uno el contador de iteración.

A continuación, si es momento de entrenar la red(parámetro de diseño), se comprueba si la lista con la experiencia tiene el tamaño máximo definido o menos, en caso de no tenerlo se extraen elementos del inicio de la lista hasta solucionarlo, y se llama a la función auxiliar *train* que entrena la red.

Esta función recibe como parámetro el tamaño del batch de entrenamiento y tras seleccionar una muestra aleatoria de movimientos de dicho tamaño de la lista de partidas anteriores sigue los siguientes pasos para cada muestra:

- Obtiene la lista de Qs desde el estado
- Si es una acción final o ilegal, se cambia su valor Q de la lista por la recompensa asociada. Si no es final ni ilegal, se cambia por el valor calculado usando la ecuación de Bellman modificada 2.3.2.
- Las listas con un valor Q modificado de cada movimiento de la muestra se usan como salida a imitar para la red en su entrenamiento, el cual solo consiste en una época.

Para calcular la recompensa futura en la ecuación de Bellman, se usa una función auxiliar llamada maxQProxima que actúa de la siguiente manera:

- Obtiene los estados hijo del que recibe como input, es decir, los estados generados por el contrincante a partir del estado actual.
- Recorriendo la lista de estados hijo del actual, en cada uno se generan los estados hijos de este, es decir, la siguiente generación de estados que son consecuencia de una acción del agente.
- Entre esos estados nieto del actual, se obtienen sus Qs con la Target net y se devuelve la de mayor valor.

Tras entrenar la red, si es el momento se copian los pesos de esta a la Target net.

Listado 3.15: maxQProxima

```
def maxQProxima(self,estado,accion):

            estado = self.juego.getHijo(estado, accion, self.player)

            estado = self.juego.getEstadoReverse(estado)

            hijos = self.juego.getHijosContrario(estado, self.player)
            #saco los hijos del estado, los que son estados del otro jugador

            nmax = 0

            for hijo in hijos: #por cada hijo

                    x = self.juego.convertir(hijo)
                    qs = self.model2.predict([[x]])

                    qhijo = np.max(qs)

                    if qhijo> nmax:

                    nmax = qhijo

            return nmax
```

Listado 3.16: Train

```
def train(self,batch):

        inputs = []
        outputs = []


        if len(self.replay) < batch:

                for jugada in self.replay:

                        x = jugada[0] #estado

                        qs = self.model.predict([[x]])[0]
                        #Predecir qs de las acciones desde el estado

                        if (jugada[2] == -10) or (jugada[2] == 10) or (jugada[2] == -5):
                        #Si es un movimiento ilegal
```

```
                qFutura = jugada[2] #Valor q futuro


            else:

                qFutura = jugada[2] + descuento * self.maxQProxima(x,jugada[1])
                #Valor q futuro

            qs[jugada[1]] = (1-alfa) * qs[jugada[1]] + alfa * qFutura
            #Actualizamos la q

            inputs.append(x) #Anadimos el estado convertido a los inputs

            outputs.append(qs)


        self.model.fit([inputs],[outputs],batch_size=batch,verbose=0, shuffle=True)

    else:

        indices = np.random.randint(len(self.replay), size=batch)

        for indice in indices:

            jugada = self.replay[indice]

            x = jugada[0] #estado

            qs = self.model.predict([[x]])[0]
            #Predecir qs de las acciones desde el estado

            if (jugada[2] == -10) or (jugada[2] == 10) or (jugada[2] == -5):
            #Si es un movimiento final

                qFutura = jugada[2] #Valor q futuro



            else:

                qFutura = jugada[2] + descuento * self.maxQProxima(x,jugada[1])
                #Valor q futuro


            qs[jugada[1]] = (1-alfa) * qs[jugada[1]] + alfa * qFutura
            #Actualizamos la q

            inputs.append(x) #Anadimos el estado convertido a los inputs

            outputs.append(qs)

        self.model.fit([inputs],[outputs],batch_size=batch,verbose=0, shuffle=True)
```

Listado 3.17: Entrenar DQNs

```
def actualizar(self, ganador):

    #Definimos recompensa del estado final
```

```
recompensa = 0

if ganador == self.player: #Si gana

        recompensa = 10

elif ganador == "-": #Si es empate

        recompensa = 5

elif ganador == (self.player + "error"): #Si ha fallado

        recompensa = -10

elif self.player == "X":

        if ganador == "O": #Si pierde

                recompensa = -5

elif self.player == "O":

        if ganador == "X": #Si pierde

                recompensa = -5

#Anadimos los movimientos a replay

primero = True

indice = len(self.partida)-1

while indice >= 0 :

        if primero:

                self.partida[indice][2] = recompensa

        self.replay.append(self.partida[indice])

        indice = indice - 1
        primero = False


self.partida = []

self.iteracion += 1

#Entrenamiento de las redes

if self.iteracion%50==0:

        if self.replaysize <= len(self.replay):

                diferencia = len(self.replay) - self.replaysize

                for i in range(diferencia):

                        self.replay.pop(0)
```

```
                self.train(32)

            if self.iteracion%100==0:

                self.model2.set_weights(self.model.get_weights())
```

3.3.0.7 Serializado y Carga

Al igual que en el anterior, para guardar y cargar los agentes se ha usado la librería de serializado de python llamada pickle, con la que se guardan la ficha con la que juega el agente, la lista con partidas anteriores y la iteración actual. Además con las propias funciones de keras se serializan las DQNs.

Listado 3.18: Serializado

```
def guardar(self, archivo):
        name = archivo
        archivo = open(archivo+'.pickle', 'wb')
        array = [self.player, self.replay, self.iteracion]
        sys.setrecursionlimit(100000)
        pickle.dump(array, archivo)
        archivo.close()
        self.model.save_weights(name+'.h5')
```

Listado 3.19: CargaD

```
def cargar(self, archivostr):
        archivo = open(archivostr+'.pickle', 'rb')
        array = pickle.load(archivo)
        self.player = array[0]
        self.replay = array[1]
        self.iteracion = array[2]
        self.model.load_weights(archivostr+'.h5')
        self.model2.load_weights(archivostr+'.h5')
        print("Agente " + archivostr + " con " + str(len(self.replay)) +
                " estados y que juega con la ficha: " + self.player + " cargado")
```

3.4 Resolución Tres en Raya

3.4.1 Configuración del entorno

El entorno se rige por las reglas del Tic Tac Toe o Tres en raya clásicas:

1. En su turno el jugador coloca su ficha en una casilla vacía
2. El jugador que alinea tres fichas gana
3. Si se llena el tablero sin que nadie gane se considera empate

El estado del juego se representa con el tablero del juego, que es una matriz 3x3, en una lista de longitud 9, donde un 0 significa que la casilla esta vacía, un 1 que esta ocupada por el jugador con la ficha *X* y un -1 que esta ocupada por el jugador con la ficha *O*.

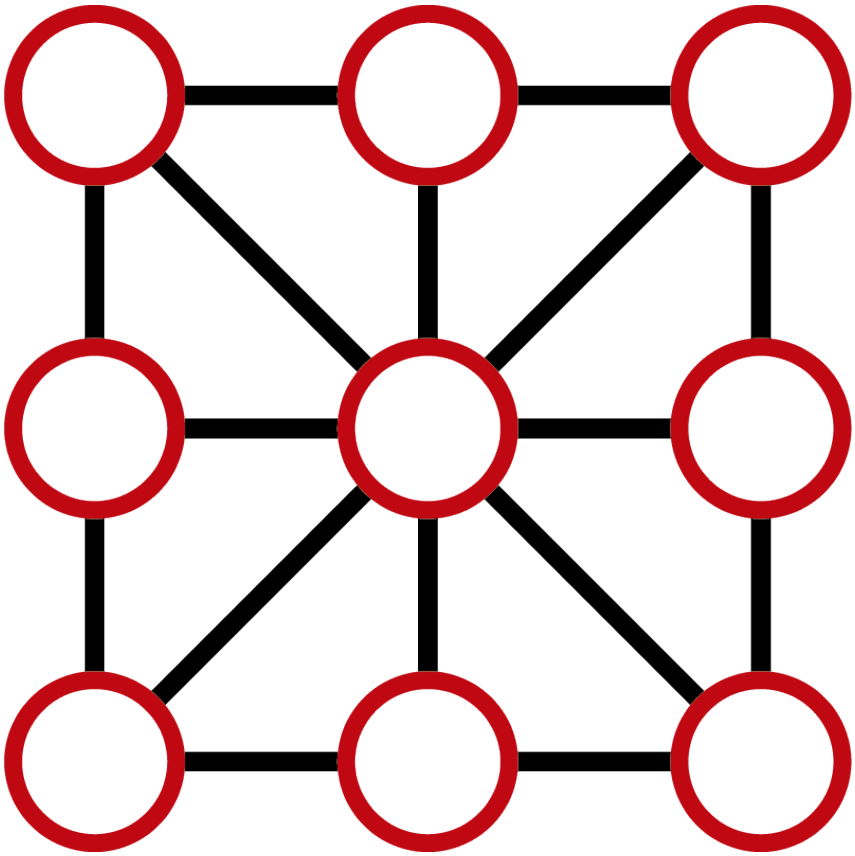

Figura 3.1: Tablero Tres en Raya

3.4.2 Configuración del agente en el entorno

El Tic Tac Toe solo permite un tipo de acción, seleccionar en que casilla colocar la ficha, por lo que será esta la única acción que el agente pueda realizar.

El agente de Q Learning no necesita mas configuración para adaptarse al entorno y en cada turno tendrá que buscar entre nueve y un Q valores.

