
TRABAJO FIN DE GRADO

# Diseño de un agente inteligente para videojuegos

Realizado por

**Manuel Jesús Domínguez Gómez**

**Para la obtención del título de**

Grado en Ingeniería Informática en Sistemas de Información

**Dirigido por**

Raúl Giráldez Rojo

**Código del proyecto**

21-22-A6

# AGRADECIMIENTOS

Quiero agradecer a mi tutor Raúl Giráldez por la ayuda que me ha aportado en el TFG.

A mis padres, que me han aguantado durante estos largos años de carrera y sin ellos no podría haber realizado este trabajo.

A mis amigos, por todo el apoyo que me han ofrecido y a mis compañeros de carrera, sin los cuales no hubiéramos aprobado tantas asignaturas sin nuestro trabajo en equipo.

# RESUMEN

La industria de los videojuegos se ha convertido en una de las industrias más importantes a nivel mundial, por delante de otras industrias de entretenimiento como la música o el cine. La inteligencia artificial ha tenido un papel destacado en esta industria, permitiendo que una maquina analice información del juego, aprenda y emule el comportamiento de un ser humano. Este trabajo de fin de carrera enmarca este contexto, desarrollando 2 agentes inteligentes capaces de jugar de manera autónoma, bien como único jugador o enfrentado a un oponente humano.

El primer agente, denominado PATIG (anagrama de IAPTG, Intelligent Agent Plays Tabletop Games), es capaz de jugar al 3 y al 4 en raya. Para ello, se ha utilizado el algoritmo minimax para el calcula y la puntualización de todos los tableros, con el fin de escoger el mejor movimiento actual.

El segundo agente, que hemos denominado GRAPI (anagrama de IAPRG, Intelligent Agent Plays Retro Games), es capaz de jugar al videojuego Pong y al videojuego del dinosaurio de Google. Para ello, utiliza el algoritmo NEAT, con el cual es capaz de entrenar diferentes redes neuronales para obtener 1 que sea capaz de jugar a un alto nivel al videojuego.

Para implementar ambos agentes, ha sido necesario desarrollar los cuatro juegos de manera paralela, para que nuestros agentes sean capaces de realizar movimientos en el juego y tengas todos los datos que proveen dichos juegos a su disposición.

Los resultados experimentales los hemos dividido en 4 bloques y hemos estudiado la calidad del agente y su tiempo de respuesta.

**Indice General**

## Indice de figuras

## Indice de códigos

# 1. Introducción

Los videojuegos han tenido como principal objetivo el entretenimiento y ofrecer una experiencia al jugador. A lo largo de los años, con el avance de la tecnología, hemos observado como los videojuegos han ido evolucionando. Hoy en día, muchos usuarios consideran los videojuegos como una nueva forma de arte y como un campo de entrenamiento usados para investigación y en simulaciones.

Los investigadores utilizan el entorno de un videojuego para el desarrollo de la inteligencia artificial. Estos desarrolladores tienen que conseguir inteligencias artificiales realistas, es decir, que realicen acciones que cualquier persona pueda realizar con el mismo videojuego o presentar un reto al jugador.

Como veremos más adelante, los videojuegos no son iguales, y tienen mecánicas diferentes entre ellos. Por ello, se han desarrollado algoritmos diferentes para conseguir que los agentes sean capaces de jugar al videojuego. En este contexto, 2 algoritmo han ganado popularidad estos últimos años. El primero de ellos, el algoritmo minimax, es un algoritmo usado en agentes con un enfoque en los juegos de mesa que estén solucionados. El objetivo de este algoritmo es el cálculo de todas las posiciones posibles en una partida, para posteriormente elegir el mejor movimiento. El segundo algoritmo, el algoritmo NEAT, es muy usado en agentes de videojuegos. El algoritmo NEAT permite el entreno de diferentes redes neuronales para encontrar una red que permita al agente jugar al videojuego de manera óptima.

## 1.1 Objetivos

El objetivo de este trabajo es:

- Desarrollar dos sistemas inteligentes capaces de aprender y jugar a diferentes juegos y de presentar un reto al jugador contrario. El

primero de dichos agentes lo denominaremos PATIG (anagrama de IAPTG, Intelligent Agent Plays Tabletop Games), y que jugara a los conocidos juegos 3 en raya y 4 en raya. PATIG aplica MINIMAX para calcular cada movimiento enfrentándose a un oponente humano.

- El segundo agente inteligente está basado en el algoritmo NEAT y lo denominamos GRAPI (anagrama de IAPRG, Intelligent Agent Plays Retro Games). En este caso, el propósito del agente es aprender y jugar a los juegos DinoGoogle y Pong. Nótese que, el primero de los juegos (DinoGoogle) es un juego de un solo jugador, mientras que el segundo (Pong) enfrenta a la maquina contra un oponente.

- Aunque no es objetivo en sí de este trabajo, es importante destacar la necesidad de desarrollar los cuatro videojuegos, de manera que el agente pueda tener la información del estado actual del juego en cada instante y calcular las siguientes jugadas.

- Por último, con el propósito de analizar el comportamiento, así como la calidad y el tiempo de respuesta de nuestras propuestas, se han realizado una batería de experimentos para cada agente.

# 2. Estado del arte

## 2.1 Videojuegos

### 2.1.1 Historia

El inicio de los videojuegos es un poco borroso y algunos historiadores sitúan 2 fechas posibles para el inicio. La primera fecha siendo 1952 cuando Alexander

S. Douglas crea el juego Nought and crosses. Esta versión del 3 en raya se ejecutaba sobre la EDSAC [4]. En el año 1958, William Higginbothanm crea Tennis for Two, un simulador del tenis de mesa, siendo ejecutado en un osciloscopio como podemos ver en la figura 1.


*Figura 1 Videojuego Tennis for Two ejecutado en un osciloscopio*

En el año 1962 Steve Russell, estudiante del MIT, crea Spacewar, videojuego donde 2 naves luchan entre ellas. Tras esto sucedieron más videojuegos hasta la eclosión, que ocurrió en el año 1972 cuando Nolan Bushnell creo, en la recién fundada Atari, la versión comercial Tennis for Two. Tras esto, se sucedieron avances tecnológicos que potenciarían la creación de videojuegos, y dando comienzo a la moda de los videojuegos, trayendo consigo salones de recreativos, donde juegos como Space Invaders o Pacman eran los más populares.

A partir del año 80 en adelante, se considera la época dorada de los videojuegos, donde se hacen avances tecnológicos que permiten la creación de gráficos en

3D y la creación de consolas portables, como la GameBoy de Nintendo y, a partir del año 1995, se van haciendo cada vez más avances tecnológicos que permiten a la industria de los videojuegos crear productos con mejores gráficos, historia, jugabilidad y algoritmo de inteligencia artificial más sofisticados.

Hoy en día, gracias a los avances tecnológicos, los videojuegos han sufrido un incremento de calidad, tanto en gráficos como en jugabilidad. Han surgido nuevas tecnologías, como la realidad virtual, que supondrán el futuro en la industria de los videojuegos. También debemos mencionar la cantidad de puestos de trabajos nuevos que han surgido a raíz de esta industria, dando lugar a personas que se ganan la vida no solo desarrollando estos videojuegos, sino a trabajos consistentes en grabar partidas en directo, jugadores profesionales, empresas dedicadas a realizar torneos de un juego especifico y un largo etcétera.

### 2.1.2 Géneros

Los géneros de los videojuegos son las categorías que se utilizan para clasificar los distintos videojuegos que existen. Para esta clasificación, se utilizan elemento claves de la jugabilidad del videojuego como pueden ser los controles del videojuego o el objetivo de este. Hay que destacar que la lista de géneros es distinta dependiendo del medio de referencia que escojamos, aunque se suelen diferenciar solo en pocos géneros. Los más habituales suelen ser:

- **Arcade**: juegos que mantienen los principios de los primeros juegos creados. Suelen tener una jugabilidad muy sencilla y adictiva. DinoGoogle y Pong, los juegos seleccionados para nuestra investigación caen en esta categoría.

- **Plataformas**: genero muy popular consistente en llevar a nuestro personaje desde un punto A hasta un punto B donde se presentan numerosos obstáculos que el jugador debe aprender a esquivar. *Super*

*Meat Boy (https://supermeatbpoy.com)* es un ejemplo muy conocido de este género.

- **Disparos**: género en el cual el jugador toma el control de una persona y/o vehículo para atacar un objetivo mediante el disparo de proyectiles, siendo estas balas y otros tipos de munición. *Counter Strike Global Offensive* es un videojuego de disparos el cual ha ganado mucha popularidad en los últimos años.

- **Lucha**: genero cuyo objetivo es pelear contra un contrincante mediante el uso de técnicas marciales reales o ficticias. El objetivo es atacar hasta vencer al rival. Videojuegos como *Street Figther (https://streetfighter.com)* o *Mortal Kombat (https://mortalkombat.com)* caen en este género.

- **Aventura:** genero donde al jugador se presenta un mundo y una historia y su objetivo es recorrer el mundo en busca de misiones, desafíos u objetos que le ayuden a continuar en la historia principal del juego. La saga de videojuegos de *Zelda* (https://www.zelda.com) es la saga más conocida en este género.

- **Rol:** genero comúnmente llamado RPG en el cual el jugador debe de crear un personaje y sus habilidades para explorar un mundo libre lleno de mazmorras o calabozos laberinticos que explorar, ganar dinero y puntos de experiencia para la mejora continua de su propio personaje. *Dark Souls* (https://es.bandainamcoent.eu/dark-souls/dark-souls) es una saga de videojuegos de rol famosa por tener unos niveles de dificultad muy exigentes y una historia y un mundo muy adictivo.

- **Lógica:** genero de videojuegos que requiere de una destreza mental del jugador para superar objetivos. Este género utiliza las capacidades

de reflejo, memoria o la capacidad de resolución de problemas. *Tetris* sería el baluarte en cuanto a videojuegos de lógica se refiere.

- **Simulación:** genero de videojuegos que permiten recrear una realizad determinada que se asemeja a la vida real en la mayoría de lo posible. *Microsoft Flight Simulator* es un videojuego que utiliza técnicas avanzadas de IA y programación en la nube para la creación de paisajes y mundos.

- **Juegos de mesa:** genero centrado en la creación de videojuegos que están directamente basados en juegos de mesa reales. El 3 en raya y el 4 en raya usados en nuestra investigación caen en esta categoría de videojuegos.

### 2.1.3 Objetos de estudio

Entre los diferentes juegos que podían ser candidatos a ser objeto de este estudio, hemos seleccionado 4 juegos muy conocidos para el público, con mecánicas sencillas y adictivas. A continuación, realizamos una breve descripción de cada uno de ellos:

- **3 en raya**: El 3 en raya (TicTacToe en inglés) es un juego de lápiz y papel en el que 2 jugadores se turnan para marcar un símbolo diferente, siendo normalmente los elegidos O y X, en un tablero cuadrado de 9x9. Gana la primera persona que sea capaz de enlazar 3 símbolos seguidos, ya sea vertical, horizontal o diagonalmente.

- **4 en raya**: El 4 en raya se juega en un cuadricula de 7x6, en la que cada jugador se turna para jugar una ficha, hasta conseguir 4 fichas juntas ya sea horizontal, vertical o diagonalmente. Cada vez que se coloca una nueva ficha, esta debe colocarse lo más bajo posible del tablero. Existen muchas variantes del 4 en raya, donde se suelen cambiar las reglas, y, sobre todo, el tamaño del tablero.

- **Dino:** Videojuego realizado por Google en el año 2014. El objetivo del juego es aguantar el mayor tiempo posible sin colisionar con los diferentes obstáculos que se van presentando a lo largo del juego. Para ello, el usuario puede realizar un salto o agacharse para esquivar dichos obstáculos. Mientras el juego avanza, aumenta la velocidad del juego, provocando una subida en la dificultad.

- **Pong:** Juego arcade basado en el tenis de mesa, realizado por Atari en el año 1972. En este videojuego se controla a una raqueta y el objetivo es devolver la bola hasta que uno de los jugadores falla, momento en el cual se ganara un punto.