3.4.2.1 Diseño de las Redes

Para que el agente de Deep Q learning pueda usar el entorno, necesita la siguiente configuración en sus DQNs:

1. Una capa de 9 neuronas de entrada: La representación del estado
2. Una capa de 9 neuronas de salida: Una por cada casilla en la que colocar ficha

La topología de red usada es Feed Forward 2.2 con una capa *Flatten* unida a una capa oculta plenamente conectada, seguida de una capa de activación *Relu* y con una capa de activación lineal tras la capa de salida.

Puesto que esta red es un modelo de regresión, la función de perdida elegida es error cuadrático medio y el optimizador usado es *Adam* con learning rate de 0.01.

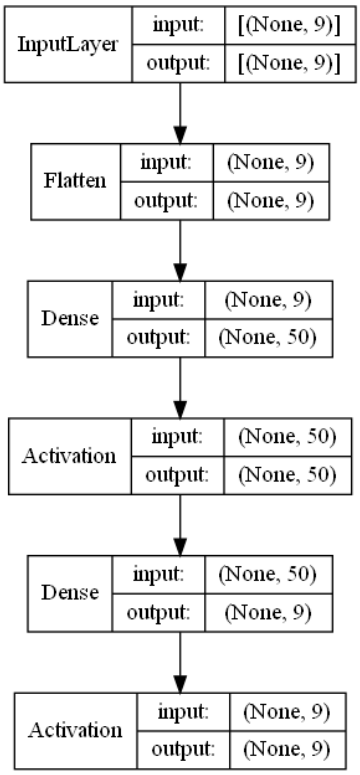

Figura 3.2: Arquitectura de red

3.4.2.2 Limitaciones impuestas a los agentes

La única limitación que ofrece este juego es que no se puede colocar una ficha en una casilla que ya tiene otra, es decir, solo se puede colocar ficha en casillas vacías.

Aunque parece trivial, es un punto importante a tener en cuenta en el diseño de los agentes, ya que parten de 0 su aprendizaje.

El agente de Q Learning ha sido diseñado de manera que (no por un proceso de aprendizaje si no programado a mano) conoce que acciones son ilegales y las evita, centrándose unicamente en aprender estrategias y reduciendo así el numero de filas de la Q-Table ahorrando memoria.

Sin embargo el agente de Deep Q Learning (que gracias al uso de redes no tiene las limitaciones de memoria que tiene el otro) no tiene este conocimiento previo de que acciones son erróneas, y recibiendo recompensas muy negativas cuando las tome debe aprender a evitarlas.

3.4.3 Diseño del experimento

El diseño del proceso de aprendizaje es el mismo en ambos agentes, dos agentes, uno jugando con X y otro con O pero con la misma configuración se enfrentan durante un número de iteraciones o partidas definido, el cual podemos monitorizar porque podemos ver la recompensa media que esta consiguiendo cada uno, la diferencia entre la recompensa media de ambos, la iteración actual y el tiempo que se tarda en jugar 1000 partidas. Se guardan versiones de cada agente durante el aprendizaje como puntos de control por si se para la ejecución por motivos ajenos al experimento y para ver posteriormente su rendimiento y el progreso de aprendizaje.

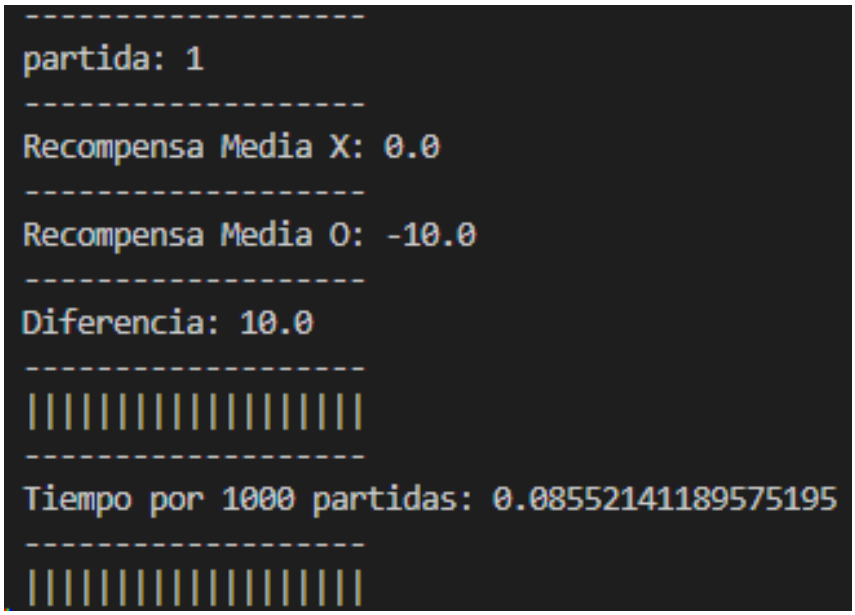

Figura 3.3: Monitorización entrenamiento

Para comparar agentes con distinta configuración una vez han sido entrenados, se genera una gráfica por cada uno, donde usando los puntos de control generados en el aprendizaje, se mide su rendimiento jugando contra un agente que juega de manera aleatoria con la recompensa media obtenida por partida.

3.5 Resolución Risk Simple

3.5.1 Configuración del entorno

El entorno emula las reglas del Risk clásico 1.2, pero para dos jugadores, lo implica las siguientes modificaciones:

1. Existe un tercer jugador neutral que no puede realizar acciones ademas de lanzar los dados para defenderse, es decir, hay un numero de territorios neutrales.
2. Para que se considere que un jugador ha ganado, deberá eliminar al otro jugador, no es necesario eliminar al jugador neutral para ganar la partida.

Además, por razones de rendimiento y recursos, si la partida llega a setenta y cinco turnos sin que nadie gane, se considerará empate. Al ser este el Risk simple, el tablero contiene solo nueve países en un mismo continente, lo cual reduce considerablemente la complejidad del juego.

El estado del entorno se representa con tres arrays binarios de longitud nueve(cada uno de los países) que indican que territorios están ocupados por el jugador al que le toca jugar, el jugador contrario y el neutral en ese orden. Además incluye un array de longitud dos *hot-encoded* que indica si la acción a realizar es decidir si atacar o pasar o si es decidir el ataque a realizar.

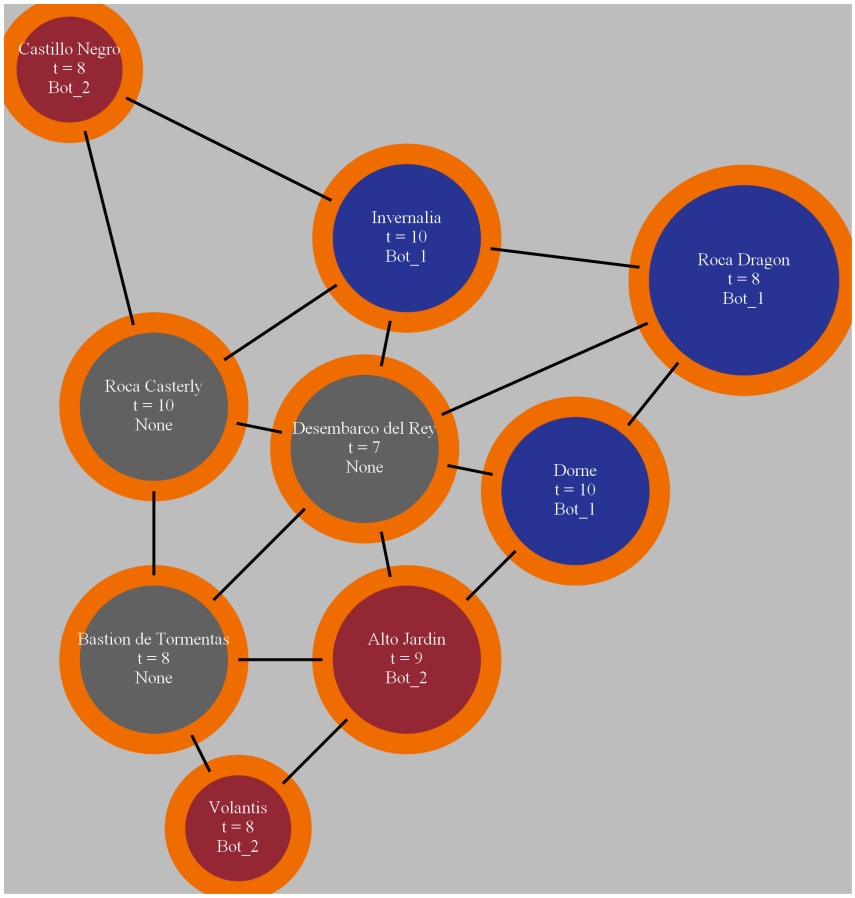

Figura 3.4: Tablero Risk simple

3.5.2 Configuración del agente en el entorno

El Risk es un juego considerablemente mas complejo y profundo que el Tres en Raya, en este juego se pueden realizar numerosas acciones que se agrupan en tres fases del turno: planificación, ataque y fortificación. En este proyecto, nuestro agente aprenderá a dominar la fase de ataque unicamente. Esta fase tiene tres acciones, decidir si atacar o no, seleccionar desde que territorio y a cual atacar, y el número de tropas a usar en dicho ataque.

3.5.2.1 Diseño de las Redes

En un principio, se optó por un agente que con un solo modelo neuronal intentase aprender a hacer las tres acciones, lo cual se descartó pronto, pues tras semanas de pruebas los avances eran nulos, por lo que se decidió dejar para el modelo las dos primeras acciones y el numero de tropas para atacar aleatorio como las acciones del resto de fases. En este punto, el reto era conseguir que el agente aprendiese a discriminar los estados erróneos, pero al igual que con el caso anterior, tras mas de un mes de pruebas sin resultados favorables, se decidió usar una máscara que una vez generada la salida de la red de valor, elimina las acciones ilegales, haciendo así que el agente solo tenga que aprender estrategias para ganar y no las reglas.

Sin embargo, aunque esto es un gran avance, los agentes no consiguen un buen aprendizaje, pues aunque tomen una decisión correcta sobre a donde atacar, como el número de tropas usadas es aleatorio, si este no es adecuado y el ataque fracasa por ello, el agente recibirá una mala recompensa y no debería ser así pues ha tomado una buena decisión.

Para solucionar esto, se dividió el aprendizaje en dos agentes:

1. El agente de tropas. Este agente es el primero en ser entrenado, mientras el resto de acciones se toman de manera aleatoria este aprende a decidir que cantidad de tropas se deberían usar en cada ataque. Las posibles cantidades de tropas a usar son de una a veinte.

   La representación del estado en este agente cambia, quitando el array *hot-encoded* para determinar el tipo de acción y añadiendo otros dos *hot-encoded* de tamaño nueve que indican el territorio desde el que se ataca y al que se ataca. Teniendo así cuarenta y cinco neuronas de entrada.

   La topología de red usada en este agente es Feed Forward 2.2 con una capa *Flatten* unida a una capa oculta plenamente conectada, seguida de una capa de activación *Relu* y con una capa de activación lineal tras la capa de salida.

   La capa de entrada es la representación del estado ya mencionada y la salida es de veinte neuronas, los Q valores de cada acción posible.

   Puesto que esta red es un modelo de regresión, la función de perdida elegida es error cuadrático medio y el optimizador usado es *Adam* con learning rate de 0.01.

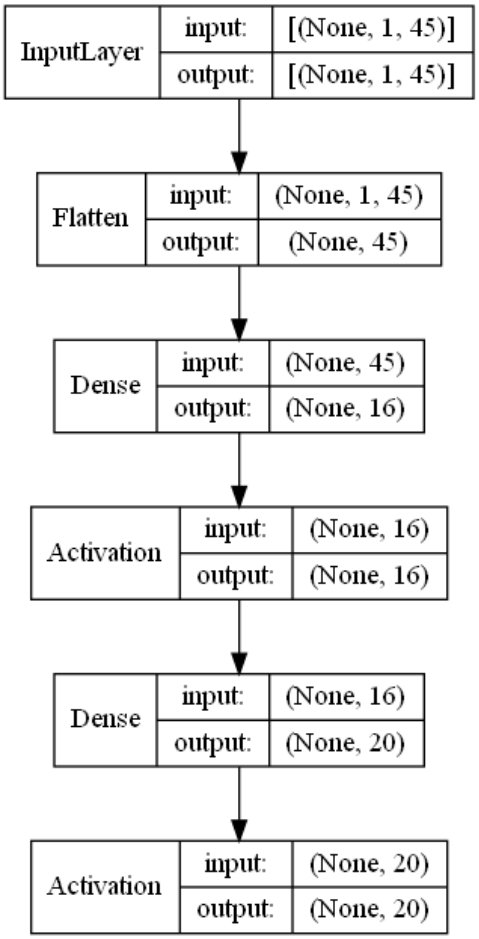

Figura 3.5: Arquitectura de red tropas

2. El agente de ataque. Este agente aprende las otras dos acciones de la fase de ataque, pero usando el agente de tropas ya entrenado para su cometido, evitando así fluctuaciones irregulares en el aprendizaje del segundo agente.

   La topología de red usada en este agente es similar a la anterior exceptuando que la entrada usa la representación estándar del estado, veintinueve neuronas, y las neuronas de la capa de salida, que es de treinta y tres, las dos primeras los Q valores de decidir si se fortifica o ataca, y las restantes los de las parejas territorio atacante y atacado posibles.

   Y puesto que también es un modelo de regresión, la función de perdida elegida es error cuadrático medio y el optimizador usado es *Adam* con learning rate de 0.01.

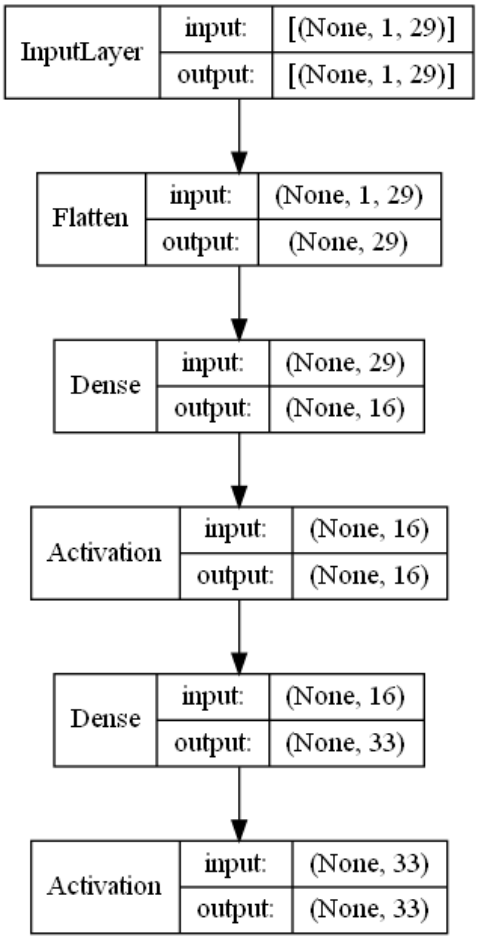

Figura 3.6: Arquitectura de red ataque

3.5.3 Diseño del experimento

El diseño del experimento es el mismo que en el Tres en raya 3.4.3 pero con el entorno del Risk y los agentes ya mencionados.

3.6 Resolución Risk

3.6.1 Configuración del entorno

El entorno de este juego es el mismo que el diseñado para el Risk simple 3.5.1, con una excepción, el tablero esta completo, es decir, tiene cuarenta y dos territorios agrupados en seis continentes. Por lo que el estado se representa con tres arrays de cuarenta y dos de longitud en vez de nueve como antes, y el mismo *hot-encoded* ya mencionado en el simple.

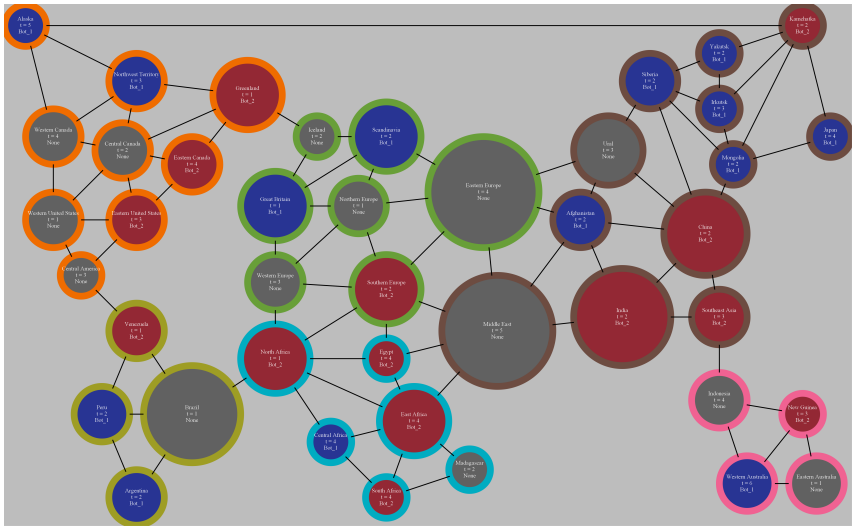

Figura 3.7: Tablero Risk

3.6.2 Configuración del agente en el entorno

Como vimos en 3.5.2, el agente se divide en dos para cubrir todas las acciones de la fase de ataque del turno. El agente que decide el número de tropas y el agente que decide cuando a donde y desde donde atacar.

3.6.2.1 Diseño de las Redes

La principal diferencia respecto al Risk simple es que al haber mas territorios el tamaño de la representación crece y la capa de entrada de las DQNs será mayor. Teniendo así la red de tropas una entrada de doscientas diez neuronas y la red de ataque de ciento veintiocho. La salida de la red de tropas sin embargo no cambia, pues se siguen usando de una a veinte tropas por ataque y finalmente la salida de la red de ataque sin cambia, pues esta depende de las combinaciones de territorio atacante y atacado posibles, que sumándole las dos neuronas de decidir si atacar o no suman ciento sesenta y siete neuronas en la capa de salida.

Como se puede apreciar, por lo demás son iguales a las redes del Risk simple, una capa oculta con activacion *Relu*, ambos modelos de regresión con función de perdida error cuadrático medio y optimizador *Adam* con learning rate de 0.01.

3.6.3 Diseño del experimento

Ver 3.5.3.

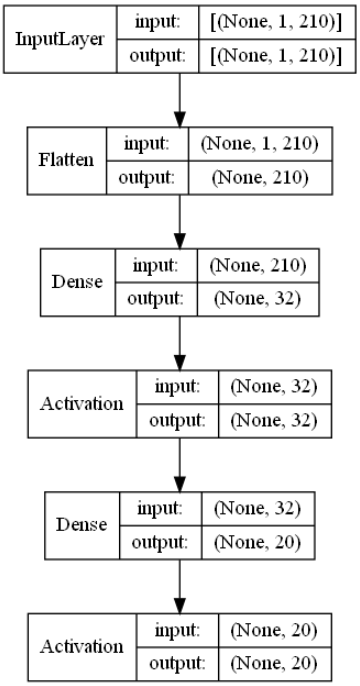

Figura 3.8: Arquitectura de red tropas

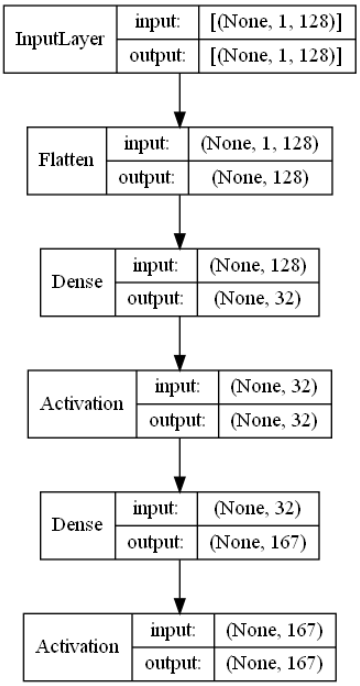

Figura 3.9: Arquitectura de red ataque

Capítulo 4

Resultados

La locura es hacer lo mismo una y otra vez y esperar resultados diferentes.