## 2.2 Inteligencia artificial aplicada a los videojuegos

Los campos donde se aplican la inteligencia artificial en los videojuegos son los siguientes:

- Investigación de algoritmos. Este apartado se enfoca en la realización de una inteligencia artificial que sea capaz de jugar o resolver un videojuego concreto, siendo los juegos de mesa los más habituales. En este campo, compañías como DeepMind o Stockfish están al alza, consiguiendo, mediante el uso de redes neuronales y algoritmos de aprendizaje, inteligencias artificiales capaces de jugar contra los mejores jugadores del mundo. En la figura dos podemos ver el porcentaje de victorias y derrotas entre AlphaZero y Stockfish.


*Figura 2 Porcentaje de victorias y derrotas de AlphaZero en diferentes juegos de mesa*

- Para la realización de PNJs (Personajes no jugables). Esta es la forma más habitual de encontrarnos inteligencia artificial en los videojuegos. Estas inteligencias artificiales controlan y van aprendiendo del jugador para presentar un mayor desafío mientras mayor sea el tiempo de juego.

- Para la realización de componentes de videojuegos, como la creación de mapas o niveles para dotar de una dificultad adaptativa a los videojuegos.

En el campo de investigación de algoritmos, dos de ellos han ido ganando popularidad a lo largo de los años. El algoritmo minimax para los juegos de mesa y el algoritmo NEAT, para juegos con mecánicas más complicadas. Con estos 2 algoritmos han surgido agentes capaces de aprender a realizar actividades que una persona realiza en su día a día con la ayuda videojuegos, como aprender a jugar al ajedrez o aprender a conducir.

## 2.3 Algoritmo Minimax

El algoritmo minimax es un algoritmo que permite recorrer un árbol de juego y puntuar todos los posibles movimientos, para poder escoger un camino optimo que lleve a la victoria o empate en el juego. Podemos ver en la figura 3 como quedaría un árbol tras aplicar el algoritmo minimax. Los pasos que sigue el algoritmo minimax son los siguientes:

1. Generación del árbol del juego. Un árbol de juego es un árbol con todas las posibles configuraciones que puede tomar el tablero desde el principio de este hasta un tablero terminal.

2. Si se ha llegado a un nodo terminal, se calculará su puntuación utilizando una función heurística concreta.

3. Calcular el valor de los nodos superiores a partir de los nodos inferiores, teniendo en cuenta en la profundidad que se encuentran, es decir, según el nivel del árbol, tendremos que escoger el valor máximo o el valor mínimo entre los nodos inferiores. Con esto, suponemos que nuestro rival siempre realiza el mejor movimiento desde su punto de vista, siendo este movimiento negativo para el agente inteligente.

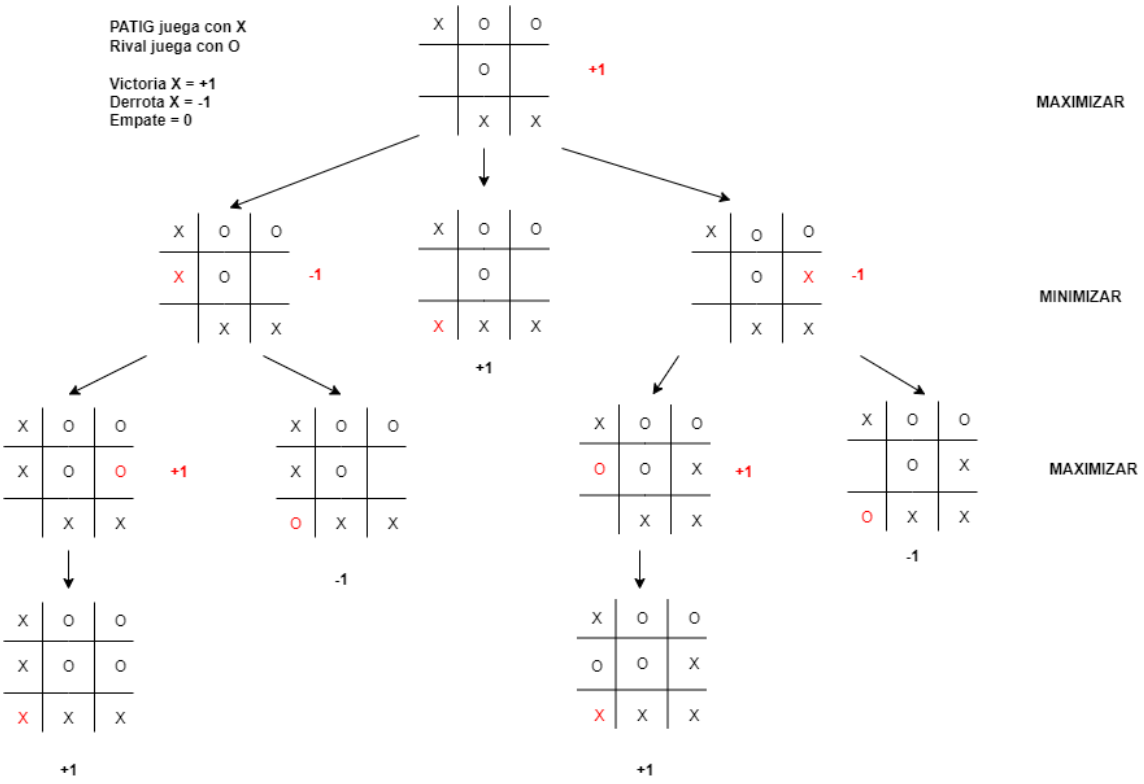


*Figura 3 árbol donde se ejecuta el algoritmo minimax*

4. Escoger la mejor jugada teniendo en cuenta los valores que han ido subiendo desde el nodo terminal.

A continuación, podemos ver cómo funciona el algoritmo minimax en el algoritmo 1.

**ALGORITMO 1: ALGORITMO MINIMAX**

| | |
|---|---|
| | **Entradas:** tablero actual, profundidad máxima y variable para saber si tenemos que maximizar o minimizar |
| | **Salida:** mejor puntuación del tablero |
| **1** | resultado ← checkGanador() |
| **2** | if resultado == 1 |
| **3** | return -1 |
| **4** | else |
| **5** | resultado == 2 |
| **6** | return 1 |
| **7** | else |

```
8           return 0
9   if isMax == true
10          mejorPuntuacion = -infinito
11          realizarMovimiento(tablero)
12          puntuación = minimax(tablero, prof + 1, false)
13          deshacerMovimiento(tablero)
14          mejorPuntuacion = máximo(mejorPuntuacion,puntuacion)
15          return mejorPuntuacion
16  else
17          mejorPuntuacion = +infinito
18          realizarMovimiento(tablero)
19          puntuación = minimax(tablero, prof+1,true)
20          deshacerMovimiento(tablero)
21          mejorPuntuacion = mínimo(mejorPuntuacion,puntuacion)
22          return mejorPuntuacion
```

## 2.4 Algoritmo NEAT

NEAT es un algoritmo neuro evolutivo, es decir, utiliza algoritmos genéticos para hacer evolucionar una población formada por redes neuronales. El algoritmo genético generara una nueva población mejor o igual que la anterior, a partir de métodos de reproducción y mutación.

Para tener una mejor comprensión del algoritmo NEAT, explicaremos antes en que consiste una red neuronal y el algoritmo genético. Tras esto, explicaremos en detalle el algoritmo NEAT.

### 2.4.1 Redes neuronales

Una red neuronal es un modelo de computación consistente en un conjunto de nodos, llamado neuronas que se conectan entre sí para el enlace de datos. A una red neuronal se les pasa unos datos de entrada, y tras una serie de cálculos, tendremos tantas salidas como nodos finales tengan. Esos modelos tratan de simular un sistema nervioso animal.

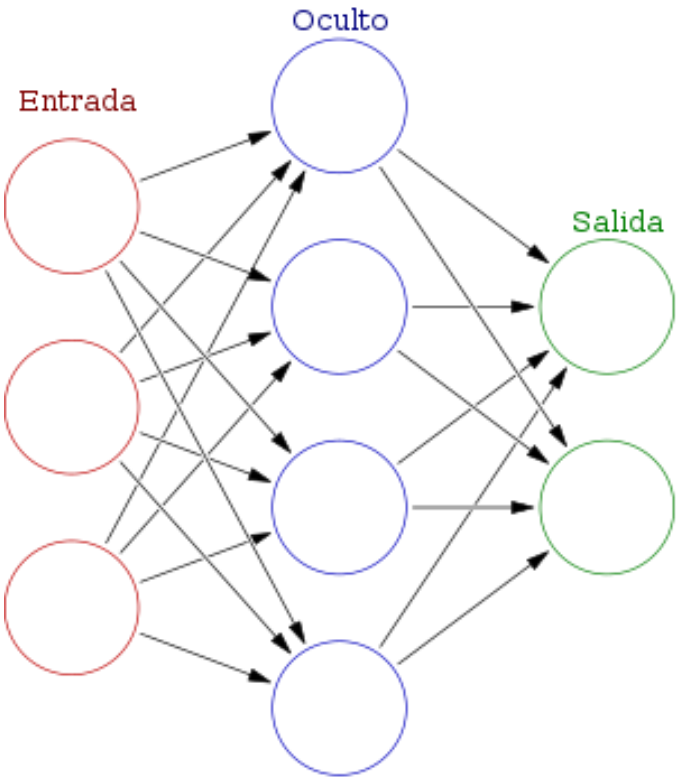


*Figura 4 Ejemplo de una red neuronal con 3 neuronas de entrada y 2 de salida*

Es nuestra investigación nos enfocaremos en las redes neuronales alimentadas hacia delante, o comúnmente conocidas como *feed forward.* En este tipo de redes, la información viaja desde la capa de entrada de la red neural hasta la capa de salida de esta, pasando por las llamadas capas ocultas, que se encargan de realizar procedimientos matemáticos que emitirán a la siguiente capa.

### 2.4.2 Componentes de las redes neuronales

El elemento más pequeño de una red neuronal es la llamada neurona. Una neurona recibe datos a partir de sus enlaces o entradas, procesa los datos recibidos y emite una salida a las neuronas que están conectadas.

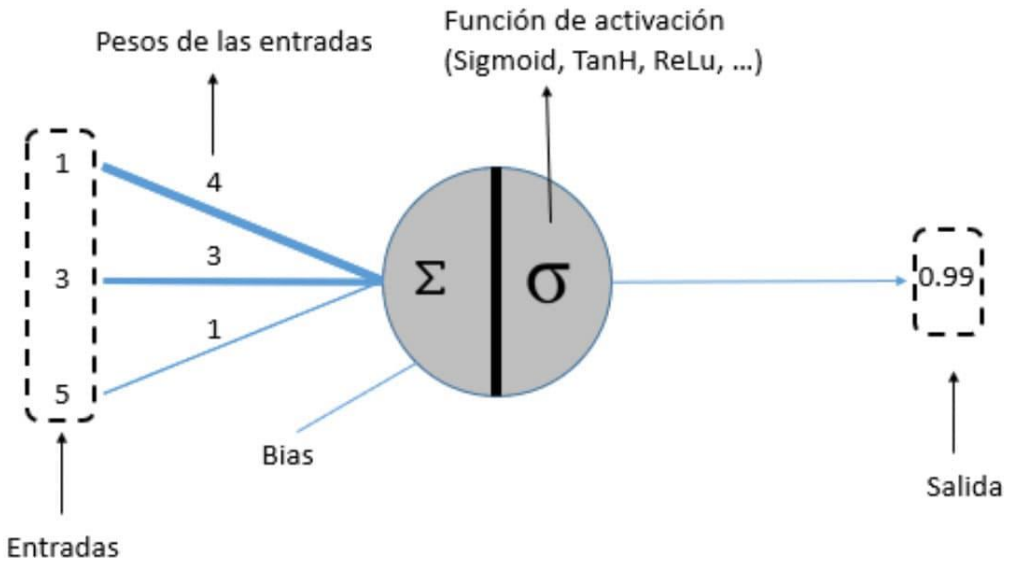


*Figura 5 Ejemplo de una neurona en una red neuronal*

La conexión entre 2 neuronas es la llamada sinapsis. Cada sinapsis tiene asociado un peso a ella, representado si la unión entre 2 neuronas es muy fuerte o débil, es decir, a mayor peso, mayor relación entre 2 neuronas. Para calcular toda la información de entrada, aparte del valor de las sinapsis, se añade un elemento bias, siendo normalmente un número fijo, que nos sirve de sesgo para los cálculos de la neurona. En la figura 5 podemos ver gráficamente como es la estructura de una neurona.