Albert Einstein

Los resultados de este proyecto han sido limitados por los recursos computacionales y tiempo disponibles.

4.1 Introducción

En este capítulo se exponen los resultados obtenidos tras aplicar las metodologías del experimento descrito anteriormente incluyendo el diseño de los agentes, la optimización de los hiperparámetros, elección y *tuning* de las arquitecturas de red, el desempeño de los agentes en sus respectivos entornos y con esto ultimo la comparativa de ambos algoritmos.

4.2 Tres en raya Q Learning

Los agentes de Q Learning se diferencian entre si por sus hiperparámetros 3.2.0.1 y variando estos se han conseguido 6 agentes que resuelven la tarea asignada. Puesto que todos resuelven la tarea con una eficacia similar, la diferenciación y selección del mejor agente se centra en la eficiencia, es decir, en la velocidad a la que aprenden.

A continuación se ven varias tablas ordenadas según velocidad de aprendizaje para cada hiperparámetro excepto el factor de aprendizaje, que no varía entre los agentes diseñados.

Tabla 4.1: Factor de Descuento

Factor de descuento	Recompensa media tras 8000 partidas	Agente
0.68	95 %	1 y 2
0.4	>90 %	3,4,5 y 6

Tabla 4.2: Factor de Exploración

Factor de exploración	Recompensa media tras 8000 partidas	Agente
0.15	95 %	1
0.5	92 %	2 y 3
0.1	90 %	4
0.05	89 %	5
0.001	87 %	6

En cuanto al valor de Q Inicial, se comenzaron los experimentos usando 0 como valor para este hiperparámetro, lo que causó un curioso comportamiento en el agente. Este se puso a la defensiva, es decir, en vez de aprender a ganar partidas, aprendió a evitar que las ganase el contrincante generando asi una política empatista, lo cual es un resultado interesante pero no es el que se busca en este proyecto por lo que el valor de Q inicial se cambió a 6 para solucionar este problema.

No es complicado dilucidar viendo las tablas que los agentes 1 y 2 son los que mejor rendimiento otorgan ya que ocupan los primeros puestos de ambas tablas, y tienen gráficas de recompensa media muy similares.

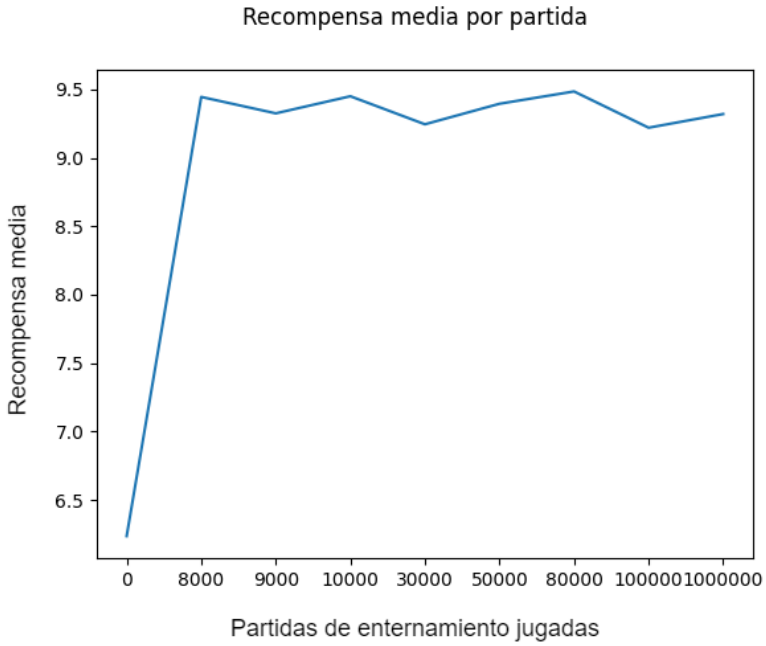

Figura 4.1: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 1

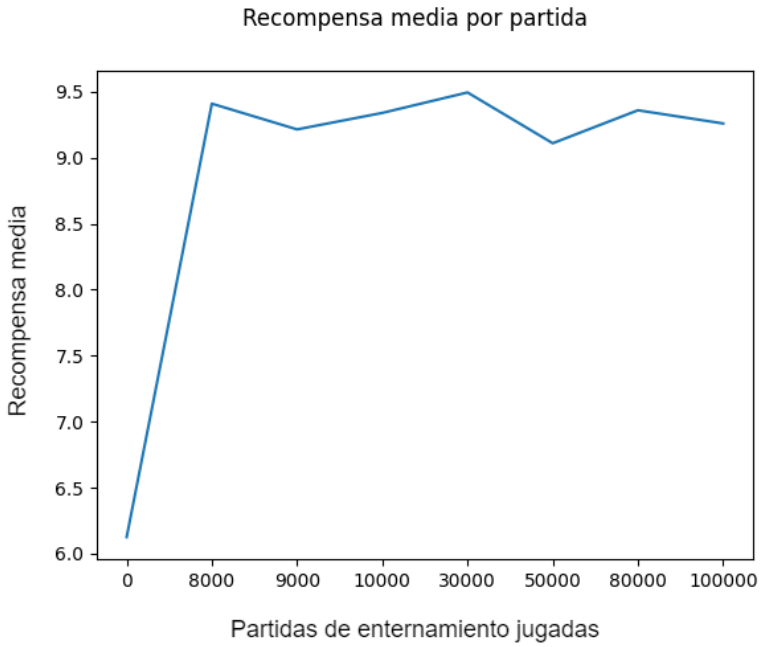

Figura 4.2: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 2

Como se puede ver, todos los agentes entrenados han obtenido excelentes resultados llegando todos a mas de un 90 por ciento de la recompensa máxima disponible. Los tiempos de aprendizaje han sido bastante bajos debido a la ínfima longitud máxima que tienen las partidas de este juego y a su baja complejidad, lo que ha permitido que en menos de 10000 partidas los mejores agentes se acerquen al 95 por ciento de la recompensa máxima. Entre todos estos agentes, los que mejor rendimiento dan y mas rápido han llegado a este son el primero y el segundo, cuyo punto en común es el factor de descuento y difiriendo en el factor de exploración. Además el tercer agente tiene el mismo factor de exploración que el segundo agente pero menor factor de descuento y sus resultados son notablemente peores. Por lo que podemos concluir que el factor de descuento es el hiperparámetro con mayor relevancia en la resolución del Tres en raya con Q Learning.

Recordando también el incidente del valor Q inicial 0 que ocasionaba que el agente fuese *empatista*, no es complicado ver a que se debía esto. Como vimos en la teoría del Q Learning 2.3.2, este valor se usa cuando se explora un estado desconocido, y en este entorno, la recompensa por victoria es 10, por empate 5 y por derrota 0. Si la Q inicial es 0, los estados desconocidos serán menos favorables que los de empate, por lo que el agente en vez de explorar buscando la victoria, se *conforma* con empatar, por ello aumentar la Q inicial a 6 evita que el agente prefiera los empates y busque ganar.

4.3 Tres en raya Deep Q Learning

Puesto que el agente uno de Q Learning consigue excelentes resultados, todos los agentes de Deep Q Learning diseñados usarán su misma configuración en hiperparámetros, siendo la arquitectura de red la única variable entre estos agentes.

Tabla 4.3: Hiperparámetros fijos copiados del agente de Q Learning con mejor rendimiento

Factor de exploración	Factor de descuento	Learning rate
0.15	0.68	0.25

Tras numerosos diseños y mediante prueba y error se consiguió diseñar un agente que consigue desenvolverse perfectamente en el entorno. Las capas ocultas de la red de este agente son dos de cincuenta neuronas seguidas ambas por capas de activación *Relu*

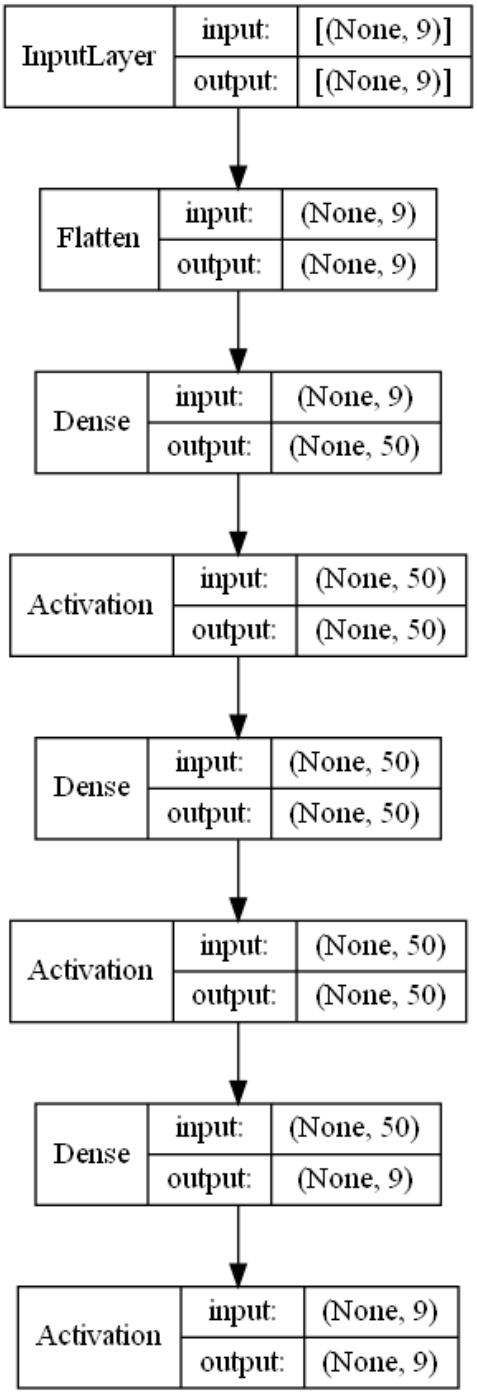

Figura 4.3: Arquitectura de red del agente 1

El segundo y último agente diseñado para este entorno tiene una versión simplificada de la red del anterior, pues este solo tiene una capa oculta de cincuenta neuronas seguida de una de activación *Relu*.

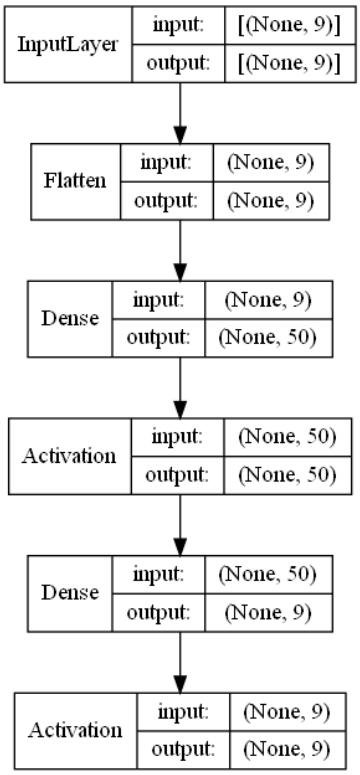

Figura 4.4: Arquitectura de red del agente 2

Ambos agentes presentan un rendimiento similar como se puede apreciar en estas gráficas de recompensa media.

Como hemos podido apreciar en las gráficas de recompensa media de los agentes de Deep Q Learning uno 4.5 y dos 4.6, ambos consiguen un rendimiento similar. Si analizamos en profundidad cualquiera de estas gráficas, podemos ver(como cabía esperar) que al principio el agente obtiene recompensas negativas pues no conoce el entorno y juega aleatóriamente, y como a partir del punto de control con veinte mil partidas, el agente ha aprendido a evitar las acciones ilegales y las recompensas no descienden del 50 por ciento del máximo disponible, alcanzando unos pocos puntos de control mas adelante el 90 por ciento.

Al tener rendimientos tan similares y siguiendo uno de los principios mas elementales del diseño de redes neuronales, el agente seleccionado es el que tiene la arquitectura de red mas simple, pues con menos recursos consigue el mismo resultado que el otro.

Es interesante también ver la siguiente gráfica 4.7 que muestra la recompensa media por partida enfrentando a dos agentes con DQNs que se han entrenado juntos, pues en ella se puede ver la dinámica del aprendizaje.

En esta se aprecia como inicialmente ambos agentes no conocen que acciones son ilegales aun, pero cuando estos llegan a treinta mil partidas, se puede ver como el agente que juega con las *O*s gana casi todas las partidas lo que obliga al otro agente a explorar nuevas jugadas cayendo a acciones ilegales. Pero tras encontrar una estrategia que gane a su rival el agente que juega con las *X*s adelanta al otro, obligándole así a encontrar mejores estrategias ...

Así esta gráfica demuestra que la dinámica de entrenamiento y aprendizaje que pretende conseguir el Q Learning y Deep Q Learning se cumple.

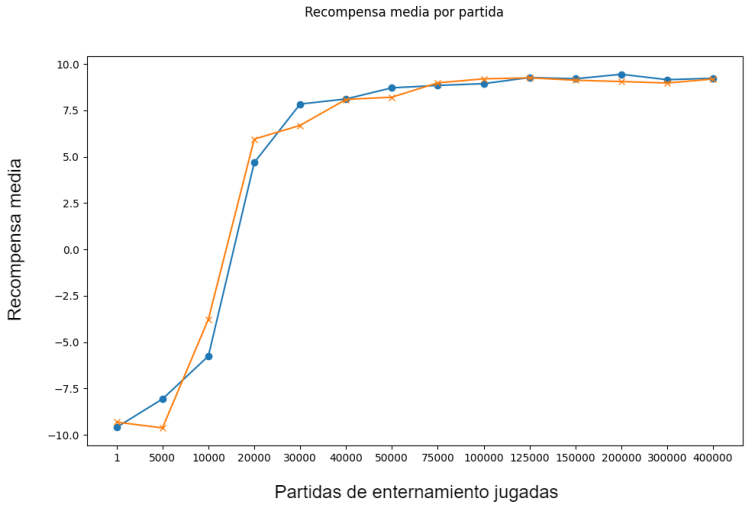

Figura 4.5: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 1. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O

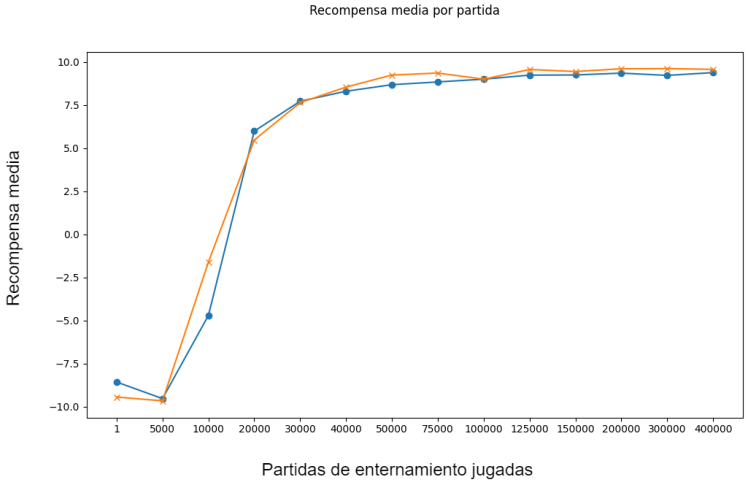

Figura 4.6: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente 2. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O

En cuanto al propio entrenamiento de la red, hemos comprobado que entrenarla cada cincuenta iteraciones en vez de cada partida disminuye los tiempos de entrenamiento notablemente y aumentan el rendimiento del modelo, pues no se generan suficientes datos como para que no se repitan demasiado en entrenamientos contiguos lo cual provoca sobreajuste. La red Target copia los pesos de la Value cada cien iteraciones y para el entrenamiento se usa un batch de tamaño treinta y dos.

En la figura 4.8 podemos ver una partida completa entre dos agentes, en la que se aprecia como han aprendido a jugar perfectamente al Tres en Raya. El agente con las X opta por empezar por la esquina por lo que el de las Os juega con el centro, desde este momento se puede apreciar como ambos agentes van cerrando las posibles lineas de tres a su oponente mientras intentan hacer las suyas propias, indistinguible del juego humano.

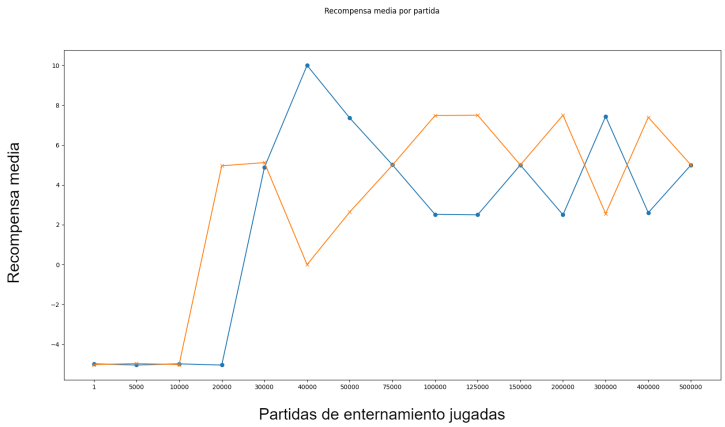

Figura 4.7: Recompensa media por partida en cada *checkpoint* de entrenamiento enfrentando dos agentes entrenados juntos. Donde la linea naranja representa el agente entrenado para jugar con la ficha X y la azul el entrenado para jugar con la ficha O

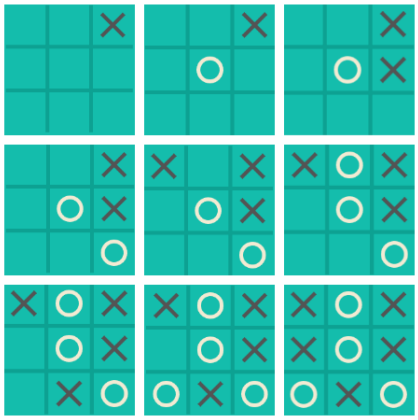

Figura 4.8: Partida completa entre dos agentes

4.3.1 Q Learning vs Deep Q Learning

Tras haber resuelto el mismo problema con ambas tecnologías, podemos apreciar que los resultados que ofrecen ambas son muy similares en eficacia, ambos tipos de agente han alcanzado porcentajes de recompensa máxima cercanos al 100 por ciento. Sin embargo, como cabía esperar en términos de tiempo de aprendizaje, los agentes con DQNs requieren periodos de aprendizaje muy largos en comparación con los agentes con Q Table.

Teniendo clara esta comparativa, podemos concluir que aunque las redes neuronales son un gran añadido al Q Learning tradicional, solo son mas eficientes en entornos con un gran espacio de estados, sin embargo en entornos pequeños como el del Tres en Raya (9!), es mas eficiente el uso de la Q Table

4.3.2 Diseño y programación desde cero de los agentes

En cuanto al diseño y programación de estos agentes, aunque como hemos visto cumplen su función a la perfección, se pueden mejorar las practicas de programación utilizadas y optimizar en gran medida el funcionamiento de estos tanto en términos de velocidad de ejecución, como de espacio y recursos consumidos.