Para el cálculo de los datos de salida, hemos utilizado la llamada función de activación. Estos tipos de funciones suelen ser de 2 tipos: lineales o no lineales. Las funciones de activación más populares son:

- La tangente hiperbólica, cuya expresión es:

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

- La función ReLu, que es la función más utilizada debido a que permite un aprendizaje muy rápido. Su expresión es:

$$relu(x) = \max(0, x)$$

- La función sigmoide o logística, cuya salida suele ser entre 0 y 1. Su expresión es:

$$sig(x) = \frac{1}{1 + e^{-x}}$$

El elemento que da forma a las redes neuronales son las capas. Las capas son un conjunto de neuronas cuyas entradas provienen de una capa anterior y cuyas salidas van a una capa posterior. Existen 3 tipos de capas:

- Capa de entrada: La capa de entrada es la capa que no tiene conexiones previas y sus neuronas no tienen una función de activación. Esta capa es la encargada de captar los datos del entorno y mandar estos datos a la siguiente capa para que se realicen los cálculos.

- Capa oculta: Las capas ocultas son aquellas que realizan los cálculos en la red neuronal. Se les llama ocultas porque no suelen estar conectadas con el entorno y son tratadas como una caja negra, es decir, se les provee de datos y ellas calculan y mandan información al exterior.

- Capa de salida: La capa de salida es la última capa de una red neuronal y es la que transmite la información final al entorno.

### 2.4.3 Algoritmo genético

Un algoritmo genético es un algoritmo de búsqueda de soluciones a un problema concreto, inspirado en la teoría de la evolución natural de Darwin. Para ello, a cada posible solución del problema se le trata como si fuera un individuo particular, formando una población de individuos y se aplica métodos biológicos para encontrar un individuo que sea lo más cercano a la solución que estamos

buscando. En la figura 6 podemos ver una representación gráfica de un algoritmo genético. Como podemos ver en la imagen, existen 5 pasos que sigue un algoritmo genético.

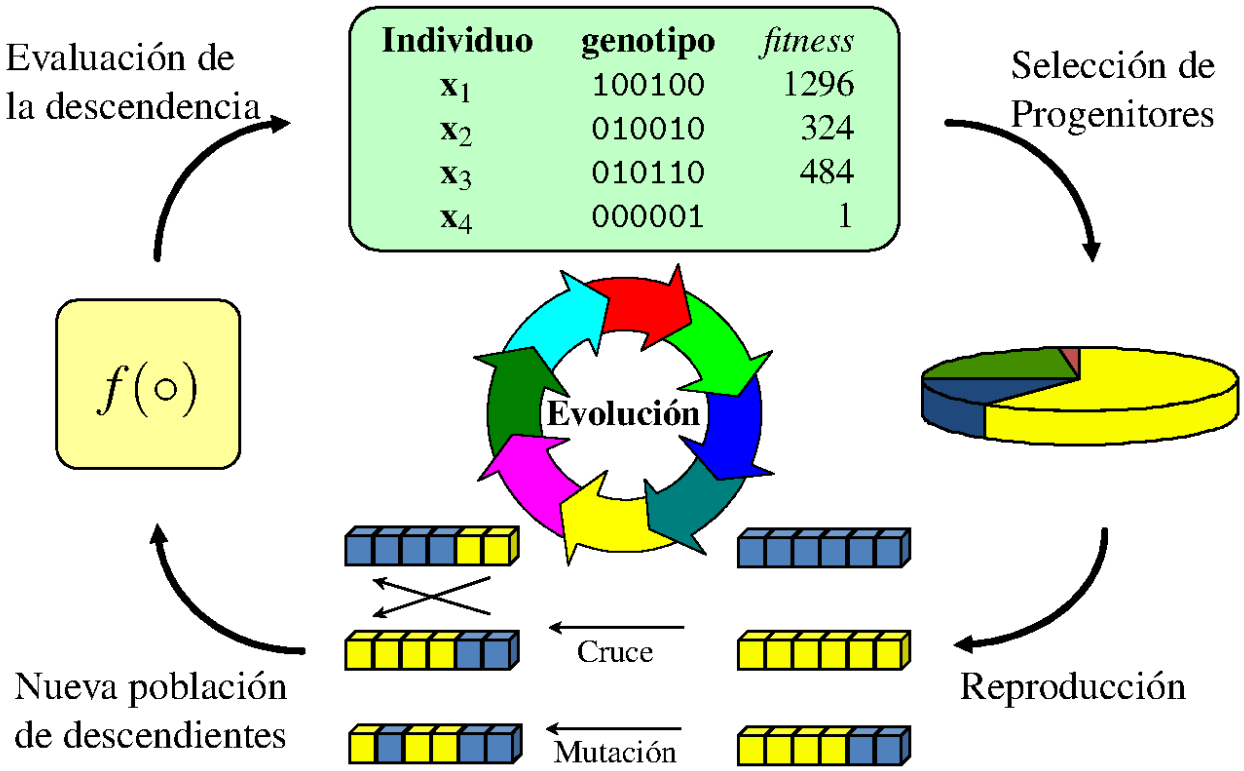


*Figura 6 representación gráfica de un algoritmo genético*

- **Población inicial:** el proceso comienza con un conjunto de individuos llamado población. Esta población está formada por individuos, los cuales son posibles soluciones a nuestro problema. Un individuo tiene un conjunto de variables conocidas como genes. Estos genes se unen para formar la solución del individuo.

- **Función objetivo:** la función objetivo o también llamada fitness, determina que tan bueno es un individuo respecto a los otros individuos en la población dándole una puntuación. Dependiendo de esta puntuación, un individuo tendrá más o menos probabilidades de ser seleccionado para la reproducción.

- **Selección:** En esta fase, se eligen los individuos más aptos para que sus genes pasen a la siguiente generación. Esta selección de dos individuos se realiza teniendo en cuenta la puntuación de cada individuo. Existen diferentes estrategias para realizar esta selección, pero todas ellas tienden a favorecer a individuos con un mayor valor objetivo. Algunas de las estrategias más comunes son:
    - **Selección elitista:** Garantiza la selección de los miembros más aptos (con un fitness mayor) de cada generación. Esta estrategia no es muy usada y la mayoría de los algoritmos genéticos utilizan una versión modificada de esta estrategia.

    - **Selección por ruleta:** La probabilidad de que un individuo sea seleccionado es proporcional a su fitness relativo, es decir, a su fitness dividido entre la suma del fitness de todos los individuos. Esta estrategia presenta problemas si existen individuos con un fitness muy superior al del resto, ya que siempre serán escogidos y los “hijos” de la siguiente generación serán “hijos” de los mismo “padres”, obteniendo poca variación en la población.

    - **Selección por torneo:** Se seleccionan 2 parejas de individuos de la población aleatoriamente. De cada pareja, escogemos el de mayor fitness. Por último, se comparan los 2 seleccionados, seleccionando el de mayor fitness. Esta estrategia genera una población más equilibrada que las estrategias anteriores.

- **Reproducción o Cruce:** Esta fase es la más importante en un algoritmo genético. Consiste en seleccionar a 2 individuos para que intercambien sus genes, producción un nuevo individuo. Las formas más comunes de cruce son: **cruzamiento de un punto**, que consiste en elegir al azar un punto de cruce dentro de los genes. Los descendientes se crean intercambiando los genes de

los padres entre ellos, generando dos individuos nuevos. En la figura 7 podemos ver un ejemplo grafico de este cruce. **Cruzamiento en dos puntos,** donde se intercambian los genes en un intervalo delimitado por 2 puntos y el **cruzamiento uniforme**, donde cada gen de un individuo se escoge entre un padre u otro, con una probabilidad del 50%.

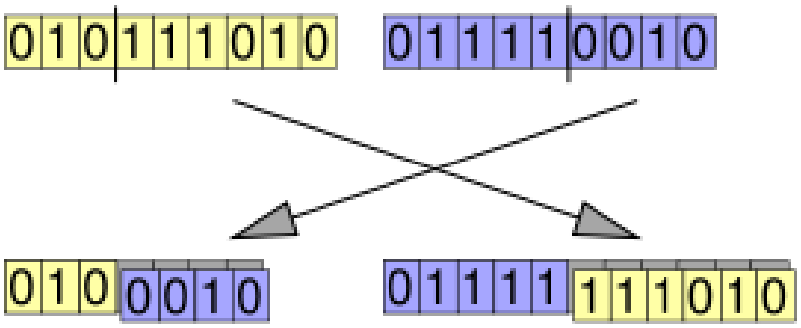


*Figura 7 Cruce de un punto entre 2 individuos en un algoritmo genético*

- **Mutación:** La mutación suele tener una probabilidad muy baja de ejecutarse. Esta mutación consiste en intercambiar el valor del gen de un individuo por otro valor totalmente distinto, causando así alteraciones en punto concretos del individuo.

Con los nuevos individuos formados a partir de la reproducción y la mutación, se forma una nueva población, y se vuelven a realizar los pasos hasta haber realizado un numero de generaciones, tras lo cual el mejor individuo será la solución a nuestro problema.

### 2.4.4 NEAT (NeuroEvolution of Augmenting Topologies)

NEAT empieza con una red neuronal lo más mínima posible, y va añadiendo neuronas y sinapsis por medio de la mutación. Cada vez que se añade algo nuevo, se le llama innovación y el algoritmo NEAT mantiene una vigilancia de todas las innovaciones que se han llevado a cabo. De esta manera, cualquier individuo de la población están identificados por la innovación realizada.

Para realizar la reproducción, NEAT compara los valores de innovación para elegir que gen debe ser escogido para formar parte del nuevo individuo. Los genes que heredan los nuevos individuos son escogidos por esta innovación.

En nuestra investigación, hemos utilizado la librería **neat-python**, es cual es muy utilizada para la ejecución de este individuo. Con esta librería, el proceso del algoritmo genético esta implementado, por lo que el usuario solo tiene que formar la población inicial de redes neuronales e interpretar las salidas de estas. La librería utiliza un archivo de configuración para que el usuario escoja que configuración de las redes neuronales y del algoritmo genético, viniendo una explicación detallada de cada atributo en la página principal de la librería.

En el algoritmo 2, podremos observar la utilización de la librería neat-python, mientras que en el algoritmo 3, observamos un ejemplo de la función fitness.

**ALGORITMO 2: ALGORITMO NEAT**

| | |
|---|---|
| | **Entradas:** config.txt |
| | **Salida:** mejor individuo tras aplicar el algoritmo genetico |
| **1** | import neat |
| **2** | poblacionInicial ← neat.Population(config) |
| **3** | poblacionInicial.addReporter(neat.Reporter(True)) |
| **4** | estadísticas ← neat.StatisticsReporter() |
| **5** | numGeneraciones = 30 |
| **6** | mejorIndividuo = poblacionInicial.run(fitness(), numGeneraciones) |
| **7** | return mejorIndividuo |

**ALGORITMO 3: EJEMPLO ALGORITMO FITNESS**

| | |
|---|---|
| | **Entradas:** poblacionInicial, config.txt |
| | **Salida:** |
| **1** | redes ← [] |
| **2** | poblacion ← [] |
| **3** | for individuo in poblacionInicial |
| **4** | individuo.fitness = 0 |
| **5** | poblacion.append(individuo) |
| **6** | redes.append(neat.Network.create(individuo,config)) |
| **7** | while runJuego |

```
8       ejecutarJuego()
9       for i,individuo in poblacion
10              salida = redes[i].activarRed(posYIndividuo,distanciaObs)
11              If salida > 0.5
12                      individuo.saltar()
13              If individuo.colisionar()
14                      individuo.fitness -= 100
15                      poblacion.pop(individuo)
16                      redes.pop(redes[i])
```

Comentarios: La función fitness cambiara dependiendo del videojuego que estemos ejecutando. Este pseudocódigo pertenece al juego DinoGoogle

# 3. PATIG: Agente inteligente basado en el algoritmo minimax.

## 3.1 Introducción

Para el agente PATIG, hemos desarrollado el funcionamiento del algoritmo minimax para que pueda jugar. Además, hemos tenido que desarrollar 2 optimizaciones para que el algoritmo sea capaz de jugar al 4 en raya. Además, hemos tenido que desarrollar los juegos de mesa, para que PATIG pueda tener un control del árbol de juego y pueda realizar los movimientos.