4.4 Risk Simple Deep Q Learning

Como hemos visto en 3.5, ha habido varios intentos de agente a lo largo del proyecto. Del primer tipo de agente, el mas generalista con un modelo que pretendía cubrir todas las acciones ni si quiera se generaron gráficas de rendimiento, pues solo con la monitorización en vivo del entrenamiento se apreciaba que no funcionaba.

Sin embargo del agente que aprende fijando el numero de tropas aleatoriamente se diseñaron varios agentes que difieren en el número de neuronas de la capa oculta, uno de dieciséis y otro de veinticuatro. En las figuras 4.9 y 4.10 se muestran sus gráficas de rendimiento contra un agente plenamente aleatorio.

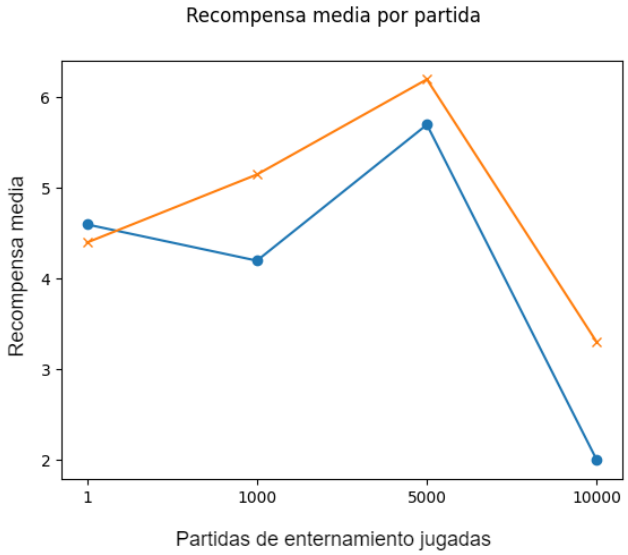

Figura 4.9: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente con 16 neuronas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

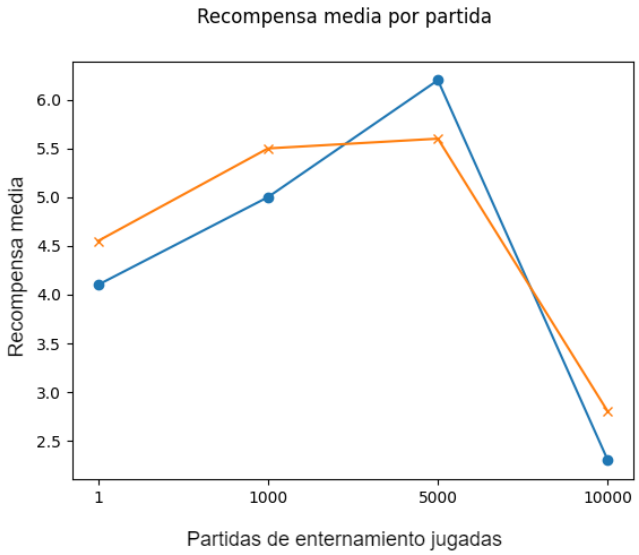

Figura 4.10: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente con 24 neuronas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

Como hemos podido apreciar en estas gráficas, los agentes que seleccionan aleatoriamente las tropas para atacar no solo no consiguen aprender a jugar al Risk, sino que llegan a recompensas muy bajas contra agentes puramente aleatorios.

Debido a su diseño fiel al Deep Q Learning, el agente decide si atacar o no aleatoriamente con una probabilidad del cincuenta por ciento al principio, lo que le hace conseguir empatar casi siempre, pues solo refuerza sus territorios por lo que cada turno es mas complicado conquistarle alguno,lo cual se puede apreciar en sus gráficas como por ejemplo 4.9, viendo que al principio los resultados son los esperados. Sin embargo, a medida que el aprendizaje avanza, el agente se vuelve agresivo, y es ahí donde se aprecia que el aprendizaje no esta siendo correcto, ya que la recompensa media disminuye incluso por debajo del cincuenta por ciento. Es decir, jugando peor que al azar.

Los agentes partícipes del modelo de agente que primero aprende a decidir el número de tropas y después el destino y origen de los ataques además de decidir si fortificar o continuar atacando, tienen ambos dieciséis neuronas en la capa oculta dando lugar a las siguientes arquitecturas 4.11 4.12.

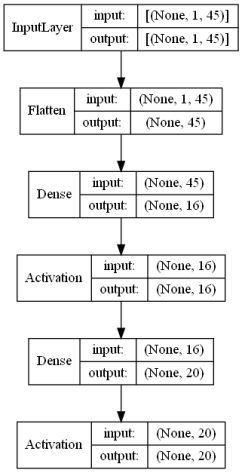

Figura 4.11: Arquitectura de red tropas

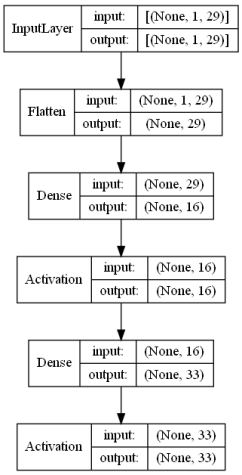

Figura 4.12: Arquitectura de red ataque

Ambos agentes, tanto el de tropas como el de ataque consiguen un buen aprendizaje como se puede ver en las siguientes gráficas de rendimiento contra un agente plenamente aleatorio 4.13 4.14.

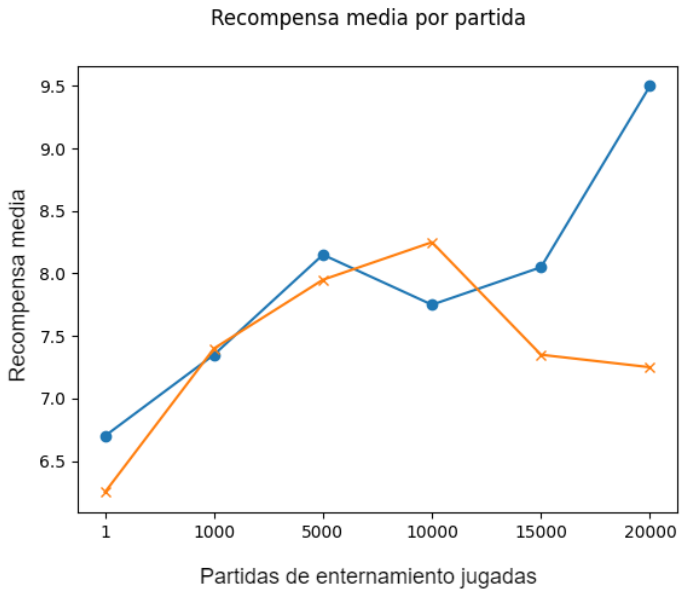

Figura 4.13: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de tropas. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

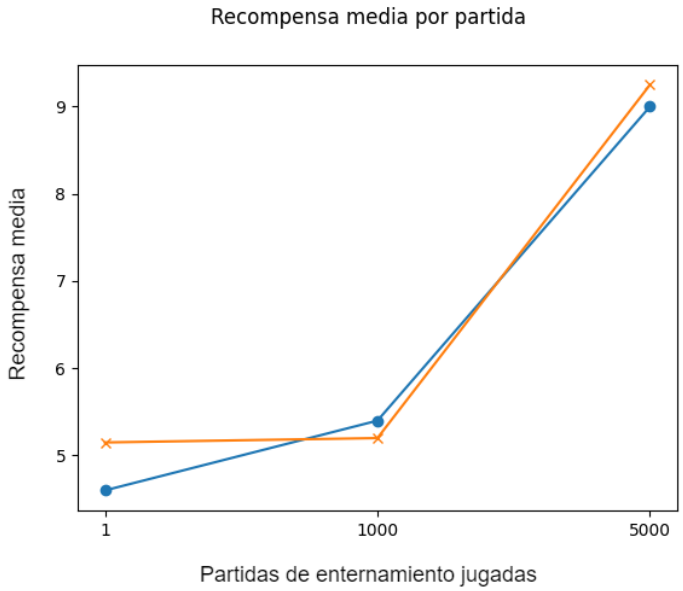

Figura 4.14: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de ataque. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

Cabe destacar que el propio agente que aprende a seleccionar el número de tropas para atacar ha demostrado una gran capacidad de aprendizaje como se aprecia en 4.13, llegando al noventa por ciento de la recompensa máxima tras veinte mil iteraciones. Sin embargo, esto no es lo mas interesante de este agente, el rendimiento tan superior de este respecto al aleatorio teniendo en cuenta la tarea tan *pequeña* que este realiza se debe en gran parte a que ha aprendido a hacer trampas.

En efecto el agente ha encontrado un error de programación en el entorno y ha aprendido a sacarle partido. El programador de esta implementación del Risk incluía el agente aleatorio, el cual cuando va a decidir cuantas tropas usar en un ataque, selecciona al azar un numero entre el uno y el máximo menos uno del territorio de origen del ataque. Por ejemplo para atacar desde un territorio con cinco tropas, seleccionará un numero al azar entre uno y cuatro. Puesto que el agente aleatorio no puede sobrepasar el limite de tropas del país, el propio Risk no considera la excepción de que este lo haga por lo que nuestro agente podría sobrepasar ese limite y en efecto lo hace. De esta manera nuestro agente puede atacar con cinco tropas desde un territorio de cinco tropas y dejar territorios propios a cero tropas sin perder su custodia, consiguiendo una ventaja en número de tropas sobre el rival.

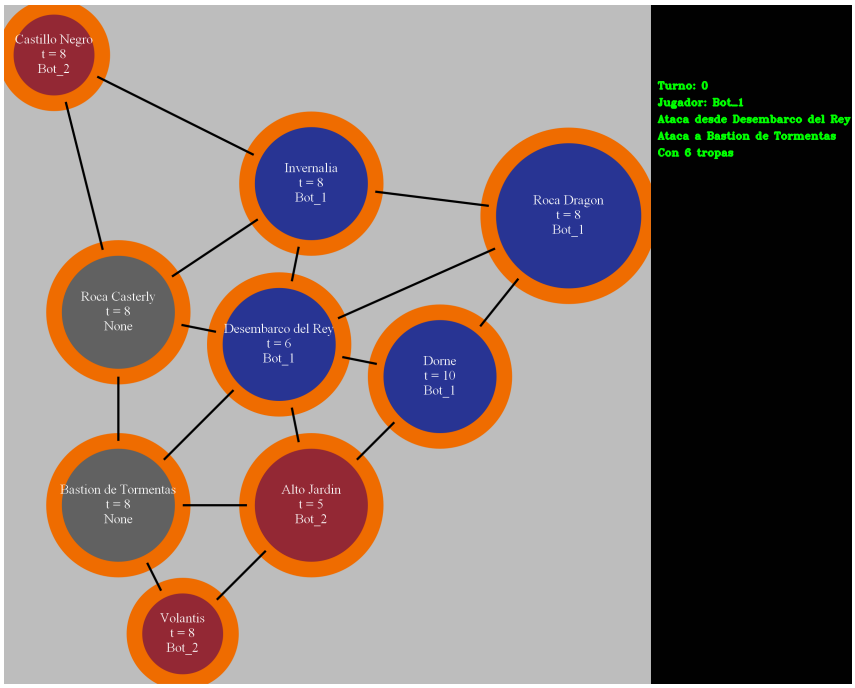

Figura 4.15: Captura del agente aprovechando el error de programación

En esta imagen del tablero 4.15 se puede ver como el agente ataca desde Desembarco del Rey usando el total de tropas que este contiene. Si en este ataque se conquista el territorio destino, todas las tropas se moverían a este dejando a 0 Desembarco del Rey.

Partiendo de este nivel de aprendizaje, es sencillo que el agente que aprende el resto de acciones del ataque consiga buenos resultados y asi lo hace. Como podemos ver en 4.14 en tan solo cinco mil partidas consigue un noventa por ciento de la recompensa máxima, y teniendo en cuenta que este juego incluye un proceso aleatorio que no depende de los agentes en los ataques, el lanzamiento de dados, podemos concluir que hemos conseguido unos resultados cerca de lo inmejorable.

A pesar de que el Risk es un juego muy superior en complejidad al Tres en Raya, teniendo un espacio de estados infinito y multitud de acciones diferentes a realizar. Los modelos que lo resuelven y las iteraciones necesarias para ello son considerablemente menores que las del otro juego. Lo cual se debe a que en el

Tres en Raya se aprende a evitar estados ilegales, lo cual constituye la mayor parte del aprendizaje, y al contrario en el Risk no es necesario gracias a la máscara que evita acciones ilegales.

4.5 Risk Completo

Tras varias, probablemente insuficientes pruebas debido a la falta de tiempo se han conseguido diseñar dos agentes que se desenvuelven con propiedad en el entorno, el de tropas y el de ataque.

Tanto el agente que primero aprende a decidir el número de tropas como el que decide el destino y origen de los ataques además de decidir si fortificar o continuar atacando, tienen ambos treinta y dos neuronas en la capa oculta dando lugar a las siguientes arquitecturas 4.16 4.17 y gráficas de rendimiento 4.18 4.19, donde se aprecia un rendimiento mejor del esperado.

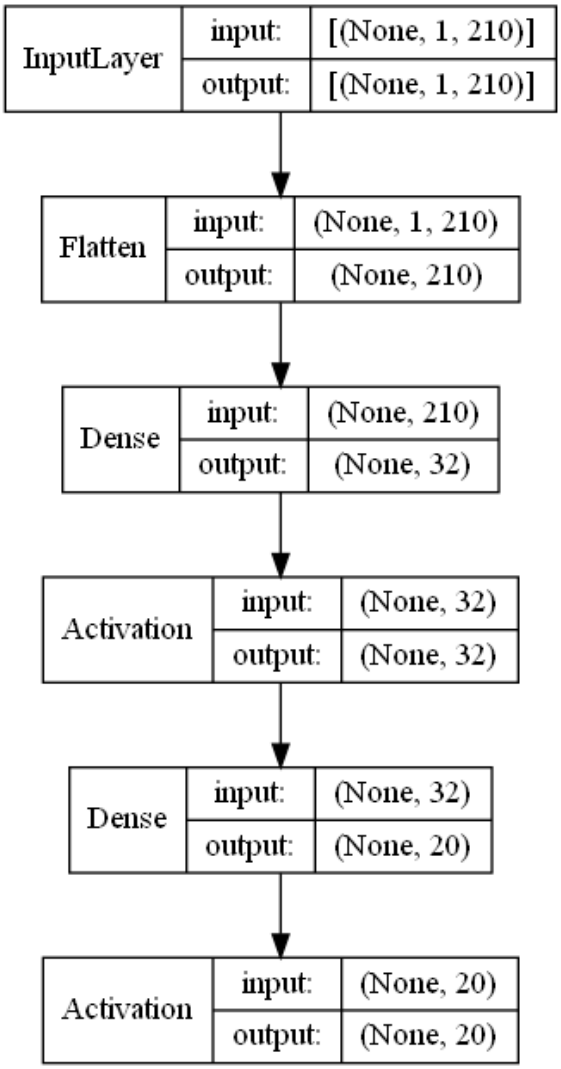

Figura 4.16: Arquitectrua de red de tropas

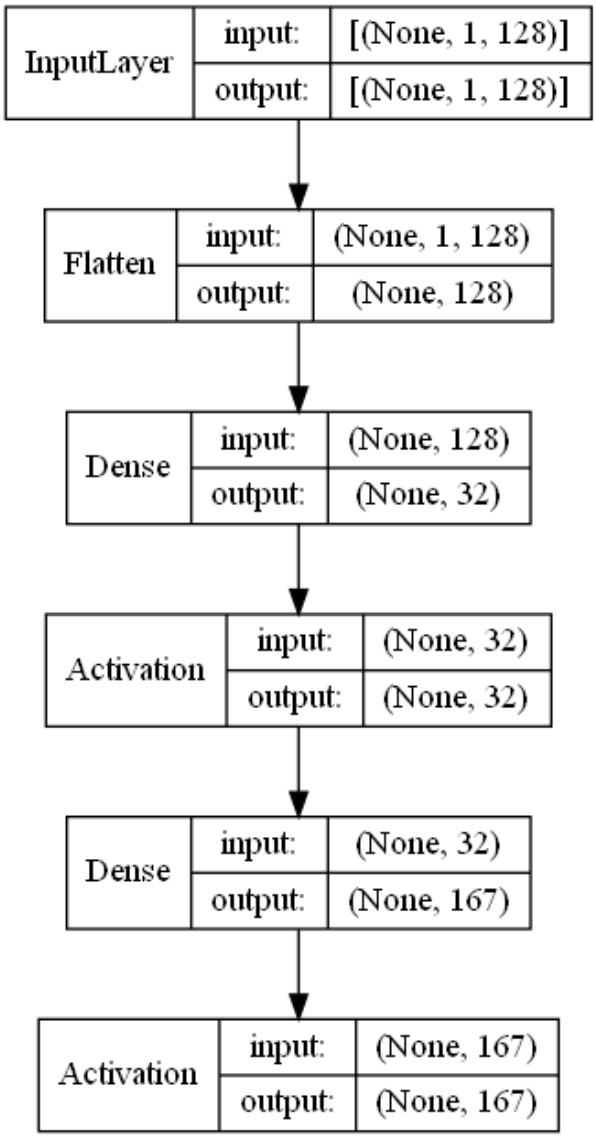

Figura 4.17: Arquitectrua de red de ataque

Tras la dificultades y el tiempo requerido para encontrar modelos que resolviesen el Risk simple, era complicado creer que se podría resolver el completo con esta tecnología. Y aunque en los resultados 4.5 se puede ver que contra un agente aleatorio, el rendimiento es incluso mayor que en el Risk simple, esto se debe a un error de enfoque.

En efecto, las gráficas indican un rendimiento de mas del ochenta por ciento de la recompensa máxima en tan solo mil partidas, mas rendimiento que el anterior agente en un entorno mas complejo y con menos iteraciones solo entrenando el agente de tropas. Este extraño suceso no se debe a nuestro agente, sino al contrincante y se puede apreciar también en el tres en raya. A medida que el entorno crece en complejidad, el juego aleatorio se vuelve cada vez menos efectivo, lo que ocasiona que se necesiten mas iteraciones para que un agente del Tres en Raya supere al uno aleatorio, que en el Risk simple y mas en este que en el completo. Por este motivo aparentemente el Risk completo se resuelve con facilidad pero si se ve una partida en directo con este agente, se aprecia que el comportamiento es prácticamente aleatorio e incluso errático en ocasiones.

Como era de esperar, los recursos computacionales disponibles para este proyecto han jugado en contra de este desde el principio.