A continuación, explicaremos cómo funciona Patrig3, el cual se encargará de jugar al 3 en raya y cómo funciona Patrig4, el cual se encargará de jugar al 4 en raya.

## 3.2 PATIG3: PATIG para 3 en raya

### 3.2.1 Implementación del juego

Para la realización del 3 en raya se ha utilizado HTML, JavaScript y CSS para dotar de estilo a la página web. A continuación, expondremos los requisitos que debe de tener el juego:

- El juego debe plantear un tablero de 3x3 con casillas claramente distinguibles.
- El juego debe permitir al jugador realizar un movimiento y que el tiempo de respuesta del agente entre en los cánones de aceptable.
- El juego debe de separar que fichas pertenecen al jugador y cuales a la inteligencia artificial.
- Debe permitir que los jugadores realicen cualquier movimiento que ellos deseen, siempre y cuando estén dentro de los límites del tablero y sigan las reglas del 3 en raya.
- Debe de permitir el reinicio de cualquier partida en cualquier momento.

Dado los requisitos y el diseño original del juego de mesa, hemos creado en HTML una tabla de 3x3 donde el jugado podrá pinchar para indicar donde desea realizar un movimiento. Este movimiento consistirá en insertar un 1 en la tupla *tablero*, donde se indica el estado del juego y sobre el que se realizan todas las operaciones pertinentes. En la figura 8 veremos un diagrama UML del videojuego.

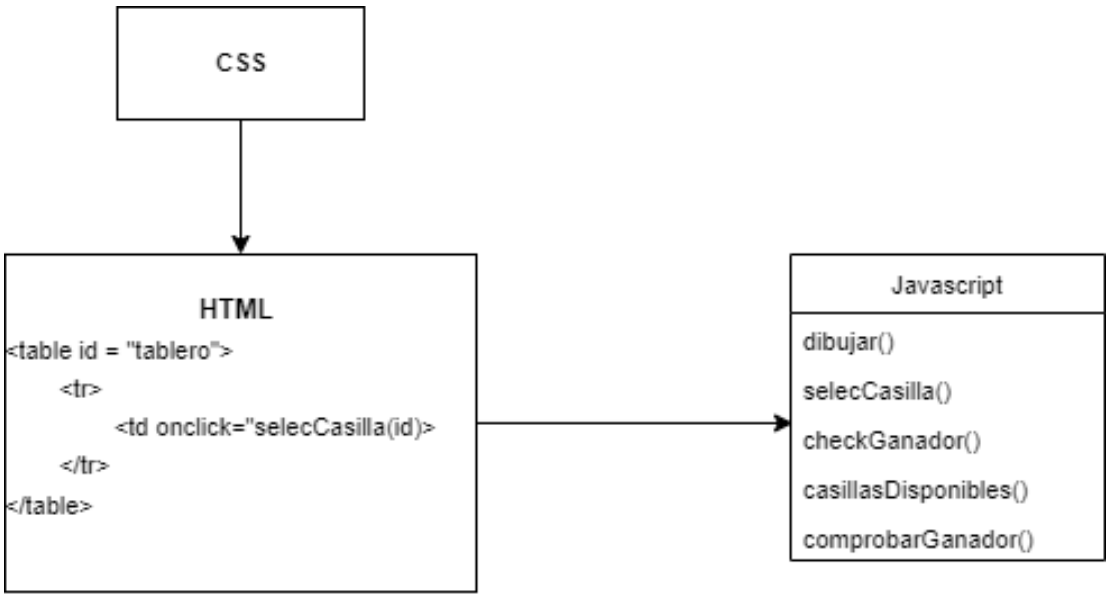


*Figura 8 Diagrama UML con el diseño de los videojuegos 3 y 4 en raya.*

A continuación, se explicará el archivo JavaScript que realiza todas las operaciones con respecto a la lógica del juego:

- ***dibujar():*** método encargado de realizar todas las acciones pertinentes a realizar los movimientos del 3 en raya. Para ello, recorre la tupla tablero, comprobando que numero existe en cada posición. Si encuentra el número 0, pintaría, utilizando la propiedad *backgroundColor* de CSS, de blanco, mientras que si encuentra un 1 pintaría esa posición de rojo y si se encontrara un 2 de azul.

- ***selecCasilla(casilla):*** método que se llama cada vez que se selecciona una posición en la tabla del documento HTML. Cuando se selecciona una posición, se comprueba si el juego esta activo. En caso de estar activo, marcaria la posición del tablero con el numero elegido para el jugador no IA, siendo 1 este caso. Una vez marcado, procede a dibujar el tablero y comprobar si se puede seguir jugando. En caso afirmativo, procedería a realizar la acción Patig3 y volvería a pintar el tablero y a comprobar si existe algún ganador. Remarcar que un jugador no puede marcar una columna ya dibujada, debido a que rompe las reglas y le saltara una alerta.

- ***checkGanador():*** método que comprueba si existe un ganador en el tablero actual. Esto lo realiza comprobando el tablero de manera horizontal, vertical y diagonal, en busca de 3 piezas iguales conectadas. Si encontrara esta conexión, devolvería el número del jugador. En caso de no encontrar, devolvería un 0 si no existen más casillas con las que jugar o un 3, indicando que se puede continuar el juego.

- ***casillasDisponibles():*** método auxiliar que devuelve un conteo de la cantidad de casillas disponibles en el tablero. Estas casillas vienen marcadas con el número 0 en la tupla tablero.

- ***comprobarGanador():*** método encargado de modificar un elemento *span* de la página HTML para marcar quien es el ganador de la partida. Esto se

realiza comprobando quien ha ganado y modificando el elemento *innerHTML* de una *section* concreta. Una vez que ha comprobado que existe un ganador, crea el *span* y desactiva el juego, para que no se pueda seguir jugando al juego una vez finalizado.

### 3.2.2 Desarrollo del agente inteligente

En el caso para PATIG3, para que el agente sea capaz de jugar al juego, hemos tenido que aplicar el algoritmo minimax para que recorra el árbol de juego completo, puntuando todos los tableros que se encuentra para así escoger el mejor movimiento posible. Una vez escogido el mejor movimiento, deberá marcar en el tablero donde colocara la ficha y deberá de esperar a que el contrario realice una jugada para realizar el siguiente movimiento. Esto lo realizamos en los siguientes métodos:

- **mejorMov():** método que utilizamos para realizar la primera jugada, dando comienzo al algoritmo minimax. Tras realizar el primer movimiento, llamara al método minimax(), el cual devolverá la puntuación del movimiento que hemos realizado. Una vez obtenida esa puntuacion, veremos si es mejor que las anteriores puntuacion. Si es mejor, guardaremos esa puntuacion y el movimiento asociado a esta. Una vez tengamos nuestro mejor movimiento, procederemos a marcar en el tablero con un 2 la posición que hemos elegido.

- **minimax():** método en el que se ejecuta el algoritmo minimax. Consiste en una función recursiva que recorre el árbol de juego completo. Una vez que ha llegado a un nodo terminal, procederá a puntuar ese tablero. La manera de puntuar el tablero es la siguiente:

    - Si el tablero es victoria de PATIG3, la puntuación del tablero será de 1.
    - Si el tablero es derrota de PATIG3, la puntuación del tablero será de -1.
    - Si el tablero es un empate, la puntuación del tablero será de 0.

Una vez puntuado el tablero, el método devolverá la puntuación. Una vez devuelta la puntuación, el método deshará los movimientos que ha realizado y se quedara con la mejor puntuación de los tableros, teniendo en cuenta si debe de maximizar la puntuación o minimizarla.

## 3.3 PATIG4: PATIG para 4 en raya

### 3.3.1 Implementación del juego

Para la realización del 4 en raya, hemos utilizado HTML, CSS y Javascript. Para ello, hemos impuesto unas requisitos y reglas básicas que nuestro videojuego debe de seguir para imitar la versión física del juego de mesa. Esas reglas son:

- El juego debe de pintar un tablero de 7x6 con una estética igual o parecida al tablero físico del 4 en raya.
- El juego debe de tener en cuenta los turnos de los jugadores.
- El juego debe de tener 2 fichas, rojas para la persona y amarillas para el agente.
- El juego debe de imitar la mecánica de la gravedad de las fichas y no permitir colocar fichas fuera de los límites del tablero o fichas “levitando”.
- El juego debe de indicar cuando un jugador ha ganado la partida o cuando se ha producido un empate.
- El juego debe permitir realizar cualquier movimiento siempre y cuando ese movimiento sea permitido dentro de las reglas del videojuego.

Con los requisitos y reglas mencionadas, hemos realizado una página en HTML donde tenemos una tabla que será nuestro tablero. Es una tabla 7x6 y en cada celda existirá un método onclick() de Javascript que llamara a nuestro script para que realice toda las mecánicas. Utilizando CSS, se ha alterado el diseño de la tabla para conseguir un tablero lo más parecido a un tablero de 4 en raya real.

Queremos destacar que el diseño del 3 en raya y del 4 en raya son los mismos, variando solo la forma del tablero y el comportamiento de los métodos, por lo

que el diagrama UML del 3 y 4 en raya son los mismos, por eso no se presenta un diagrama UML del 4 en raya.

El archivo Javascript contiene los siguientes métodos:

- **selecCasilla():** método cuya llamada se realiza cada vez que se clica en una celda del tablero. Este método mirara la columna seleccionada y colocara en el primer hueco libre un 1 en la matriz tablero. Tras esto, procederá a dibujar la ficha en el archivo HTML y procederá a llamar al agente para que se realice el mejor movimiento posible. Por último, dibujara la ficha del agente y comprobara si existe algún ganador en el tablero.

- **dibujar():** función que recorre la matriz tablero y va cambiando el estilo de las celdas de la tabla en el archivo HTML, cambiando la propiedad *backgroundColor* del elemento HTML. Si encuentra un 0, pondrá el color a blanco, si encuentra un 1, el color será rojo y si encuentra un 2, el color será amarillo.

- **casillasDisponibles():** método auxiliar que utilizamos para saber la cantidad de casillas disponibles que quedan en el tablero.

- **comprobarGanador():** método que comprueba si existe un ganador en el tablero actual. En el caso de existir un ganador o un empate, modificara un elemento *span* del HTML para anunciar el resultado y desactivara el juego para que el jugador no pueda realizar ninguna acción más salvo reiniciar el juego.

- **checkGanador():** método que recorre la matriz tablero en busca de 4 piezas juntas. Este método comprueba si existen 4 piezas juntas del mismo color de manera vertical, horizontal y diagonal (tanto ascendente como descendente). Si no encuentra 4 piezas juntas y no existe más espacio en el tablero, devolverá un número que indicara que se ha llegado

a un empate. Si existe más espacio, significa que se puede seguir jugando todavía. En caso de encontrar un ganador, devolverá el numero perteneciente a dicho jugador (1 para la persona, 2 para el agente).

### 3.3.2 Desarrollo del agente inteligente

Para el desarrollo del agente Patrig4, debemos de tener en cuenta que en un juego del 4 en raya se calculan que existen 4.531.985.219.002 tableros diferentes en 1 partida. Esta cantidad de tableros es algo que el algoritmo minimax no puede visitar en un tiempo razonable. Para ello, hemos decidido implementar una serie de optimizaciones para conseguir que Patrig4 presente un desafío aceptable a su rival, aunque no sea capaz de resolver el juego del 4 en raya al 100%. Las optimizaciones que hemos seleccionado son:

**Límite de profundidad**

Esta optimización consiste en limitar la profundidad a la que el algoritmo llega en el árbol. Esto hace que el agente calcule una cantidad muy inferior de tableros, pero los suficientes para realizar un movimiento que se considere aceptable. Un problema con esta optimización es que debemos tener una función heurística que nos permita puntuar un tablero que no sea terminal y no podamos encontrar un ganador o un empate en el tablero. En el algoritmo 4 podremos observar el funcionamiento de esta optimización.

La función heurística, llamada score(), saltara cada vez que hayamos llegado a la profundidad máxima permitida y realizara la puntuación del tablero siguiendo los siguientes criterios:

- Si encuentra 3 piezas juntas (ya sean horizontales, verticales o diagonales) que sean del agente, es decir, que tengan el número 2, a la puntuación se le sumaran 5 puntos.

- Si encuentra 2 piezas juntas del agente en cualquier posición, a la puntuación se le sumaran 3 puntos.

- Si encuentra 3 piezas juntas de la persona en cualquier posición, a la puntuación se le restara 4 puntos.

En el algoritmo 5 observamos un pseudocodigo donde vemos como hemos implementado esta función score().

**ALGORITMO 4: ALGORITMO MINIMAX CON LÍMITE DE PROFUNDIDAD**

| | |
|---|---|
| | **Entradas:** tablero actual, profundidad máxima y variable para saber si tenemos que maximizar o minimizar. |
| | **Salida:** mejor puntuación del tablero |
| **1** | resultado ← checkGanador() |
| **2** | if resultado == 1 |
| **3** | return -1 |
| **4** | else |
| **5** | resultado == 2 |
| **6** | return 1 |
| **7** | else |
| **8** | return 0 |
| **9** | if prof > profMaxima |
| **10** | return score() |
| **11** | if isMax == true |
| **12** | mejorPuntuacion = -infinito |
| **13** | realizarMovimiento(tablero) |
| **14** | puntuación = minimax(tablero, prof + 1, false) |
| **15** | deshacerMovimiento(tablero) |
| **16** | mejorPuntuacion = máximo(mejorPuntuacion,puntuacion) |
| **17** | return mejorPuntuacion |
| **18** | else |
| **19** | mejorPuntuacion = +infinito |
| **20** | realizarMovimiento(tablero) |
| **21** | puntuación = minimax(tablero, prof+1,true) |
| **22** | deshacerMovimiento(tablero) |
| **23** | mejorPuntuacion = mínimo(mejorPuntuacion,puntuacion) |
| **24** | return mejorPuntuacion |

**ALGORITMO 5: ALGORITMO HEURÍSTICO PARA PUNTUAR TABLEROS NO TERMINALES**

| | |
|---|---|
| | **Entradas:** |
| | **Salida:** puntuación del tablero |
| **1** | puntuación = 0 |
| **2** | for(recorridoTablero) |
| **3** | if(piezasJuntas = 3 && piezasJuntas = Patig) |
| **4** | puntuación += 5 |
| **5** | else |
| **6** | if(piezasJuntas = 2 && piezasJuntas = Patig) |
| **7** | puntuación += 3 |
| **8** | else |
| **9** | if(piezasJuntas = 3 && piezasJuntas = Humano) |
| **10** | puntuación -=4 |
| | return puntuación |
| | **Comentarios:** El recorrido del tablero lo realizamos mirando todas las posiciones verticales, horizontales y diagonales. |

**Poda alfa-beta**

La poda alfa beta es una técnica que reduce el número de nodos evaluados en un árbol de juegos, en este caso, en el árbol de juego del 4 en raya. Esta poda utiliza 2 parámetros:

- Alfa: es la mejor opción a lo largo del camino para el jugador que queremos maximizar. Tendrá el valor más alto en cada llamada recursiva.

- Beta: es la mejor opción para el jugador que queremos minimizar. Tendrá el valor más bajo en cada llamada recursiva.

El algoritmo minimax debe de ir actualizando estos valores mientras va recorriendo el árbol. Cuando en algún momento nos encontremos con algún valor que sea peor que los valores alfa beta actuales, podara esa parte del árbol, es decir, no visitara esos nodos. En la figura 9, podemos observar cómo queda un árbol tras aplicar una poda alfa-beta.

La poda alfa beta es una optimización muy usada cuando se aplica el algoritmo minimax, debido a que nos permite podar partes del árbol de juego que serían inútiles visitar, ya que no cambiarían en nada el mejor movimiento ni la mejor puntuación. Destacamos que la poda alfa beta es más efectiva cuando los primeros nodos que visitamos sea los que más probabilidades tengan de ser los mejores. Si realizamos esta ordenación visitaremos menos nodos en el árbol de juego.

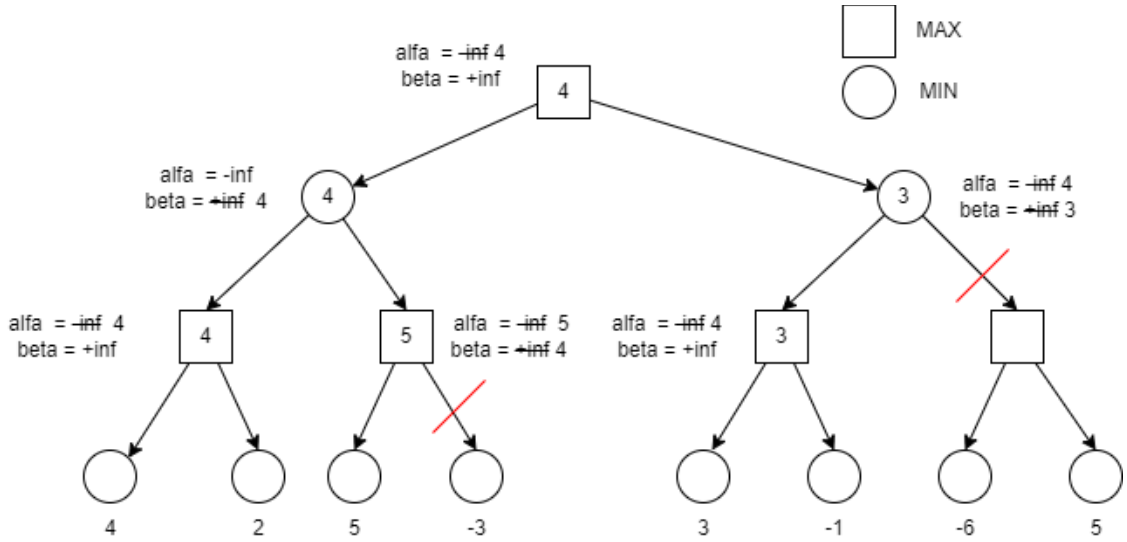


*Figura 9 Esquema de la poda alfa-beta en un árbol de ejemplo*

En nuestra investigación hemos implementado esta técnica de poda dentro del método **minimax(),** donde podemos ver un pseudocodigo en el algoritmo 6.

**ALGORITMO 6: ALGORITMO MINIMAX CON PODA ALFA-BETA**

| | |
|---|---|
| | **Entradas:** tablero actual, profundidad máxima, variable para saber si tenemos que maximizar o minimizar, alfa, beta |
| | **Salida:** mejor puntuación del tablero |
| **1** | resultado ← checkGanador() |
| **2** | if resultado == 1 |
| **3** | return -1 |

| | |
|---|---|
| **4** | else |
| **5** | resultado == 2 |
| **6** | return 1 |
| **7** | else |
| **8** | return 0 |
| **9** | if prof > profMaxima |
| **10** | return score() |
| **11** | if isMax == true |
| **12** | mejorPuntuacion = -infinito |
| **13** | realizarMovimiento(tablero) |
| **14** | puntuación = minimax(tablero, prof + 1, false,alfa,beta) |
| **15** | deshacerMovimiento(tablero) |
| **16** | mejorPuntuacion = máximo(mejorPuntuacion,puntuacion) |
| **17** | alfa = máximo (alfa, mejorPuntuacion) |
| **18** | If (alfa >= beta) |
| **19** | break |
| **20** | return mejorPuntuacion |
| **21** | else |
| **22** | mejorPuntuacion = +infinito |
| **23** | realizarMovimiento(tablero) |
| **24** | puntuación = minimax(tablero, prof+1,true,alfa,beta) |
| **25** | deshacerMovimiento(tablero) |
| **26** | mejorPuntuacion = mínimo(mejorPuntuacion,puntuacion) |
| **27** | beta = mínimo (beta, mejorPuntuacion) |
| **28** | if(alfa >= beta) |
| **29** | break |
| **30** | return mejorPuntuacion |
| | **Comentarios:** Los valores iniciales de alfa y beta son -infinito y + infinito respectivamente |

# 4. GRAPI: Agente inteligente basado en el algoritmo NEAT

## 4.1 Introducción

Para el agente GRAPI, hemos implementado el algoritmo NEAT. Para ello, hemos tenido que desarrollar el proceso de entrenamiento de los individuos, así como la función objetivo. Debido a este motivo, hemos tenido que desarrollar los videojuegos, para que GRAPI pueda jugar a ellos y obtener datos que le

ayudaran en el proceso de entrenamiento, como puede ser la puntuación actual o su posición en la pantalla.

En los siguientes apartados, explicaremos como hemos realizado los videojuegos y como GRAPI ha entrenado y ha jugado a ellos.

## 4.2 DinoGRAPI: GRAPI para DinoGoogle

### 4.2.1 Implementación del juego

Para comenzar el desarrollo del videojuego, primero hemos expuesto los requisitos que debe de cumplir el videojuego:

- Debe de ser un juego que se asemeje al juego de Google.
- El jugador debe de esquivar cactus para conseguir la mayor puntuación posible.
- El juego debe de aumentar la velocidad a medida que el jugador obtiene puntos, dificultado así la tarea de esquivar los obstáculos.
- El juego debe de indicar cuantos puntos está consiguiendo el jugador.
- El jugador debe de tener animaciones para correr y saltar.
- El juego debe permitir entrenar a los agentes o cargar el mejor agente del entrenamiento previo.
- El juego debe de guardar al mejor agente del entrenamiento que se ha realizado anteriormente.

Para realizar el videojuego hemos utilizado el lenguaje de programación Python. En concreto, hemos utilizado la librería **pygame** para la realización del videojuego. Los recursos gráficos han sido extraídos de itch.io y modificados en alguna medida utilizando Gimp.

A continuación, en la figura 10 se muestra un diagrama con las clases que componen el videojuego y como se relacionan entre ellas.

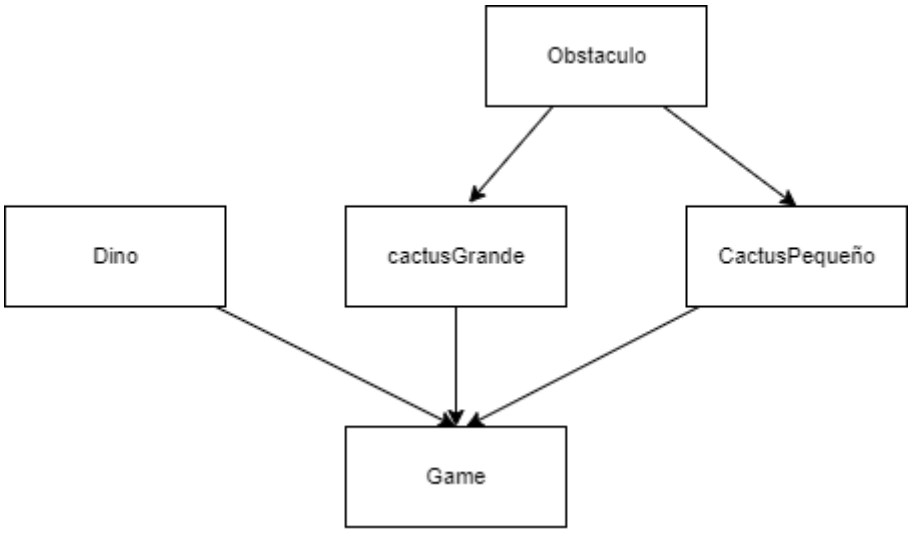


*Figura 10 Componentes del Dino Game y como se relaciones entre ellos*

Pasaremos a explicar cada clase y ver en más detalles sus clases y métodos.

**Dino**

Clase encargada de gestionar al protagonista de nuestro juego. Se encarga de mostrar al dinosaurio en pantalla y de gestionar todas sus animaciones, así como calcular su velocidad y manejar el salto. Sus métodos principales son:

- **update():** método que utilizamos para cambiar el estado de nuestro dinosaurio, es decir, pasar de correr a saltar y viceversa.

- **dinoSaltar():** método encargado de hacer saltar a nuestro personaje. Esto lo realiza disminuyendo la coordenada y de nuestro protagonista mientras reduce la velocidad mientras salta. Una vez que nuestro dinosaurio alcance el cenit de su salto, su velocidad de salto será 0, y comenzara a disminuir. Una vez llegue al suelo, la velocidad de salto será el mismo valor inicial, solo que en este caso en negativo. Justo en ese momento, cambiaremos el estado de nuestro dinosaurio de saltar a correr y reestableceremos la velocidad de salto a la inicial.

- **dinoCorrer():** método encargado de hacer correr a nuestro dinosaurio. En realidad, nuestro dinosaurio no corre, es el fondo el

que se está moviendo, dando así la sensación de que nuestro personaje está corriendo, pero en realidad solo está parado. En este método cambiaremos las imágenes del dinosaurio cada 2 frames del juego, para dar la sensación de que el dinosaurio se está moviendo.

- **dibujar():** método que dibuja a nuestro dinosaurio en la pantalla que le pasamos y en la posición x e y que tiene actualmente y con el sprite elegido.

**Obstáculo**

Clase encargada de gestionar los obstáculos que existen en el juego. Solo existe un tipo de obstáculo, pero cambian los tamaños y la altura de estos, pero su núcleo de funcionamiento es el mismo para todos. Sus métodos son:

- **update():** método que realiza el desplazamiento de los obstáculos en la pantalla, para así dar la sensación de que el dinosaurio se mueve hacia delante. Esto lo realiza disminuyendo la posición en el eje x por cada frame en el juego, restándole la velocidad actual del juego. Con esto conseguimos dar la sensación de que el dinosaurio va más rápido y sea más difícil de esquivar el obstáculo.

  Si el obstáculo ha avanzado y ha desaparecido de la pantalla, se eliminará de la lista de obstáculos que existe en el juego actualmente para dejar paso a los demás obstáculos.

- **dibujar():** método que dibuja al obstáculo en la posición y ventana que se le pasa.

### Cactus

Clase que hereda de obstáculo y se encarga de crear 2 tipos de obstáculo, un cactus pequeño y otro grande, modificando la coordenada y de los mismo, ya que un cactus es más pequeño que otro.

### Game

Clase encargada de ejecutar el juego y de ejecutar el agente que se encargara de entrenar en el juego, o, en caso de haber terminado de entrenar, cargar al mejor agente del entrenamiento. Los métodos principales son:

- **menuInicial():** método que se lanza cada vez que se inicia el juego. Este método carga un título y una serie de botones en una ventana con un tamaño preestablecido. En la figura 11 vemos el diseño gráfico del menú inicial. Dependiendo del botón que presionemos, realizar una acción diferente:
    - Si presionamos la X de la ventana, el videojuego dejara de funcionar.
    - Si presionamos el botón “Train”, el videojuego cargara una serie de agentes y un recorrido para que ellos lo esquiven.
    - Si presionamos el botón “Best” el videojuego cargara al mejor jugador que existe en el sistema, siendo este el mejor individuo del entrenamiento.


*Figura 11 Menú inicial*

- **puntuar():** método encargado de actualizar la puntuación del juego. Cada 100 metros, aumentara en 1 la puntuación global del juego y aumentara en 1 la velocidad actual del juego. Además, modificara el texto puntuación en la esquina superior derecha de la pantalla.

- **fondoDino():** método que se encarga de mover la pista por la que el dinosaurio está corriendo, dando así la sensación de que el que corre es el dinosaurio y la pista no se está moviendo. Esto lo realiza dibujando diferentes secciones de la pista. Una vez que hemos dibujado la pista completa, se volverá a empezar a dibujar desde el principio, para dar la sensación de que la pista es infinita.

- **distancia():** método auxiliar que nos permite conocer la distancia entre 2 puntos, en este caso, el dinosaurio y el obstáculo.

- **dinoJuego():** método que crea una instancia del juego y lo ejecuta con una lista de dinosaurios y una lista de obstáculos. Por cada dinosaurio, se deberá ejecutar el algoritmo Neat, la cual explicaremos en la siguiente

sección de este documento. En términos de diseño de juego, este método crea una instancia del dinosaurio, de la pista y va creando obstáculos a medida que el dinosaurio avanza en la pista, siendo estos obstáculos totalmente aleatorios. Además, maneja la colisión del dinosaurio con los obstáculos, haciendo que el dinosaurio desaparezca de nuestra instancia del videojuego y volviendo al menú principal si este dinosaurio es el único que existe. Por último, crea una serie de textos en la pantalla que nos ayudara a saber la velocidad actual del juego y cuando dinosaurios quedan vivos.

- **bestNeat():** método que crea una instancia del videojuego y carga al mejor individuo que existe en el sistema tras un entrenamiento. Para guardar y cargar al mejor individuo hemos utilizado la librería pickle, que nos permite guardar en archivos binarios un objeto de Python.

- **dinoNeat():** método que se encarga de ejecutar el algoritmo Neat en nuestro sistema.

### 4.2.2 Desarrollo del agente inteligente

Para la realización del agente DinoGrapi, hemos utilizado el algoritmo NEAT. Para ello, hemos utilizado la librería **neat-python**. Con esta librería, el proceso del algoritmo genético esta implementado y no hace falta que tengamos que implementarlo. Con el uso de esta librería, lo único que debemos de realizar es la creación de la red neuronal base, la creación de una población inicial y la función objetivo. La librería cuenta con un archivo de configuración, la cual nos ayudara a escoger las configuraciones del algoritmo genético y de las redes neuronales. En el caso de DinoGrapi, la red neuronal base que hemos realizado ha sido la siguiente:

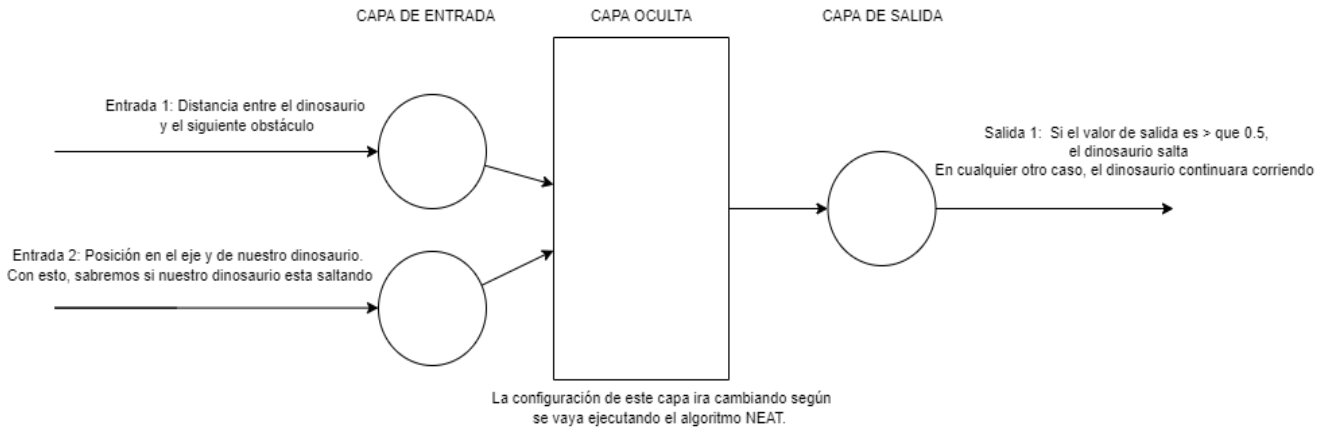


*Figura 12 Red neuronal base usada para el videojuego Dino*

Para el desarrollo de DinoGrapi, hemos implementado los siguientes métodos:

- **dinoNeat()**: método donde se crea la población inicial de redes neuronales, teniendo en cuenta las configuraciones del archivo *config.txt* y se lanza el algoritmo genético. Tras esto, el mejor individuo será guardado en un archivo binario.

- **dinoJuego()**: método que crea una instancia del videojuego y crea tantos dinosaurios como individuos existan en la población. Una vez creados, los individuos comienzan a jugar al juego. Esto se hace activando las redes neuronales de cada individuo y traduciendo los valores de salida a mecánicas del juego. En este método también asignamos el valor objetivo de cada individuo, siendo este la puntuación que ha obtenido en el juego.

- **bestNeat():** método con el cual se carga a DinoGrapi tras un entrenamiento y lo pone a jugar. Para ello, carga desde un archivo binario a DinoGrapi y activa su red neuronal, traduciendo la salida a mecánicas del juego.

## 4.3 PongGRAPI: GRAPI para Pong

### 4.3.1 Implementación del juego

Para comenzar el desarrollo software del videojuego, hemos expuesto los requisitos que debe de cumplir el videojuego:

- El juego debe de parecerse a la primera versión del videojuego Pong.
- Debe permitir al usuario manejar su raqueta.
- Debe de llevar un conteo de la puntuación de cada jugador.
- El juego debe permitir entrenar a los agentes o cargar el mejor agente del entrenamiento previo.
- El juego debe de guardar al mejor agente del entrenamiento que se ha realizado anteriormente.

Antes de pasar a explicar las clases del juego, tenemos que destacar que las clases Ball, Paddle y Game han sido extraídas del Github del usuario *techwithtim* siendo el link del repositorio del github https://github.com/techwithtim/NEAT-Pong-Python.git.

Tras esto, pasaremos a explicar las clases del videojuego y sus métodos.

**Ball**

Método encargado de simular la pelota en el videojuego. Tiene 2 atributos que controlan el radio de esta y la velocidad con la que viaja. Sus métodos son:

- **Init()**: método constructor que nos permite crear un objeto de la clase Ball. Para ello, se le pasan las coordenadas donde se genera la pelota, que será el centro de la pantalla, y se calcula matemáticamente su velocidad y su ángulo para que el juego de comienzo.

- **Get_random_angle:** método auxiliar que nos genera un ángulo aleatorio, el cual la pelota apuntara al principio del juego.

- **draw():** método que dibuja en la ventana la pelota por cada frame con la posición y el radio pasado. Además, la pelota se dibuja de rojo para que destaque en la pantalla.

- **move():** método que mueve la pelota por la ventana. Para ello, suma a sus coordenadas de posición, la velocidad de la pelota.

- **reset():** método que reinicia la pelota. Para ello, cambia las coordenadas de posición de la pelota para que vuelva a la posición inicial, elige un nuevo ángulo aleatorio para empezar el juego u calcula la velocidad de la pelota.

**Paddle**

Clase que simula a nuestras raquetas en el juego. Están raquetas deben de estar en los extremos de nuestra ventana e interactuar con nuestra pelota. Además, deben de moverse hacia arriba o abajo según nosotros queramos. Sus métodos serian:

- **init():** método constructor de la clase. En este caso, creo una raqueta en las coordenadas x e y pasadas.

- **draw():** método que dibuja nuestra raqueta en la pantalla, en blanco, con un ancho y un alto definido y con la posición de la raqueta.

- **move():** método que mueve la raqueta. Para ello, suma o resta a la coordenada y de nuestra raqueta el valor velocidad, dependiendo de si queremos que vaya hacia arriba o abajo respectivamente.

- **reset():** método que reinicia la posición de nuestras raquetas.

**Game Information**

Clase que inicializa la información del juego. Esta información consiste en el número de veces que cada raqueta ha devuelto la pelota y el marcador de una partida. Esto lo realiza con el método **init().**

**Game**

Clase que inicializa una estancia de un juego de Pong. Hace uso de las clases anteriormente mencionadas para crear la instancia del videojuego. Además, dibuja la puntuación actual de la partida y maneja el sistema de colisiones entre la pelota y la raqueta. Sus métodos son:

- **init():** método constructor que crea una instancia de un juego de Pong. Para ello, crea la ventana donde se ejecuta el videojuego, las raquetas y la bola, además de inicializar las puntuaciones.

- **draw_score():** método que los textos de la puntuación y los pinta en pantalla.

- **draw_hits():** método que crea el texto para el conteo de golpes de cada raqueta y lo pinta en pantalla.

- **draw_divider():** método que dibuja la línea divisoria en mitad de la ventana para darle una estética de tenis de mesa.

- **handle_collision():** método que maneja la colisión entre la pelota y cualquiera de las 2 raquetas. Para ello, comprueba si el extremo de la pelota hace contacto, y en caso afirmativo, devuelve la pelota hacia el lado contrario con una velocidad calculada.

- **draw():** método que dibuja en la ventana la línea divisoria, la puntuación, el número de golpes, las 2 raquetas y la pelota.

- **move_paddle:** método que mueve la raqueta izquierda y derecha. Para ello hace uso del método encontrado en Paddle llamado **move()**. También evita que las raquetas no sobrepasen la ventana.

- **loop():** método que ejecuta una única instancia del juego y devuelve la información obtenida.

- **reset():** método que reinicia el juego al completo.

**Pong Game**

Clase realizada por nosotros en la cual ejecutaremos el algoritmo NEAT para entrenar a PongGRAPI. También podremos jugar contra PongGRAPI una vez allá terminado de entrenar. Los métodos encontrados en esta clase son:

- **init():** método constructor que inicializa una instancia del juego de Pong.

- **jugar():** método que utilizamos para jugar contra PongGRAPI una vez terminado de entrenar. Para ello, inicializa un juego de Pong y nos permite manejar la raqueta izquierda, mientras que PongGRAPI manejara la raqueta derecha. El juego no se parará

en ningún caso, siendo este un juego infinito y permitiendo al usuario para el juego saliendo de la ventana en cualquier momento.

- **fitness():** método que utilizamos para calcular el fitness de los individuos que están entrenando. Para ello, suma la cantidad de golpeos que han realizado para cada individuo.

- **entrenarIA():** método que utilizamos para entrenar a los diferentes individuos en el juego. En este método, se inicializa una instancia del juego con 2 individuos diferentes que jugaran entre ellos. En el momento en que alguna de las raquetas falle o consiga realizar 50 golpes consecutivos, calculara el fitness de los individuos y cargara al siguiente.

- **moverRaqueta():** método auxiliar que nos permite mover la raqueta de los individuos. Para ello, se les pasará los valores de la capa de salida de cada individuo y lo traducirá a mecánicas del juego.

Aparte de las clases mencionadas, tenemos una serie de métodos que no pertenecen a ninguna clase, los cuales son:

- **menuInicial():** método que realiza un menú inicial para nuestro juego. Este menú constará un título y 2 botones, los cuales nos permitirá entrenar a PongGRAPI o jugar contra él.

- **neatPong():** método que ejecuta el algoritmo NEAT, con una población inicial. El funcionamiento del algoritmo NEAT los explicaremos en el siguiente apartado.

- **eval_individuos:** método que utilizamos para puntuar cada individuo en la población inicial, dependiendo de lo bien que han jugado al videojuego.

- **cargarMejor():** método que carga a PongGRAPI y nos permite jugar contra él una partida.

### 4.3.2 Desarrollo del agente inteligente

Para la realización de PongGRAPI, hemos utilizado el algoritmo NEAT. Para ello, hemos mantenido la misma configuración que tenía DinoGRAPI, aunque hemos aumentado el número de individuos en la población y hemos cambiado el valor máximo objetivo. Tras haber realizado estos cambios, mostraremos la red neuronal base que utilizara el algoritmo NEAT. Comentar también la utilización de la librería **pickle**, la cual hemos utilizado para almacenar a PongGRAPI en un archivo binario para poder jugar contra el en cualquier momento. La red neuronal que hemos realizado para PongGRAPI ha sido la siguiente:

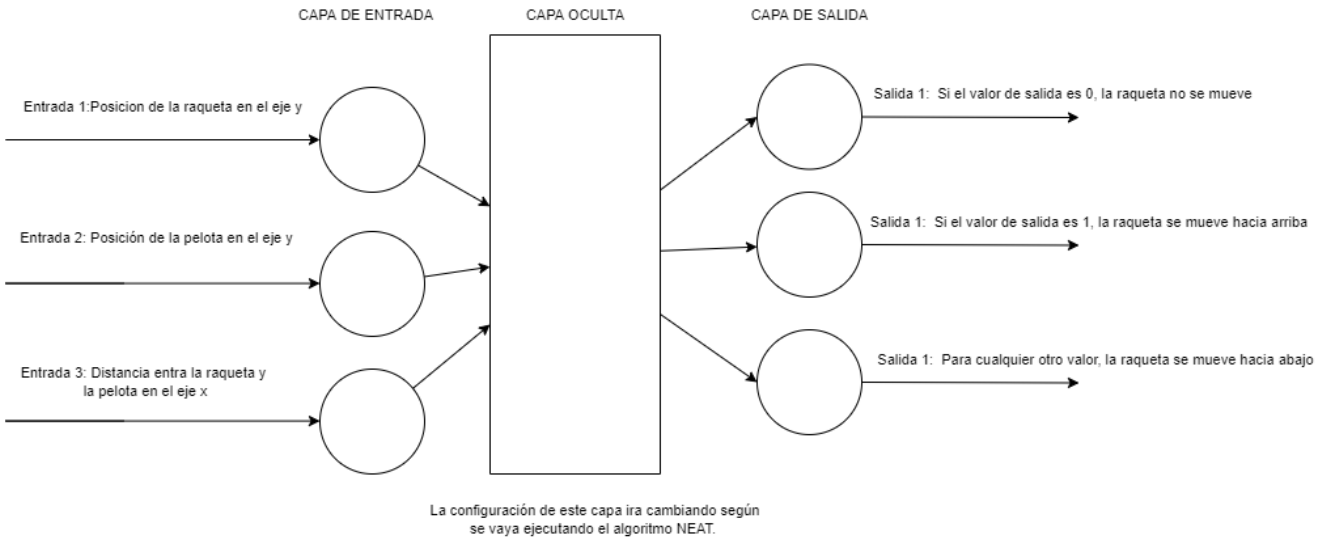


*Figura 13 Red neuronal base usada para el videojuego Pong*

Para el desarrollo de PongGrapi, hemos implementado los siguientes métodos:

- **neatPong()**: método en el cual se crea la población inicial a partir del archivo de configuración *config.txt*. Una vez realizada la población inicial, debemos de ejecutar el algoritmo NEAT. Para ello, la población inicial inicia el método *run*, indicando la función objetivo y el número de generaciones máximas. Una vez terminado el método *run()*, este devolverá al mejor individuo, el cual será PongGrapi. Este mejor individuo será guardado en un archivo binario llamado *mejor.pickle*.

- **eval_individuos()**: método que nos indica el valor objetivo de cada individuo en la población. Para ello, enfrenta a 1 individuo contra todos los individuos de la población y suma el número de golpes que ha realizado en total. Esto lo realiza creando una instancia del videojuego y llamando al método *entrenar_ia()* que explicaremos a continuación.

- **entrenar_ia():** método que utilizamos para crear una instancia del videojuego. En esta instancia, 2 individuos se enfrentarán en un juego de Pong hasta que uno de los 2 falle o alguno de los 2 individuos consiga realizar 50 golpes consecutivos. Para ello, activa las redes neuronales de cada individuo y les pasa los datos de entrada. Una vez pasado los datos de entrada, se recogen los datos de salida de las redes neuronales y se traducen estos valores a mecánicas del videojuego, es decir, se realizan los movimientos pertinentes que ha decidido cada individuo. Tras esto, se realiza la llamada al método *fitness()* para que calcule el valor objetivo de los 2 individuos.

- **cargarMejor():** método que utilizamos para cargar a PongGRAPI desde un archivo binario que guardamos en la raíz del proyecto. Una vez cargado a PongGRAPI, se realiza la llamada al método *jugar()*.

- **jugar():** método que crea una instancia del videojuego para permitir que el usuario se enfrente a PongGRAPI. Para ello, se carga la red de PongGRAPI y se le pasa los datos de entrada y se traducen los datos de salida de la red a movimientos en el juego. También traduce los

movimientos de la raqueta del contrario leyendo las teclas pulsadas en el teclado.

# 5. Experimentación

Los experimentos realizados lo hemos dividido en 4 bloques separados y numerados.

## Experimentos con PATIG3

En el primer bloque de experimentación hemos jugado 40 partidas contra el agente Patig3. En 30 de estas partidas, hemos realizado los movimientos perfectos, y en los 10 restantes hemos realizado el peor movimiento posible. Los resultados han sido:

- En las 30 partidas en la que hemos realizado el movimiento perfecto, hemos obtenido 30 empates.

- En las 10 partidas en las que hemos realizado el peor movimiento posible, Patig3 ha ganado las 10 partidas.

También hemos realizado una medición para saber el tiempo de respuesta de Patig3 con respecto a la cantidad de tableros que está visitando mientras juega. Hemos realizado una gráfica donde en el eje y medimos el tiempo en milisegundos y en el eje x mostramos un intervalo de nodos.

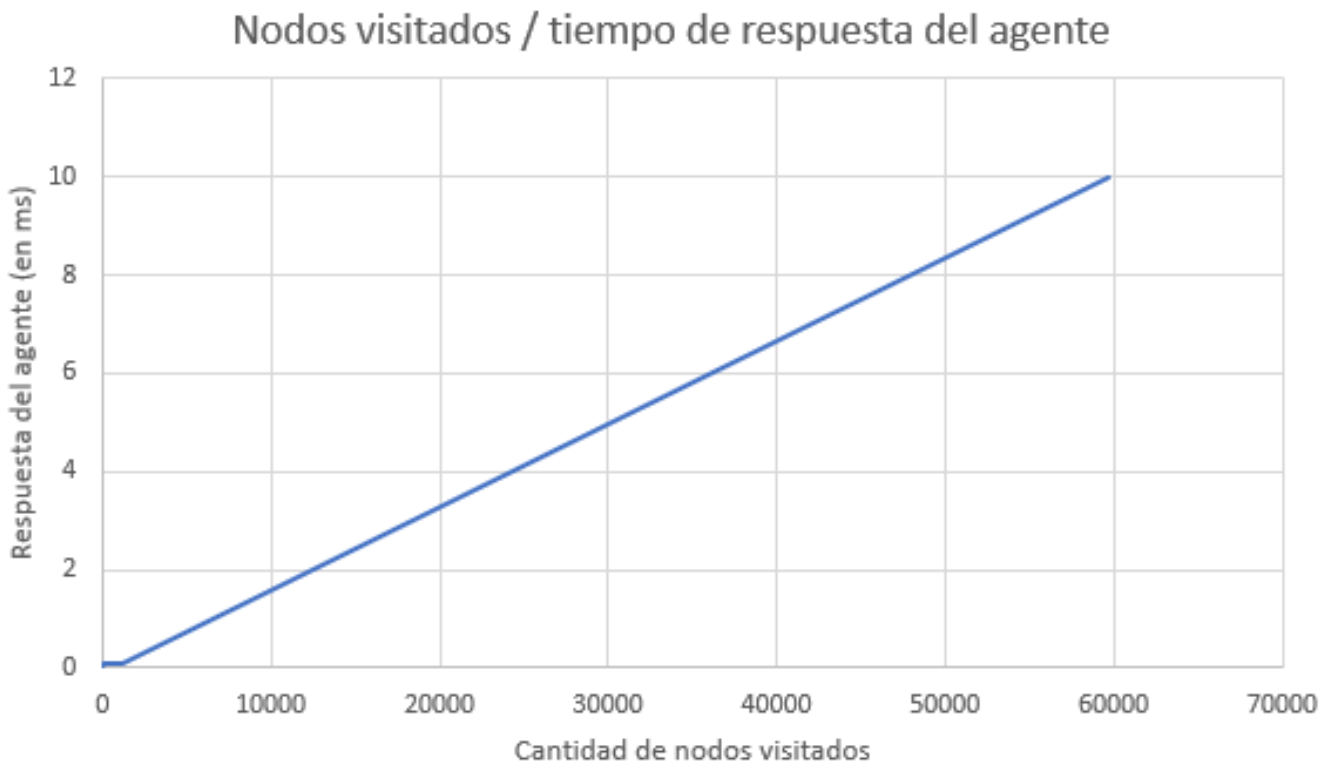


*Figura 14 Grafico en el que medimos el tiempo de decisión del agente Patig3 respecto al número de nodos visitados*

Como observamos en la gráfica de la figura 14, el tiempo de decisión máximo es de 10 milisegundo, mientras que el tiempo de decisión mínimo es de apenas 1 milisegundo.

## Experimentos con PATIG4

En el segundo bloque de experimentación hemos jugado 10 partidas contra el agente Patrig4.En estas partidas, hemos jugado el mejor movimiento posible en cada situación. Para estas partidas, hemos seleccionado como profundidad máxima 4, debido a que es la profundidad que nos ofrece un tiempo de respuesta aceptable, mientras mantiene un nivel de dificultad medio-alto. Es decir, Patrig4 ha sido capaz de calcular 4 movimientos a futuro. Los resultados de estas partidas han sido:

- De las 10 partidas, 7 de ellas han sido victoria para Patrig4, mientras que los 3 restantes han sido victorias para la persona.

A continuación, en la figura 15 se muestra una gráfica en el que podremos ver como aumenta el tiempo de decisión al aumentar la profundidad máxima:

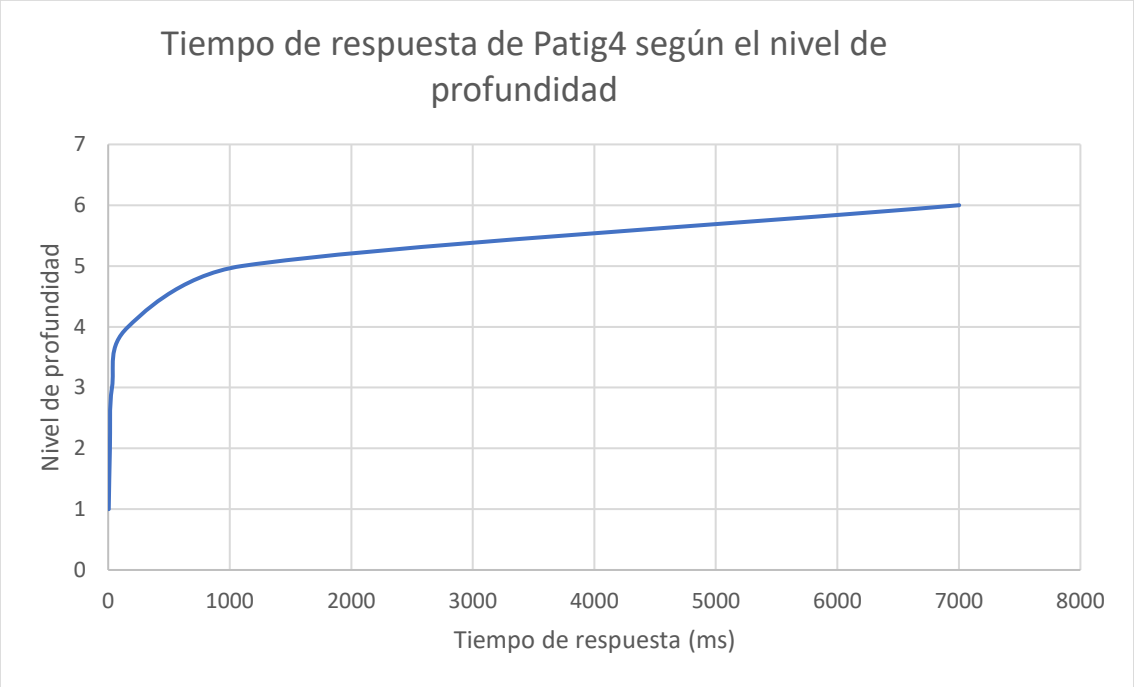


*Figura 15 Grafico en el que medimos el tiempo de respuesta de Patig4 respecto a la profundidad en el árbol.*

Tenemos que comentar que a partir de un nivel de profundidad 6, Patig4 tarda demasiado tiempo en responder, dándonos un error en el navegador.

Para finalizar este bloque de pruebas, hemos enfrentado a Patrig4 contra un solucionador completo del 4 en raya (https://connect4.gamesolver.org/es/). Tras haber jugado un total de 10 partidas, en las cuales Patrig4 tenía un nivel de profundidad de 4, ha obtenido un total de 0 victorias y 10 derrotas.

## Experimentos con DinoGRAPI

En el tercer bloque de experimentación, hemos configurado a DinoGRAPI para tener una cantidad de 20 individuos por población. La función de activación elegida ha sido la función ReLu y el valor maximo objetivo ha sido de 5000.

Tras haber realizado esta configuración, hemos variado el número de generaciones de entrenamiento de DinoGRAPI. Los resultados han sido:

- Con un total de 10 generaciones, ha tardado 8 minutos en terminar el entrenamiento. Hemos obtenido también que el mejor fitness ha sido de 3734, con una velocidad cercana al valor 50.

- Con un total de 30 generaciones, ha tardado 30 minutos en terminar el entrenamiento. Hemos obtenido que el mejor fitness ha sido de x y ha alcanzado velocidades cercanas a 60

- Con un total de 50 generaciones, ha tardado 53 minutos en terminar el entrenamiento. Hemos obtenido que el mejor fitness ha sido de 4562 y ha alcanzado velocidades cercanas a 65.

## Experimentos con PongGRAPI

Para finalizar, en el bloque 4 hemos realizado 2 entrenamientos con PongGRAPI. En este caso, hemos mantenido una configuración de 50 individuos por población, con una función objetivo máxima de 10000 golpes y la función de activación ReLu, pero hemos variado el número de generaciones. Los resultados han sido:

- PongGRAPI con 10 generaciones: el entrenamiento ha tardado un total de 14 minutos en completarse. Una vez terminado, hemos jugado un total de 5 partidas contra él, para ver quien fallaba primero. En total, PongGrapi ha fallado 3 veces en golpear la bola.

- PongGRAPI con 30 generaciones: el entrenamiento ha tardado un total de 26 minutos en completarse. Una vez terminado, hemos juego un total de 5 partidas contra él, para ver la calidad tras el entreno. En las 5 partidas, ha devuelto todas y cada una de las pelotas lanzadas, ya sean desde el inicio o controladas por el jugador.

# 6. Conclusiones

Tras haber desarrollado a nuestros agentes y haber realizado diferentes experimentos para ver su rendimiento, podemos concluir que:

- Patig, al utilizar el algoritmo minimax, es un agente muy potente que será capaz de jugar a casi cualquier juego de mesa. El único problema real será el tiempo, debido a que el algoritmo minimax es un algoritmo de fuerza bruta, tardara bastante en calcular todos los movimientos posibles de los juegos de mesa.

- Con respecto a Grapi, este agente solo necesita tiempo para realizar un entrenamiento adecuado, aunque eso cueste mucho tiempo en finalizar. Cabe destacar que hemos mantenido la misma configuración para el agente, por lo que es conveniente que debamos experimentar con distintas configuraciones para obtener un agente con mayor calidad.

Para finalizar, nos gustaría destacar algunas de las competencias que mas hemos desarrollado en la realización de nuestro trabajo final de grado y en la que hemos mejorado enormemente. En concreto, nos gustaría destacar las siguientes competencias:

- Competencia de reunir e interpretar datos relevantes para emitir un juicio o reflexiones en temas científicos.

- Poder transmitir información, ideas y problemas a un público tanto especializado como no especializado.

- La capacidad para resolver problemas con iniciativa y creatividad.

# 7. Trabajos futuros

Para finalizar, algunas de las líneas futuras que queremos enfocarnos a partir de ahora son las siguientes:

- **Mejora continua:** En este caso, nos gustaría mejorar a GRAPI y la forma de puntuar cada individuo en el algoritmo NEAT. Este tipo de mejoras podrían ser la de probar diferentes configuraciones del algoritmo NEAT y realizar un torneo para escoger al mejor agente, normalizar la forma de puntuación de los individuos y optimizar algunas partes de nuestro código para conseguir disminuir el tiempo de entrenamiento del individuo. En el caso de Patig, nos gustaría refinar la puntuación de tableros no terminales. Para ello, hemos decidido realizar una especia de competición entre distintos agentes Patig, con distintas configuraciones, para al final, escoger al mejor Patig entre ellos.

- **Probar nuevos juegos:** En este caso, nos gustaría que nuestros 2 agentes sean capaces de aprender a jugar nuevos juegos. Estos juegos queremos que sean más complicados, para observar cómo se comportan y la manejar de mejorar a los agentes. Para Patig, el siguiente juego objetivo sería el Otelo, mientras que para GRAPI, el siguiente juego objetivo sería el Galaga.

- **Proyectos no orientados a videojuegos:** Para finalizar, nos gustaría comprobar si nuestros agentes serian óptimos para trabajo no orientados a videojuegos, y pretendemos probar a nuestros agentes para proyectos en diversos sectores de economía, logística o toma de decisiones.

# 8. Referencias bibliográficas


[1]. Google Deep Mind, "AlphaZero". https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go . Accedido: 2022

[2]. Javier Sotillo Mallo, Cristian Martínez Ruiz. "Inteligencia Artificial en los Videojuegos.". http://www.it.uc3m.es/jvillena/irc/practicas/13-14/03.pdf. Accedido: 2022

[3]. ABC, "La inteligencia artificial AlphaZero ya juega y aprende como un superhombre". https://bit.ly/2BWrSiY. Accedido: 2022

[4]. J.J. Velasco, "Historia de la tecnología: OXO, un videojuego para uno de los primeros computadores de la historia". https://bit.ly/3pMP6Qt. Accedido: 2022

[5]. Wikipedia, "Tennis for Two". https://en.wikipedia.org/wiki/Tennis_for_Two. Accedido: 2022

[6]. Stefan Edelkamp y Peter Kissmann, "Symbolic Classification of General Two-Player Games". https://bit.ly/3wxrx23 . Accedido: 2022.

[7]. Fernando Sancho Caparrini, "Minimax: Juego con adversario". http://www.cs.us.es/~fsancho/?e=107 . Accedido: 2022

[8]. Marissa Eppes, "Game Theory – The Minimax Algorithm Explained". https://bit.ly/3enmT0i . Accedido: 2022

[9]. Stefan Edelkamp, Peter Kissman. "Symbolic Classification of General Two-Player Games".

https://link.springer.com/chapter/10.1007/978-3-540-85845-4_23. Accedido 2022

[10]. Dario Blasco, “Estudio del algoritmo NEAT aplicado al videojuego Pac-Man”. https://bit.ly/3wR7N9J . Accedido: 2022.

[11]. Kenneth O. Stanley, Bobby D. Bryant, Risto Miikkulainen. “Real-Time Neuroevolution in the NERO Video Game”. https://bit.ly/3ehMM1r . Accedido: 2022