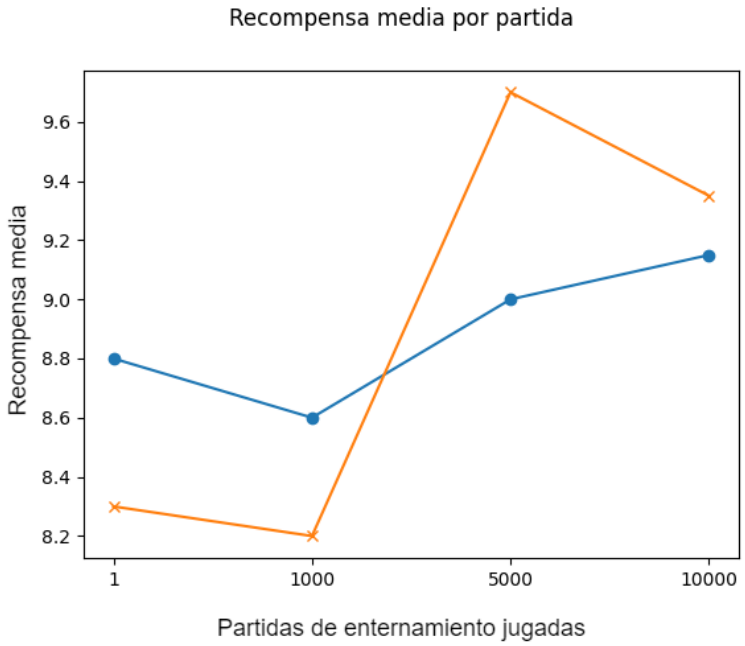

Figura 4.18: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de tropas del Risk completo. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

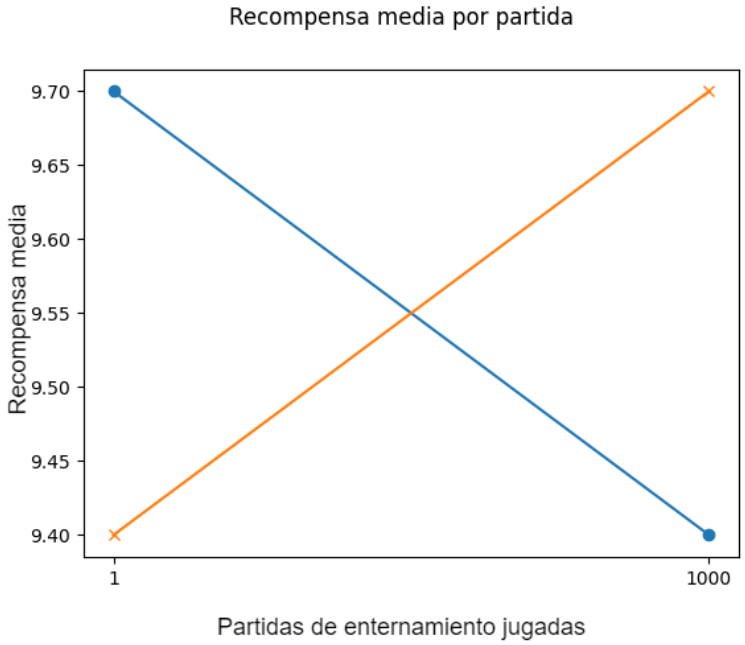

Figura 4.19: Recompensa media por partida en cada *checkpoint* de entrenamiento del agente de ataque del Risk completo. Donde la linea naranja representa el jugador 1 y la azul el jugador 2

Capítulo 5

Conclusiones y líneas futuras

El uso de juegos de mesa para realizar un acercamiento a la metodología de aprendizaje que propone el Aprendizaje Reforzado ha sido bastante esclarecedor. Siguiendo las directrices impuestas por esta hemos diseñado con éxito agentes que superan el rendimiento de políticas de juego aleatorias en el Tres en Raya, el Risk Simplificado y el Risk completo.

Durante este proceso hemos podido adentrarnos en la teoría detrás del Q Learning y del Deep Q Learning, descubriendo el peso e importancia que una buena configuración de hiperparametros tienen en el desempeño de un agente, asi como la arquitectura de red y en el juego del Risk y la selección previa de que acciones aprenderá el agente y cuales no. Hemos podido comprobar que un agente de RL aprende a maximizar recompensas encontrando soluciones que escapan a la comprensión humana del entorno, como el caso en el que el agente aprende a hacer trampas.

En un principio se intentaron diseñar agentes con modelos muy amplios que abarcaban en una sola red la información para todos los tipos de acción, tras el fracaso de ello se ha dividido el agente en varios, el de tropas y el de ataque, consiguiendo así unos resultados excelentes, por lo que podemos concluir que la modularización facilita el aprendizaje y mejora el desempeño de los agentes.

Y por ultimo y aunque no entraba en los planes iniciales del proyecto, hemos podido ver como los resultados que ofrece un agente se ven sumamente condicionados por la vara con la que se mide. Es decir, que para comprobar el rendimiento de un agente competitivo, se debe tener muy en cuenta contra quien compite este.

Tras haber diseñado todos estos agentes, tanto de Risk como de Tres en Raya, seria interesante medir su rendimiento con agentes mas inteligentes que el aleatorio como por ejemplo programados a mano, con arboles de decisión o redes neuronales evolutivas. Es casi seguro que el rendimiento del agente de Risk completo decaerá como hemos postulado en las conclusiones.

En cuanto a la resolución del Risk completo, una a nivel humano como mínimo, sería interesante utilizar algoritmos de aprendizaje reforzado mas potentes y complejos como Alpha Go y sus derivados o incluso algún tipo de algoritmo de redes neuronales evolutivas, que están ganando mucho peso en el diseño de agentes que resuelven juegos de ordenador.

Bibliografía

[1] R. Asale and Rae, "Inteligencia: Diccionario de la lengua española." [Online]. Available: https://dle.rae.es/inteligencia

[2] A. M. Turing, "Computing machinery and intelligence," *Parsing the Turing Test*, p. 23?65, 2007.

[3] F. ROSENBLATT, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," *Principles of Neurodynamics. Perceptrons and the theory of Brain Mechanisms*, 1965.

[4] M. Minsky and S. Papert, "An introduction to computational geometry," *Cambridge tiass., HIT*, vol. 479, p. 480, 1969.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[6] DeepMind, "Deep reinforcement learning." [Online]. Available: https://www.deepmind.com/blog/deep-reinforcement-learning

[7] M. Sonesson, "Creating an ai for risk board game," Jan 2018. [Online]. Available: https://martinsonesson.wordpress.com/2018/01/07/creating-an-ai-for-risk-board-game/

[8] E. Blomqvist, "Playing the game of risk with an alphazero agent," *Playing the Game of Risk with an AlphaZero Agent*, p. 1?58, 2020.

[9] A.-a. Arman-Aminian, "Arman-aminian/risk-game-ai-agent." [Online]. Available: https://github.com/arman-aminian/risk-game-ai-agent

[10] J. Torres, "Métodos value-based: Deep q-network," May 2021. [Online]. Available: https://medium.com/aprendizaje-por-refuerzo/9-m%C3%A9todos-value-based-deep-q-network-b52b5c3da0ba

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *Playing Atari with Deep Reinforcement Learning*, p. 1?9, Dec 2013.

[12] F. S. Caparrini, "Aprendizaje por refuerzo: Algoritmo q learning." [Online]. Available: http://www.cs.us.es/~fsancho/?e=109

[13] Na8, "Aprendizaje por refuerzo," Dec 2020. [Online]. Available: https://www.aprendemachinelearning.com/aprendizaje-por-refuerzo/

[14] M. Silva, "Aprendizaje por refuerzo: Procesos de decisión de markov?-?parte 1," Jun 2019. [Online]. Available: https://medium.com/aprendizaje-por-refuerzo-introducci%C3%B3n-al-mundo-del/aprendizaje-por-refuerzo-procesos-de-decisi%C3%B3n-de-markov-parte-1-8a0aed1e6c59#:~:text=Los%20MDP%20(Procesos%20de%20Decision,el%20RL%2C%20donde%20el%20medio

[15] N. Na8, "Breve historia de las redes neuronales artificiales," Sep 2018. [Online]. Available: https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/

[16] T. S. e. E. D. y. M. e. S. d. l. T. I. d. s. i. e. e. e. d. I. L. C. d. T. Fran Ramírez Ingeniero/Grado en Informática de Sistemas, "Historia de la ia: Frank rosenblatt y el mark i perceptrón, el primer ordenador fabricado específicamente para crear redes neuronales en 1957," Jun 2021. [Online]. Available: https://empresas.blogthinkbig.com/historia-de-la-ia-frank-rosenblatt-y-e/

[17] IBM. [Online]. Available: https://www.ibm.com/docs/es/spss-modeler/saas?topic=networks-neural-model

[18] X. Islas, "?? ¿qué es un perceptrón?: Inteligencia artificial [2021]." [Online]. Available: https://www.crehana.com/blog/desarrollo-web/que-es-perceptron-algoritmo/#que-es-perceptron

[19] A. Baranovskij, "Multi-output model with tensorflow keras functional api," Dec 2020. [Online]. Available: https://towardsdatascience.com/multi-output-model-with-tensorflow-keras-functional-api-875dd89aa7c6#:~:text=Keras%20functional%20API%20provides%20an,models%20and%20improve%20code%20quality.

[20] N. en crecimiento, "¿qué es la inteligencia?" Aug 2019. [Online]. Available: https://neuropediatra.org/2016/02/01/que-es-la-inteligencia/

[21] Oracle, "¿qué es la inteligencia artificial (ia)?" [Online]. Available: https://www.oracle.com/es/artificial-intelligence/what-is-ai/

[22] I. Corporativa, "¿somos conscientes de los retos y principales aplicaciones de la inteligencia artificial?" [Online]. Available: https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial

[23] R. Redacción, "La relación entre la inteligencia artificial y los juegos de mesa," Nov 2020. [Online]. Available: https://www.elmagacin.com/la-relacion-entre-la-inteligencia-artificial-y-los-juegos-de-mesa/

[24] Hasbro, "Reglas risk para 2-5 jugadores."

[25] Otravolta, "Albert lamorisse," Jun 2021. [Online]. Available: https://es.wikipedia.org/wiki/Albert_Lamorisse

[26] WikiDasher, "Risk," Apr 2022. [Online]. Available: https://es.wikipedia.org/wiki/Risk

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá