Proyecto de desarrollo de un motor de juego para hex utilizando imitation learning

CONVOCATORIA: Extraordinaria

ALUMNO: Miguel Tello Gómez

TUTOR: Georgy Nuzhdin Gelfand

GRADO: Ingenieria de Software

TIPO: Convencional

Índice

1 Resumen

1.1 Resumen

Este trabajo de fin de grado consiste en el estudio y realización de una inteligencia artificial dedicada a participar como un jugador en el juego de mesa de estrategia Hex. El objetivo no es realizar un algoritmo que sea capaz de jugar de jugar de forma competente, sino que sea capaz de aprender de sus rivales a través de aprendizaje por imitación. Para ello se ha realizado una investigación sobre algunos métodos que podrían estar relacionados.

En este proyecto se estudia la posibilidad de desarrollar mediante técnicas de Deep Learning un modelo de aprendizaje de imitación que pueda interpretar un tablero del juego hex y dar como resultado un movimiento apropiado.

1.2 Abstract

This project consists in the study and development of an artificial ingeligence dedicated to participate as a player in the strategy board game Hex. The goal is not to make and algorithm capable to winning to others agents but to learn to play like the player learning from him through imitation learning. For this there has been an investigation about some of the methos that might be related.

In this project we study the posibility of developing by means of Deep Learning an imitation learning model capable of interpret a board of hex and return as a result a proper move.

2 Introducción

2.1 Motivación

A la hora de desarrollar modelos de inteligencia artificial en los juegos de mesa la mayoría piensa en los juegos más populares como es el caso del ajedrez y la supercomputadora Deep Blue de IBM o en los más difíciles como el go y el AlphaGo. Sin embargo, considero que no podemos dejar atrás otros juegos menos populares pues podrían aportar una inmensa cantidad de información.

Por consiguiente, este trabajo se enfoca en el juego de mesa Hex. Hex se distingue por ser un juego estratégico con reglas sencillas, jugado en un tablero no convencional, lo que ofrece una complejidad idónea para el diseño y empleo de agentes de inteligencia artificial. En este proyecto se abordará la discusión de diversos agentes conocidos, siendo el método más difundido el Monte Carlo tree search.

Otra motivación que ha impulsado la realización de este trabajo es la oportunidad de desarrollar una idea que ha sido concebida desde hace varios años: el Aprendizaje por Imitación. Para llevar a cabo este objetivo, se requiere la implementación de una inteligencia artificial capaz de aprender de manera iterativa con cada movimiento ejecutado por un agente humano. En este contexto, se ha optado por utilizar una red neuronal alimentada con los movimientos efectuados en el juego como parte del proceso de aprendizaje. En resumen, este juego ha sido elegido por la complejidad de su extraño tablero, así como la oportunidad de utilizar aprendizaje por imitación.

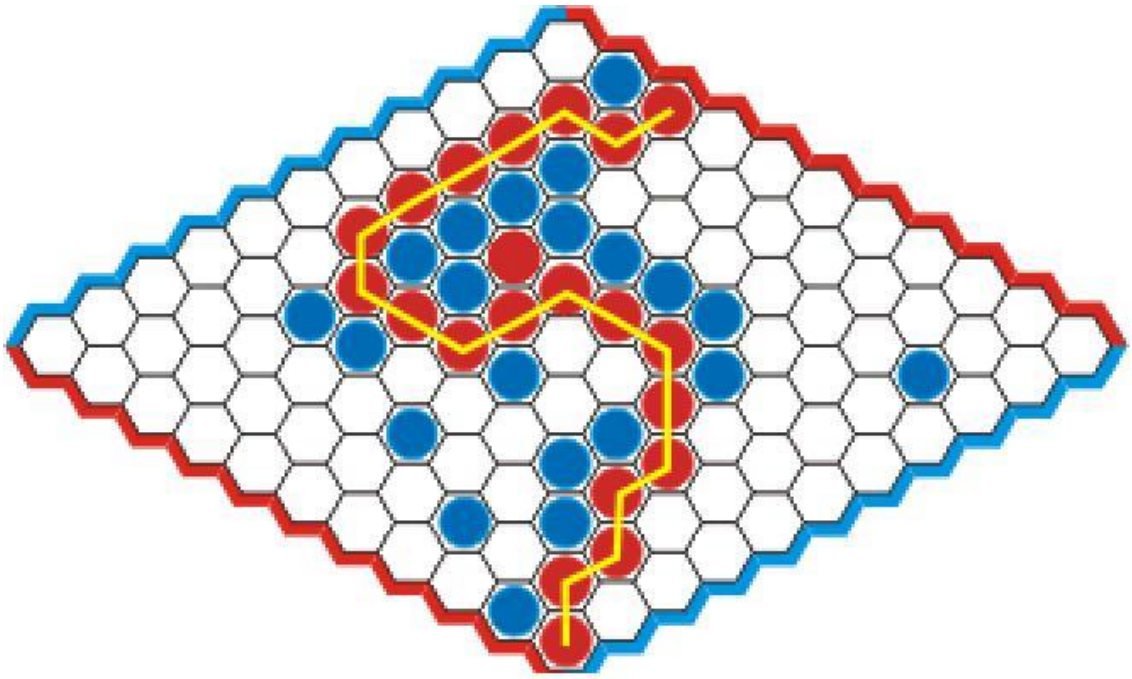
2.2 Planteamiento del problema

Para entender este trabajo hay que empezar con unos conocimientos básicos. Para empezar, tenemos que conocer en que consiste el juego Hex.

Hex es un juego de dos jugadores creado en 1942 por el matemático danés Piet Hein y redescubierto por John Nash más tarde en esa misma década. El objetivo de este juego es crear un camino entre lados opuestos de un tablero en forma de rombo con casillas hexagonales. Esta demostrado matemáticamente que no es posible empatar, solo es

posible que uno de los jugadores gane, lo que convierte a este en un juego de suma cero. Hex es un juego de tipo maker-breaker los cuales consisten en que los dos jugadores tienen roles diferentes concretamente un jugador tratará de conectar sus lados mientras que el otro deberá crear una estrategia para impedirlo. En este juego el primer jugador tiene mucha ventaja por lo que, en 1949, John Nash invento una regla que permite al segundo jugador elegir si quitarle la primera ficha al primero pero no vamos a usar esta regla en este proyecto. El formato original del tablero de juego es de 11x11 con algunos tableros más grandes también siendo populares. Sin embargo, en este proyecto el tablero utilizado para entrenar la red de neuronas es de 8x8. Este juego ha sido conocido a lo largo de la historia con muchos nombres. Por ejemplo, es llamado Nash en homenaje al hombre que lo popularizo. Este juego fue comercializado con el nombre Con-tac-tix hasta que en 1952 la empresa juguetera Parker Brothers empezó a venderlo con el nombre de hex. La complejidad computacional del árbol del juego para un tablero de 11x11 es de 10^98 frente a la del ajedrez que es de 10^123.

Tablero 11x11 típico de una partida de Hex:



Una vez establecido el contexto del juego, es pertinente plantear la siguiente pregunta: ¿Se podría desarrollar una inteligencia artificial capaz de aprender de las partidas que tiene con el jugador?

Existen actualmente técnicas de inteligencia artificial que han demostrado competencia equiparable a la de los humanos en el juego Hex. Este proyecto se centra en la exploración de aquellas técnicas que puedan adaptarse específicamente al contexto del juego Hex. El desafío planteado posee todas las características inherentes a un problema de aprendizaje por refuerzo, método ampliamente utilizado en el desarrollo de inteligencias artificiales capaces de jugar juegos de mesa clásicos, como el ajedrez y el go.

No obstante, es importante señalar que los modelos existentes ya poseen un nivel de habilidad elevado al momento de su implementación, mientras que la propuesta en este proyecto consiste en que el modelo de inteligencia artificial aprenda de manera progresiva durante las partidas con el jugador humano. Para lograr que el modelo adquiera conocimiento exclusivamente a partir de la interacción con el jugador, se requieren técnicas más específicas que el simple aprendizaje por refuerzo. Este requerimiento ha conducido a la investigación sobre el *Imitation Learning*.

El aprendizaje por imitación que se va a utilizar en este proyecto es una versión del aprendizaje supervisado. En el aprendizaje por refuerzo la inteligencia artificial aprende a través de la recompensa y el castigo lo que hace que la inteligencia artificial busque estrategias correctas mientras que el aprendizaje por imitación trata de buscar que hace un humano en cada situación sin pensar en si es un buen movimiento o no.

En definitiva, la respuesta a la pregunta que se plantea en este proyecto consiste en realizar un modelo que aprenda a jugar Hex usando alguna técnica de *Imitation Learning*.

2.3 Objetivos

Los objetivos obligatorios son la base del desarrollo de este proyecto:

1. La identificación de la técnica óptima de Aprendizaje por Imitación constituye un aspecto fundamental de este estudio. Se llevará a cabo una revisión exhaustiva de las distintas técnicas disponibles en el ámbito del Aprendizaje por Imitación, con el propósito de determinar la alternativa más adecuada para su aplicación en este proyecto.
2. Desarrollo de un modelo que juegue a Hex. Se establece como requisito fundamental que dicho modelo sea competente para ejecutar movimientos conforme a las reglas establecidas para el juego de Hex.
3. Que el modelo sea capaz de aprender de la experiencia propia o de otro jugador. Combinando lo aprendido sobre Imitation Learning darle al modelo la habilidad de aprender de cualquier movimiento que se realice durante el transcurso de la partida y que sea capaz de usarlo en futuras partidas e incluso en otro entorno diferente.

Los objetivos necesarios son partes del desarrollo que, aunque no son el objetivo principal, el proyecto no se podría completar sin realizarlos. Si es posible estos objetivos se delegarán a herramientas externas:

1. Desarrollo de un motor para el juego Hex. Se necesita tener un programa que sea capaz de simular una partida y permita a jugadores de distintos orígenes participar.
2. Desarrollo de una interfaz gráfica para el juego Hex. Sera necesario tener la posibilidad de ver como se desarrollan las distintas partidas para poder realizar un análisis de las jugadas que realizan los distintos jugadores.

Los objetivos opcionales están pensados para desarrollarse si fuera posible pero no añadirían nada al propio estudio de este proyecto:

1. Introducción de la regla de robo de ficha en el primer turno por parte del segundo jugador. Es una regla que sirve para contrarrestar la ventaja táctica que posee el jugador que empieza en primer lugar ya que en ocasiones puede resultar en una partida completamente desnivelada.
2. Introducción de una opción para poner el modelo en el turno que se quisiera en vez de un turno predeterminado. El modelo debería poder aprender independientemente del turno que le ha tocado. Sin embargo, sería posible realizar este ejercicio poniendo al modelo a jugar siempre con el mismo rol.
3. Introducción la posibilidad de cambiar el jugador humano por un modelo ya fuera uno ya existente o de fabricación propia. Es interesante que se pueda enfrentar este modelo contra otros modelos que ya han sido desarrollados por otros y poder comparar los resultados.

3 Estado de la cuestión

Antes de desarrollar este trabajo con detalle es necesario hablar de las tecnologías que precedieron a este trabajo e inspiran muchas de las técnicas utilizadas hoy en día.

3.1 Marco teorico

Muchas de las cosas que se van a mencionar en esta sección son conceptos básicos que se han dado durante el desarrollo de la carrera así que si ya los conocéis podéis saltároslo sin ningún tipo de problema para entender el resto del proyecto.

3.1.1 La teoria de juegos

La teoría de juegos es un área de las matemáticas aplicadas que utiliza modelos para estudiar interacciones en estructuras conocidas como juegos. La teoría de juegos sugiere

que debe haber una forma racional de jugar cualquier juego o problema, especialmente frente a muchas situaciones engañosas. Así, la anticipación mutua de las intenciones de los oponentes que ocurre en algunos juegos crea una cadena teórica infinita de razonamiento que puede traducirse al ámbito de la resolución real y compleja de conflictos. Cabe destacar que uno de los eruditos más importantes de la teoría de juegos es el propio John Nash, el creador de hex que según sus propias palabras era un juego matemáticamente perfecto.

La disciplina de la teoría de juegos fue iniciada a principios del siglo 20 por los matemáticos Ernst Zermelo (1913) y John von Neumann (1928). El descubtimiento vino por la publicación del libro Theory of Games and Economic Behavior en 1944 por John von Neumann y Oskar Morgenstern. Esto fue seguido por la importante labor de John Nash (1950-51) y Lloyd Shapley (1953). La teoría de juegos tuvo una gran influencia en el desarrollo de muchas ramas de las ciencias económicas (organización industrial, negocios internacionales, economías del trabajo, macroeconomía, etc.). Con el tiempo el impacto de la teoría de juegos se extendió tanto a otras ramas de las ciencias sociales (ciencias políticas, relaciones internacionales, filosofía, sociología, antropología, etc.) como a campos fuera de las ciencias sociales como la biología, la ciencia de la computación, lógica, etc.

Entre los juegos que se manejan en la teoría de juegos hay que mencionar los juegos de suma cero entre los cuales se encuentra el propio hex. Los juegos de suma cero se caracterizan porque el beneficio total de todos los participantes del juego es siempre 0. En juegos de dos jugadores esto quiere decir que siempre gana uno de los dos jugadores y obtiene un punto mientras que el otro jugador pierde un punto a no ser que haya un empate que entonces no ganan nada y suman cero. El caso del hex es especial ya que no existe la posibilidad de hacer un empate ni siquiera forzándolo.

3.1.4 Marcov decision processes

Marcov decision process o MDP es un concepto matemático que provee a un usuario la posibilidad de hacer tomas de decisión en escenarios definidos matemáticamente. Una de las primeras personas en utilizar el término MDP fue Richard E. Bellman en su libro de 1957 "A Markovian Decision Process", y el concepto fue utilizado por primera vez

por Ronald A. Howard en el año 1960 con "Dynamic Programming and Markov Processes", Alvin W. Drake propuso el primer problema en 1962, y en 1965 Karl Astrom amplió el concepto.

En MDP un entorno está modelado como un conjunto de estados y acciones que se pueden desempeñar para controlar el estado del sistema y el objetivo del MDP es decidir qué acción es la más beneficiosa. El agente debe tomar decisiones teniendo en cuenta el estado del mundo (entorno) y las acciones para cambiar su estado; debe aprender el valor de la recompensa y no dejarse engañar por las recompensas más pequeñas porque puedes obtener recompensas mayores al acumularlas. Esta forma de aprendizaje le ayuda a aprender comportamientos que necesitas recordar o recompensas que recibirás a largo plazo. Una vez que aprende a resolver una tarea en particular, el conocimiento que aprende es específico para esa tarea. De lo contrario: el agente calcula la política que maximiza el flujo de recompensas.

Las reglas del MDP se ajustan de manera adecuada para describir el juego de Hex, lo que convierte a los algoritmos basados en MDP en herramientas apropiadas para abordar este juego.

3.1.8 Deep learning

La inteligencia artificial es un conjunto de habilidades intelectuales simuladas por un sistema computacional. Una de las ideas principales detrás de esta disciplina es la capacidad de imitar el cerebro humano. No podemos olvidar mencionar a las grandes figuras de la informática que nos trajeron este concepto. Alan Turing fue el primero en desarrollar un test que pudiera comprobar si una maquina pueda ser indistinguible de un humano. En 1956 el conocido informático John McCarthy describió por primera vez este campo de la computación como inteligencia artificial.

Como ejemplo paradigmático, en 1996, la supercomputadora de IBM conocida como Deep Blue logró una victoria significativa al derrotar al gran maestro de ajedrez Gary Kaspárov en una partida histórica.

El aprendizaje automático, más conocido por su nombre en inglés Machine Learning, es una disciplina de la inteligencia artificial. Su objetivo es el desarrollo de programas

algorítmicos capaces de aprender por si solos a través de los datos recibidos. Si bien la inteligencia artificial trata de imitar la inteligencia humana para resolver problemas el aprendizaje automático trata de explorar uno de los rasgos de la inteligencia humana más interesantes: La experiencia.

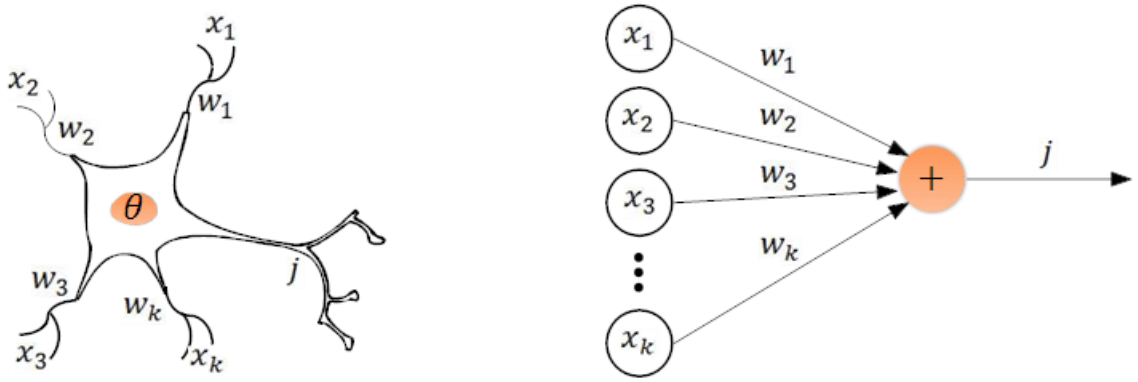
Arthur Lee Samuel (1901-1990) fue un pionero americano en el campo de los juegos por computadora y de la inteligencia artificial. El programa de juego de damas Samuel pararece ser el primer programa de autoaprendizaje del mundo, y como tal una temprana demostración del concepto fundamental de la inteligencia artificial (IA).

Deep learning consiste en un conjunto de técnicas que pretende ser aún más preciso que el machine learning para solucionar problemas complejos. La técnica principal y más popular son las redes de neuronas profundas que atraen la atención de la prensa y los fanáticos de lo tecnológico debido al interés de la gente por la ciencia ficción. El Deep Learning es posiblemente la disciplina que mejor engloba la necesidad intelectual del ser humano por crear una entidad artificial inteligente. Hoy en día existen muchas tecnologías como los chats tipo chat-GPT o los modelos generativos de arte.

El Deep Learning puede ser computacionalmente muy pesado por lo que en los últimos años se ha avanzado el uso de sistemas distribuidos en línea y el de las tarjetas gráficas.

Las neuronas artificiales que se suelen usar en inteligencia artificial es una imitación de una neurona humana real diseñada a partir de procesos matemáticos. En su fórmula podemos encontrar algo llamado pesos que son la parte de la neurona que va variando a lo largo del aprendizaje para dar los resultados deseados. Este proceso de aprendizaje es conocido como *backpropagation*. El primer modelo matemático de una neurona se lo debemos a Walter Pitts y a Warren McCulloch que lo publicaron en 1943.

Representación gráfica comparativa de una neurona real y una neurona artificial:



a) Neurona biológica

a) Neurona artificial

La función de activación es la parte de la neurona que procesa el resultado a la salida. Entre las cuales podemos encontrar la función RELU, softmax, sigmoidal y otras muchas más. La función RELU es una función con formula **y = max(0, x)** y es habitual utilizarla en las capas intermedias de una red.

Una red de neuronas es un método de inteligencia artificial que utiliza neuronas artificiales para resolver problemas. Las neuronas artificiales se organizan en capas. Hay tres partes en una red de neuronas: una capa de entrada; una o varias capas ocultas; y una capa de salida. Los datos de entrada se presentan en la primera capa, y los valores se propagan desde las neuronas hasta cada neurona de la capa siguiente. Al final, se envía un resultado desde la capa de salida. La red aprende realizando el algoritmo de backpropagation. Este proceso se repite muchas veces y la red sigue mejorando sus predicciones hasta que el desarrollador considere oportuno parar que por lo general es cuando haya terminado de pasar por todos los datos de entrada tantas veces como se le haya indicado.

El algoritmo de backpropagation es una herramienta usada por muchos de los modelos de aprendizaje automático, pero especialmente por las redes de neuronas que consiste en modificar los pesos de las neuronas desde la salida hasta la entrada haciendo uso de una fórmula matemática.

El algoritmo de blackpropagation tiene varias etapas diferentes. En la primera etapa se elige el punto de entrada y de salida. Después se asignan valores secundarios que le permitirán modificar parámetros dentro de cada capa de la red neuronal. Con el análisis de los nodos se determina el error total y con ello el algoritmo empieza a minimizar su

efecto en toda la red. Ahora ajusta reduce el error ajustando los pesos de las neuronas las veces que sean necesarias. Después de este entrenamiento, cuando se le presenten patrones de entrada arbitrarios o incompletos, la red de neuronas podrá encontrar la solución más adecuada porque podrá reconocer los mismos patrones y características que aprendió durante el entrenamiento.

El aumento de datos, comúnmente conocido por su nombre en inglés: *Data augmentation*, es el proceso de generar nuevos datos a partir de datos ya existentes con el objetivo de entrenar modelos de aprendizaje automático. El data augmentation se usa principalmente en procesamiento de imágenes y aprendizaje automático, específicamente en tareas de visión por computador, como clasificación de imágenes, detección de objetos y segmentación semántica, así como en el reconocimiento de gestos y acciones. En esencia, se emplea para generar variaciones de datos de entrenamiento y mejorar el rendimiento y la generalización de los modelos. Es importante considerar cuestiones éticas al crear datos sintéticos mediante el uso de data augmentation. Debe asegurarse de que la información no introduzca malentendidos o malas interpretaciones que puedan dar lugar a discriminación o decisiones equivocadas. También deberá respetar la confidencialidad y el anonimato de las personas cuyos datos se utilizan como base. Es importante comunicar y obtener el consentimiento, en su caso. También debe asegurarse de que los datos generados cumplan con las leyes de privacidad y protección de datos. Comprender el proceso de generación de datos sintéticos es fundamental para resolver estos problemas.

Transfer Learning es una herramienta de Deep Learning que utiliza un modelo entrenado para una tarea como punto de partida para otro modelo que realiza una tarea similar. Actualizar y volver a entrenar una red con Transfer Learning es más fácil y rápido que entrenarla desde cero. De hecho, se puede almacenar una red de neuronas en un fichero de texto. Esto es tremendamente versátil ya que permite que una red pueda aprender todas las veces que sean necesarias en una infinidad de situaciones diferentes.

Hay que mencionar que el modelo más popular de aprendizaje profundo para juegos de mesa es alphaGo que usa una red convolucional. Muchos de los agentes de hex utilizan alguna forma de Deep learning especialmente basados en los modelos de Go.

3.1.9 Aprendizaje por refuerzo

El aprendizaje por refuerzo o *reinforcement learning* es un paradigma que se encarga del aprendizaje en problemas de decisión secuenciales. Trata de mejorar el rendimiento a través de la prueba y error y eso permite que el modelo aprenda de sus experiencias. Entre las técnicas de aprendizaje por refuerzo más utilizadas para desarrollar modelos para Hex podemos encontrar el *Q-learning*. En este tipo de aprendizaje enseña cómo tomar decisiones para lograr mejores resultados. Simula el proceso de aprendizaje de prueba y error que utilizan las personas para lograr sus objetivos. Se fomentan las actividades informáticas que sean coherentes con su propósito, mientras que se ignoran aquellas que se desvían de ese propósito. Los algoritmos de aprendizaje por refuerzo utilizan el principio de recompensa y castigo al procesar datos. Aprenden de cada acción y descubren por sí mismos la mejor manera de procesarla para lograr el resultado final. El mejor curso de acción puede requerir sacrificios a corto plazo, por lo que el mejor resultado puede venir acompañado de algún castigo o revés. Este tipo de aprendizaje es una tecnología potente que permite que los sistemas de inteligencia artificial logren mejores resultados. El aprendizaje por refuerzo tiene como objetivo maximizar las recompensas a largo plazo adaptándolo a situaciones donde las acciones tienen consecuencias a largo plazo. Esto es especialmente adecuado para situaciones del mundo real donde no hay retroalimentación instantánea disponible en cada paso, ya que se puede aprender de las recompensas retrasadas. Se pueden aplicar en entornos complejos con muchas reglas y dependencias. Al mismo tiempo, incluso si una persona tiene un alto nivel de conocimiento sobre el entorno, no puede elegir el mejor camino. Por el contrario, los algoritmos de aprendizaje por refuerzo sin modelos pueden adaptarse rápidamente a entornos cambiantes y encontrar nuevas formas de optimizar los resultados. Para los algoritmos tradicionales de aprendizaje automático, se escriben dos conjuntos de datos para impulsar el algoritmo. Esto no es necesario cuando se utilizan algoritmos RL. Un algoritmo puede aprender por sí solo.

Q-Learning es una técnica de aprendizaje por refuerzo que trata de aprender una serie de normas que le describan que decisión tomar bajo unas circunstancias concretas.

Existe una versión llamada *Deep Q-Learning* que trata de resolver este problema utilizando redes de neuronas profundas.

NeuroHex es un agente que usa Deep Q-Learning para jugar a hex.

3.1.10 Imitation learning

Imitation Learning es un tipo de aprendizaje social que consiste en adquirir nuevos conocimientos mediante la imitación. Lo importante de este tipo de aprendizaje es que existen técnicas de aprendizaje automático que tratan de simular este tipo de aprendizaje. En la disciplina de machine learning este campo es conocido también como Learning from Demostration porque, a diferencia del aprendizaje por refuerzo que esta más basado en prueba y error, este aprendizaje le da al modelo la posibilidad de aprender a partir de un dataset creado por un agente humano que en este proyecto es el propio jugador.

A alto nivel 'Imitation Learning' intenta aprender una habilidad después de observar algunas demostraciones de un experto realizando la habilidad. Crucialmente, se espera que, esta habilidad aprendida, pueda generalizar a situaciones en las que el aprendiz no ha observado todavía las acciones del experto. Mas formalmente, sea S_t una representación de un set de observaciones obtenidos en un tiempo t que constituye un estado. Sea a_{et} una representación de un set de acciones (Por ejemplo, el voltaje enviado a los motores de un robot) cogido por el experto en tiempo t. Sea T una representación de una trayectoria que es un set de n estados, pares de acción de experto $[(S_t, a_{et}) \text{ for } 0 \leq t \leq n]$. Imitation Learning esencialmente asume que dado un m de estas trayectorias como dato de entrada. La salida esperada es una política $\pi\theta(at, st)$ (donde θ representa un set de parámetros de política) que mapea estados a acciones. Específicamente, se toma como una distribución de probabilidad que es una función de acciones para ser tomada y los estados observados en cada intervalo de tiempo. Crucialmente, Imitation Learning espera que la distribución de las acciones de la política de aprendizaje encajara con la distribución de acciones de los expertos (i.e. la distribución de donde salió a_{et})

Este concepto tiene sus aplicaciones. Enfocadas a situaciones en las que imitar a los humanos es más importante que buscar una respuesta correcta. Por ejemplo, en el

tráfico de una ciudad una inteligencia artificial que maneje un coche autónomo intentara mayormente seguir las normas de tráfico. El problema es que en un lugar con tanto tráfico y tantas situaciones aleatorias sucediendo a la vez es posible que recurra seguir las normas sobre la seguridad de los demás. Con imitation learning esa inteligencia artificial podría saber en qué momento puede saltarse las reglas para proteger la seguridad y la salud de la gente tal y como lo haría un humano de verdad. También se puede usar el imitation learning en los videojuegos aprendiendo de los jugadores un videojuego podría crear un jugador controlado por el pc que tenga el mismo nivel que el propio jugador.

El imitation learning tiene por delante aún un largo recorrido para poder compararse a otras disciplinas del aprendizaje automático. Por eso yo creo que se debe investigar más esta tecnología y esa es la razón por la que la he elegido en este proyecto de fin de grado. El mayor problema que tiene este tipo de aprendizaje es la calidad de los datos ya que aprender a solucionar un problema con demostraciones de nivel bajo perjudica al entrenamiento y genera modelos que no sirven para nada. Además, si no tiene suficientes datos tiene problemas a la hora de encontrarse con situaciones nuevas porque no sabe cómo se comporta un humano en esa situación.

Hay dos corrientes principales dentro del *Imitation Learning*: *Behavioral Cloning* e *Inverse Reinforcement Learning*.

Behavioral Cloning es una técnica de aprendizaje supervisado en la que el agente aprende a moverse con precisión en los estados en los que ha entrenado. Siendo especialmente defectuoso cuando se encuentra con estados nuevos.

Inverse Reinforcement Learning es una técnica de *machine learning* que trata de aprender a imitar a un humano a través de descubrir las funciones que le servirían para encajar el problema dentro de los *Markov Decision Processess*.

Existe un artículo realizado en la universidad de Stanford que asegura que el Deep Imitation Learning da increíbles resultados en juegos de estrategia en tiempo real como StarCraft II. Este modelo utilizaba CNN con cuatro capas convolucionales con funciones de activación ReLU. El modelo encontró maneras de atacar a sus enemigos, pero no avanzo a la hora de descubrir estrategias más complejas.

3.1.11 Convolutional neural network

Las redes convolucionales son un tipo de redes de neuronas utilizado principalmente en visión por computador. Las redes convolucionales permiten procesar matrices grandes de forma muy eficiente. Sin embargo, este tipo de redes maneja matrices más grandes de las que se usan en este proyecto ya que se utilizan en imágenes que no suelen bajar de 200 pixeles de lado mientras que los tableros de hex, aunque se puedan cambiar de tamaño nunca llegarían a niveles tan altos con tableros de 12x12 siendo ya excesivamente grandes.

El proceso que las matrices, generalmente imágenes, recorren al llegar a una red convolucional de neuronas está dividido en varias etapas:

El primer paso es el de los filtros. En este paso se le pasa a una imagen por una serie de filtros y modificaciones que tratan de resaltar los datos más importantes.

El segundo paso es propio uso de la convolución. La convolución es una operación matemática de algebra matricial. En este paso se trata de reducir las dimensiones de la imagen quedándose lo más importante de cada región.

El último paso es la propia red de neuronas y de pendiendo como este hecha podría usarse para detección de objetos u otros muchos análisis.

Al manejar matrices las redes convolucionales de neuronas son muy compatibles con los juegos de mesa que usen un tablero cuadrado o cubico. Por ello algunas de las tecnologías que veremos a continuación usan este tipo de redes.

Si existen trabajos de Imitation Learning que usan redes convolucionales, pero se usan para videojuegos donde la salida por pantalla es necesaria para poder aprender. Por ejemplo, en juegos de carreras donde la red tiene que sacar información no solo de su entorno sino también de la información obtenida por las imágenes que recibiría el propio jugador.

Más adelante en este trabajo se discutirá sobre KataHex, un agente de Hex que integra redes convolucionales con árboles de Monte Carlo

3.1.12 Hex GTP

Hex GTP también conocido como GTP de forma simplificada es una interfaz de texto utilizable en la consola de comandos y compatible con interfaces graficas. Permite a motores de estrategias como mopyhex conectarse a interfaces graficas como HexGui. Está basado en Go Text Protocol, un protocolo de texto creado para el juego Go como parte del proyecto GNU Go.

Dado que se utiliza este protocolo en el proyecto es este el lugar oportuno para hablar de los comandos de Hex GTP:

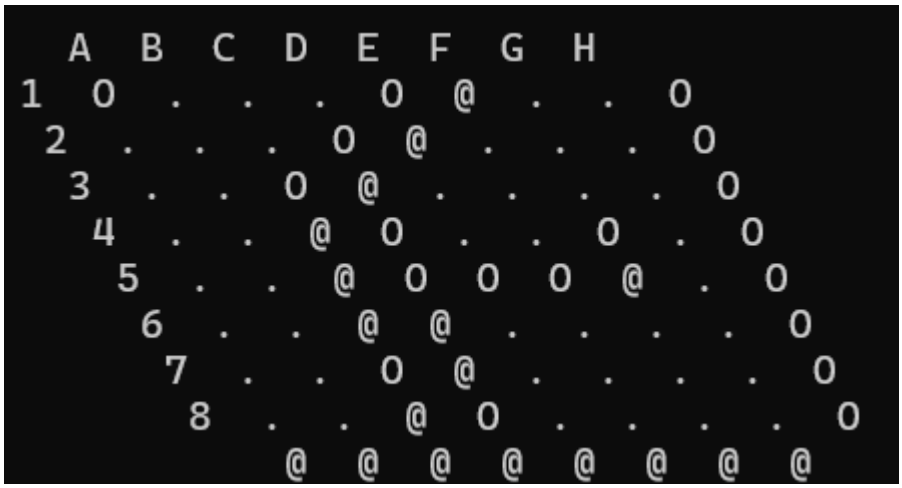
Los argumentos habituales son:

Argumento	Valores posibles
n	Un numero entero
player	Un jugador que puede ser ‘black’ / ’b’ / ‘B’ o ‘white’ / ‘w’ / ’W’
move	Una celda en formato letra / numero. Ej: ‘a1’, ’b2’, ‘h6’

Los comandos habituales son:

Comando	Explicación
list_command	Escribe una lista de todos los comandos
quit	Rendirse
boardsize	Cambia el tamaño del tablero con el ancho y el alto como argumentos y reinicia el tablero
clear_board	Reinicia el tablero
play	Realiza un movimiento. Para ello necesita el jugador y una celda
genmove	Realiza un movimiento generado por el motor de estrategia predeterminado. Para ello necesita un jugador
showboard	Muestra en formato texto la disposición del tablero
winner	Escribe por pantalla si hay ganador y quien es

Representación del tablero utilizando Hex GTP en consola de comandos con el jugador de las fichas negras como ganador:



3.2 Trabajos relacionados

En esta sección se examinarán diversas tecnologías empleadas en contextos similares. Entre ellas se incluyen interfaces desarrolladas específicamente para jugar Hex y motores de estrategia diseñados para este juego.

3.2.1 Virtualización de hex

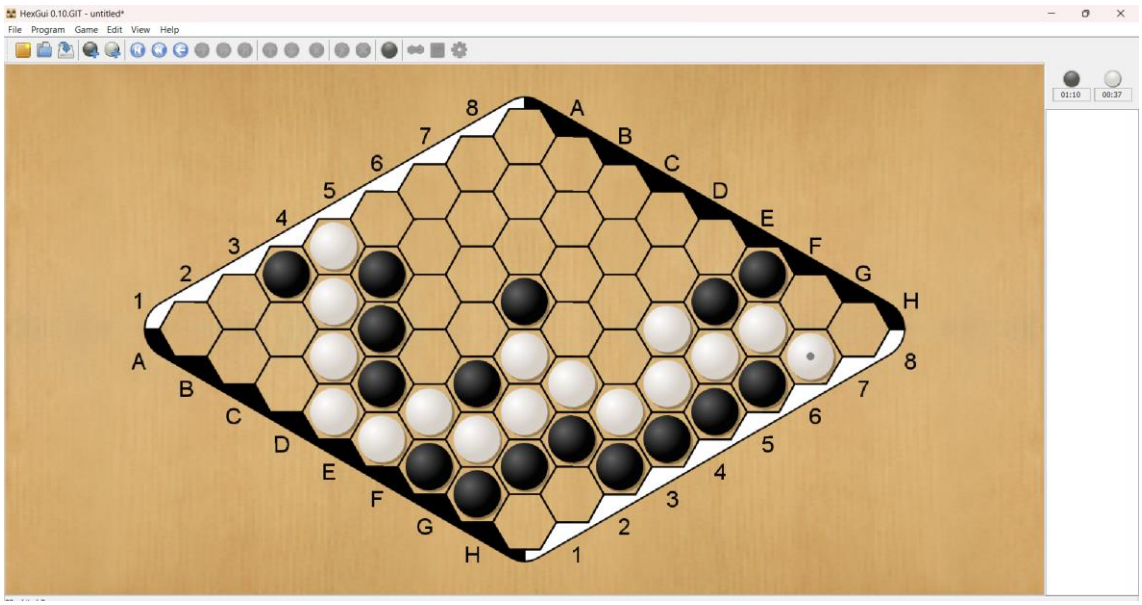
Hay unos cuantos proyectos que tratan de virtualizar a Hex. Entre los cuales se deben destacar MIHex y HexGui.

El sistema MIHex también conocido como *the machine learning Hex project* es una herramienta que automatiza aspectos de Hex. Fue creado para la escuela de ingeniería eléctrica y ciencia de la computación de la universidad de Newcastle en Australia para presentarlo como un ejercicio a sus alumnos en forma de torneo. Está pensada para que cualquier persona que quiera investigar usando inteligencia artificial con Hex no tenga que preocuparse por tener que programar el juego desde 0. La estructura de MIHex escrita en java es parte de un servidor capaz de albergar múltiples estrategias del juego

hex conectadas como clientes. Lastimosamente el código de este sistema es inalcanzable y solo el artículo prueba que alguna vez haya existido. Los demás trabajos de los que voy a hablar a continuación si tienen un código disponible en alguna plataforma por lo menos en la fecha de realización de este trabajo.

HexGui es una aplicación para mostrar una partida de Hex que permite introducir distintos modelos de juego. Creada por Broderick Arneson y escrita en java, HexGui puede manejar agentes que usen el protocolo de comunicación hex GPT. Permite incluso guardar y cargar partidas. Es una versión de GoGui, una aplicación de software de código abierto creada por Markus Enzenberger para virtualizar el juego Go.

Interfaz gráfica de HexGUI con el jugador de las fichas blancas como ganador:



3.2.2 Algoritmos de hex

Minimax es una técnica de decisión de la teoría de juegos. Esta estrategia inventada por John von Neumann en 1928 trata de minimizar la perdida máxima que sale de las decisiones que toma el jugador. Es importante que en las simulaciones no se deje al programa conocer los patrones desde el principio, sino que los aprenda jugando ya que si el jugador sabe de ello podría saber que estrategia usar para contrarrestarlo.

Alpha-beta pruning es un algoritmo de búsqueda que trata de reducir el número de nodos que tiene que evaluar el algoritmo de minimax. Este algoritmo se le atribuye a John McCarty y ha sido recreado por múltiples personas a lo largo de la segunda mitad del siglo pasado.

Existe un bot llamado Hex-bot creado con el algoritmo de minimax que puede jugar hex. Utiliza Alpha-beta y fue escrito en Python.

Monte Carlo tree search es un algoritmo de árbol heurístico de búsqueda muy utilizado en una gran cantidad de juegos de mesa. La forma en que actúa este algoritmo es generando un árbol de estados a través de una simulación de la partida seleccionando acciones al azar y luego decidiendo una acción entre las que den mejores resultados.

MCTS tiene muchos atractivos: es un algoritmo estadístico para el cual un gran poder computacional generalmente lleva a un mejor rendimiento. Puede ser usado con poco conocimiento sobre el tema y ha tenido éxito en problemas difíciles en donde otras técnicas han fallado.

El origen de este enfoque se encuentra en Abramson, quien, en lugar de utilizar una función de evaluación estática, combinó la búsqueda minimax con el modelo de probabilidad para generar aleatoriamente todas las jugadas. Este método se perfeccionó en trabajos posteriores hasta 1992, cuando se utilizó por primera vez en el juego de Go de una forma muy similar a su comprensión actual. En 2006, inspirado por estos trabajos, Remi Coulom describió por primera vez el uso generalizado del método Monte Carlo en juegos de búsqueda de árboles y acuñó el término búsqueda de Monte Carlo Tree Search.

En la edición de 2009 de las olimpiadas de computación gano MoHex. Un agente de hex que usaba el Monte Carlo tree Search y estaba a la par con las técnicas clásicas de alpha-beta.

Los modelos de Hex más tradicionales utilizan métodos de Monte Carlo. Entre ellos se pueden encontrar MoHex, Mopyhex y Yopt.

MoHex es un programa de software libre realizada en C++ del juego hex que utiliza los árboles de búsqueda de Monte Carlo. Desarrollado en la universidad de Alberta en Estados Unidos por Philip Henderson, Broderick Arneson y Ryan Hayward. MoHex usa el framework de UCT combinado con la heurística AMAF para seleccionar el mejor de

los nodos del árbol. MoHex juega mejor cuando apaga el UCT y se deja llevar únicamente por AMAF. Utiliza hex GTP y por lo tanto es compatible con hexGUI. Existe una versión en Python llamada mopyhex a la que dedico su propia sección.

Pero también se puede encontrar modelos más nuevos que combinan métodos de Monte Carlo con redes de neuronas. Entre estos modelos se encuentra KataHex.

KataHex usa arboles de Monte Carlo con redes convolucionales. Es libre y de código abierto y puede ganar a jugadores de alto nivel. Está basado en KataGo adaptado en 2022 por "HZY". Usa una versión fuera del estándar de GTP. KataHex ha sido entrenado únicamente jugando consigo mismo por lo que tiene algunos problemas en ciertas situaciones concretas. A día de hoy el modelo pre-entrenado se encuentra disponible en internet, así como el código fuente y una interfaz gráfica compatible.

Otro modelo de redes de neuronas que no usa Monte Carlo es NeuroHex.

NeuroHex es un modelo que usa Deep Q-Learning con una red convolucional. Al igual que mopyhex este también pertenece a Kenny Young y también se encuentra en internet a día de hoy. Esta red es capaz de aprender con auto juego y ha sido entrenado especialmente en tableros de 13x13.

3.2.3 Mopyhex

Mopyhex es una implementación del método de Monte Carlo para el juego hex en Python. Esta implementación hace uso de hex GTP que le permite conectarse a una interfaz gráfica como hexgui o incluso se puede usar en una consola de comandos.

No hay mucha información sobre este programa así que se va a analizar cómo funciona a partir de su código:

Lo primero que el agente hace es generar las simulaciones, representadas con la función search. Una vez actualizado el árbol se consigue el mejor nodo. El nodo es usado como el movimiento seleccionado actualizando el tablero.

Al utilizar el comando 'genmove player' lo primero que se va a ejecutar es una función que selecciona el nodo del árbol. Siguiendo lo que sabemos de los árboles de búsqueda

de Monte Carlo sabemos que el mejor nodo es, de hecho, la casilla del tablero de juego que durante las simulaciones se ha usado con mayor frecuencia.

Una vez seleccionado una casilla se procede a realizar el movimiento. Al hacerlo se actualiza el árbol seleccionando el siguiente hijo. Si el movimiento no está en los hijos del árbol MCTS tiene que reiniciar el árbol.

Después de analizar este agente he llegado a la conclusión de que es el modelo ideal para imitar y hacer las pruebas. Aunque no siempre realice los mismos movimientos siempre usa las mismas estrategias.

3.2.4 Imitation learning

Imitation es una librería de Python pensada para que cualquier investigador utilice algoritmos de Imitation Learning. Estos son los algoritmos que están incluidos en la librería Imitation:

1. Behavioral Cloning: el modelo aprende directamente una política mediante el uso de aprendizaje supervisado en pares de observación-acción a partir de demostraciones de expertos.
2. DAgger: igual que el behavioral cloning entrena con aprendizaje supervisado con demostraciones de expertos. A diferencia del anterior, DAgger entrena con un dataset que se asemeja más a las observaciones que se va a encontrar.
3. Density-Based Reward Modeling: es una técnica de inverse reinforcement learning que trata de dar mejores recompensas a las decisiones que más se repiten en los datos de entrada.
4. Maximum Causal Entropy Inverse Reinforcement Learning: Es una implementación del principio de máxima entropía causal.
5. Generative Adversarial Imitation Learning: También conocida como GAIL. Esta red adversaria entrena junto a un discriminador para diferenciar entre las decisiones que son de un experto y las que no lo son.
6. Adversarial Inverse Reinforcement Learning: igual que el anterior. Esta más preparado para los cambios en el entorno.

7. Deep RL from Human Preferences: el algoritmo aprende una función de recompensa de preferencias entre pares de trayectorias. Las comparaciones se modelan como generadas a partir de un modelo Bradley-Terry (o racional de Boltzmann), donde la probabilidad de preferir la trayectoria A sobre B es proporcional a la exponencial de la diferencia entre el retorno de la trayectoria A menos B. En otras palabras, la diferencia en los rendimientos forma un logit para un problema de clasificación binaria y, en consecuencia, la función de recompensa se entrena utilizando una función de pérdida de cross-entropy para predecir la comparación de preferencias.
8. Soft Q Imitation Learning: aprende a imitar a través de demostraciones usando el algoritmo de Deep Q lerning. Durante el proceso de aprendizaje las demostraciones de experto son recompensadas con un valor de 1 mientras las demostraciones del entorno tienen asignadas una recompensa de 0. Esto enseña al algoritmo a parecerse a las demostraciones y evitar las que no se parezcan.

4 Aspectos metodologicos

4.1 Sprints

Este TFG es un trabajo que ha sido planteado en tres sprints. El primer sprint estaba enfocado a la investigación de las tecnologías que pudieran estar relacionadas. El segundo trataba de solucionar los requisitos necesarios para poder desarrollar el proyecto y el tercero está dedicado al propio motor de estrategia del juego.

4.1.1 Sprint 1

Este sprint ha estado enfocado en la investigación de tecnologías que podrían resultar útiles para el proyecto. Es en este sprint en el que se empieza a cuestionarme qué tipo de proyecto quiero hacer. Desde el principio suponía que debía usar un agente basado en redes de neuronas.

Al investigar distintos métodos es fácil darse cuenta de que el agente más recomendado y por lo tanto el mejor está basado en *Monte Carlo tree search.*

Al explorar otros agentes de Hex, se constató la escasez de aquellos basados en redes neuronales, y no se identificó ninguno que satisficiera plenamente los requisitos del presente estudio. Por ende, esta situación representó una oportunidad para desarrollar un modelo capaz de emular el comportamiento humano en este juego.

En cuanto a aprendizaje por imitación, comúnmente llamado como *imitation learning*, me encontré con apenas una librería en Python con la que poder trabajar.

4.1.2 Sprint 2

Como parte de este sprint investigue interfaces gráficas y otras formas de encajar mi futuro código con el juego hex planteándome incluso realizar mi propio juego. Esta parte estuvo dedicada a resolver los objetivos necesarios, pero no principales como el propio juego hex y su interfaz gráfica.

Se destaca la presencia de Hex GTP, un protocolo de texto que facilita la interacción con el juego Hex y su conexión con interfaces gráficas. Se agradece a los creadores de Mopyhex por proporcionar el código fuente.

En mopyhex no solo se puede encontrar una implementación de hex GTP que podría usar en este TFG, sino que, además, se identificó un modelo de Monte Carlo con el cual se podrían contrastar los resultados obtenidos.

En este sprint, se identificó una interfaz gráfica que se propone como una opción viable para aquellos que deseen jugar al Hex. HexGui es una interfaz sencilla y accesible, capaz de gestionar cualquier agente de Hex, siempre y cuando se comunique utilizando HexGTP.

4.1.3 Sprint 3

Ahora es turno de desarrollar el agente de imitación. Tras un periodo de pruebas se ha llegado a la conclusión de que es preferible que, para este proyecto, se utilicen modelos de aprendizaje por imitación de fabricación propia.

El desarrollo del propio agente llevó a la realización de múltiples pruebas con distintos modelos algunos sacados de la propia librería Imitation ya mencionada antes con la idea de desarrollar un agente original que pudiera ajustarse a este problema. Al final se decidió por una red de neuronas de Keras capaz de modificarse si el tamaño de los datos de entrada no era lo suficientemente grande. Esta decisión fue tomada porque para poder desarrollar este trabajo se entendió que se debía no depender de librerías que hicieran el trabajo sin controlar cómo funcionan si no hacerlo con las herramientas que he recibido a lo largo de la carrera.

Durante las pruebas del modelo se hicieron los ajustes necesarios a la red para realizar esta tarea. Se descarto por completo la posibilidad de usar una red convolucional ya que una convolución afectaría demasiado al tablero de entrada y no dejaría que el agente aprendiera correctamente.

También en esta etapa se planteó el uso de data augmentation para incrementar la tasa de aprendizaje del modelo. Finalmente se repitió el aprendizaje de cada movimiento en un número determinado que ha ido cambiando durante la etapa de aprendizaje final según el experimento al que se le ha sometido.

La red ha sido entrenada de varias maneras para comprobar su evaluación. Se le ha entrenado mediante el único uso de sus propios movimientos y aprendiendo de jugadores reales humanos. También se le ha enseñado a jugar imitando a otro agente basado en el algoritmo de árboles de búsqueda de Monte Carlo.

4.1.4 Sprint 4

En este sprint se ha desarrollado un nuevo modelo mejorado, se ha optimizado el modelo anterior y se han comparado los distintos modelos de hex.

Este nuevo modelo es una versión mas compleja que el anterior que usa dos redes de neuronas idénticas para elegir la fila y la columna de la ficha a colocar por separado. Para su entrenamiento se ha realizado el mismo tipo de data augmentation y se ha entrenado con el mismo algoritmo de Monte Carlo.

El modelo anterior a sido optimizado tomando una métrica de rendimiento distinta. El MAE. Que a diferencia del accuracy es una métrica mucho mas adecuada para evaluar un modelo de imitación.

Tambien se ha hecho uso de Google Colab ya que permite mostrar graficas que serán utilizadas para realizar comparaciones entre las versiones del modelo de imitación.

Se han comparado los distintos modelos con el propósito de analizarlos individualmente. Para ello se ha llevado acabo una serie de simulaciones en partidas con tableros de 8x8 en las que se han enfrentado entre ellas no solo para analizar cual es la mejor sino tambien comparar sus comportamientos.

4.2 Tecnologías

Durante el desarrollo de este trabajo de fin de grado se han utilizado diversas tecnologías.

4.2.1 Python 3

Python es un lenguaje de programación de alto nivel de propósito general. Es un lenguaje interpretado, orientado a objetos e interactivo.

Python fue revelado al público en 1991 por su creador, Guido van Rossum. Durante este desarrollo ya existían clases con herencia, manejo de excepciones, funciones y tipos modulares: list, dict, str, etc. Además, en esta primera versión apareció el sistema modular adoptado en Modula-3; van Rossum describió este módulo como "una de las unidades de programación más grandes en Python". Los patrones de excepción en Python son similares a Modula-3, pero con una cláusula adicional agregada. En 1994, se fundó comp.lang.python como el principal foro de discusión de Python, y esto supuso

un hito en el crecimiento de la base de usuarios del lenguaje. Python alcanzó la versión 1.0 en enero de 1994. Parte de esta versión son herramientas de programación funcionales: lambda, map, filter y reduce.

La versión que ha sido utilizada en el desarrollo de este proyecto es la 3.8 en la computadora y 3.10 en Google Colab.

4.2.2 Pycharm

PyCharm es un entorno de desarrollo integrado (IDE) creado para Python propiedad de la compañía jetbrains. PyCharm es multiplataforma, con versiones para Windows, macOS y Linux. La Community Edition (edición comunitaria) se publica bajo la Licencia apache, y también hay una Professional Edition (edición profesional) con características adicionales publicada bajo una licencia propietaria financiada por suscripción y también una versión educativa. La versión que se ha utilizado es la Community, pero solo se ha utilizado como editor de texto ya que para ejecutar el código se ha usado la consola de Anaconda a pesar de que esta es compatible con PyCharm.

4.2.3 Anaconda

Anaconda es una distribución abierta y libre de Python diseñada para agilizar la utilización de técnicas de aprendizaje automático. Para este proyecto se ha usado la consola de comandos de la versión conda 23.1.0

Anaconda fue fundado en 2012 por Peter Wang y Travis Oliphant por la necesidad de traer Python dentro del business data analytics. Desde ese momento, el uso de Python ha aumentado enormemente convirtiéndose en el lenguaje más popular hoy. Anaconda tiene ahora más de 300 empleados en todo el mundo y está orgulloso de ser usado por más de 40 millones de usuarios.

4.2.4 Keras

Keras es una librería de Python para el uso de redes de neuronas y conecta Python con la librería tensorflow. Keras fue creado originalmente por François Chollet para el proyecto ONEIROS. Keras tiene muchas implementaciones de arquitecturas de redes neuronales, como por ejemplo los layers, funciones objetivo, funciones de activación, optimizadores matemáticos. Su código está alojado en GitHub y existen foros y un canal de Slack de soporte. La versión utilizada de keras en este trabajo es la 3.1.1.

TensorFlow es una librería libre de código abierto para aprendizaje automático e inteligencia artificial creada por el equipo Google Brain para el uso interno y es la librería que keras utiliza para realizar los modelos de Deep Learning.

4.2.5 HEX GTP

Hex GTP es un protocolo de texto que permite al usuario jugar al juego hex mediante una consola de comandos o conectándolo a una interfaz gráfica que sea compatible. La versión utilizada de hex GTP es una modificación de la que se puede encontrar en el repositorio de github de mopyhex como ya he mencionado anteriormente.

Este protocolo sigue un modelo cliente servidor. El formato del comando consiste en una simple palabra que puede ir acompañada de parámetros. La respuesta consiste en una o más líneas de texto que empiezan por el carácter ‘=’ si es exitoso y ‘?’ si no lo es.

4.2.6 La computadora

La computadora utilizada para la realización de este proyecto es el propio ordenador portátil personal del autor de este trabajo. El cual es un lenovo ideapad gaming Windows 11 con un procesador AMD R YZEN 5000 y una tarjeta gráfica nvidia geforce gtx.

4.2.6 Google colab

Para el desarrollo de ciertas pruebas se ha utilizado Google colab. Colab es un servicio web que pertenece a Google creado con la intención de manejar notebooks de Python y R gracias a tecnologías cloud.

5 Desarrollo del trabajo

En este apartado se va a analizar el método utilizado para el desarrollo de este problema. El proyecto ha sido realizado en Python utilizando keras. El proyecto aprovecha el código de MoPyHex para la utilización de hex GTP modificándolo para adaptarlo a mi agente propio original. El tamaño del tablero utilizado por defecto y utilizado durante todo el desarrollo de este proyecto es de 8x8.

5.1 La primera versión

5.1.1 Red

La red de neuronas utilizada es una red secuencial con 5 capas ocultas y una capa de entrada con un tamaño ajustable al tamaño del tablero.

```
"se crea la red de neuronas que se va a utilizar con tantas neuronas
en la capa de entrada como grande sea el tablero"
self.model.add( keras.Input( shape = ( self.dim,)))
self.model.add( Dense( self.dim, activation = 'relu'))
self.model.add(Dense(64, activation='relu'))
self.model.add(Dense(64, activation='relu'))
self.model.add(Dense(64, activation='relu'))
self.model.add(Dense(64, activation='relu'))
self.model.add( Dense( 16, activation = 'relu'))
self.model.add( Dense( 8, activation = 'relu'))
"queremos que la salida sean dos(fila y columna) y con funcion de
activacion relu para sacar numeros enteros enteros positivos"
self.model.add( Dense( 2, activation = 'relu'))
self.model.compile( los = 'mse', optimizar = 'adam', metrics =
```

```
['accuracy'])
"cargamos pesos anteriores"
self.model.load_weights( 'model' + str( self.dim))
print( self.model.summary())
```

Hay que destacar la capa de salida. La capa de salida debe poseer dos neuronas lo que nos dará como resultado dos salidas las cuales serán interpretadas como una casilla indicada por las coordenadas. La función de activación es RELU ya que se necesita que dé un numero entero mayor o igual de 0.

Otro dato importante es que al crear la red cargo los pesos de una red anterior lo que me permite seguir entrenando la red, aunque el juego haya sido cerrado. A esta técnica se la conoce como transfer learning y con la configuración correcta se podría utilizar para que la misma red sirva en distintos tamaños guardando las capas ocultas y de salida y creando nueva solo la capa de entrada y alguna capa de más entre la capa de entrada y las capas ocultas en caso de que el tamaño sea más grande de lo normal y sea necesaria una red más compleja.

Ejemplo de una red de neuronas para un tablero de 8x8

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                4160

 dense_1 (Dense)             (None, 64)                4160

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 64)                4160

 dense_4 (Dense)             (None, 16)                1040

 dense_5 (Dense)             (None, 8)                 136

 dense_6 (Dense)             (None, 2)                 18

=================================================================
Total params: 17834 (69.66 KB)
Trainable params: 17834 (69.66 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

5.1.2 El entrenamiento

Cada vez que se mueve una ficha se entrena el modelo independientemente de que jugador o agente haya movido la ficha. Este modelo va a entrenar incluso con sus propios movimientos. Los datos de entrenamiento están compuestos por unos datos de entrada que no es otra cosa que la propia matriz del tablero y como datos de validación el propio movimiento.

```
  def move( self, move):
      """
mover ficha elegida y entrenamiento
"""

      trainX = self.anterior.board.flatten().astype( int).tolist()
      trainy = move
      self.databaseX = pd.DataFrame( [trainX])
      self.databaseY = pd.DataFrame( [trainy])

      "se realiza data augmentation"
      for i in range( 100):
          X_train = pd.concat( [self.databaseX,
pd.DataFrame([trainX])], ignore_index = True)
          y_train = pd.concat( [self.databaseY,
pd.DataFrame([trainy])], ignore_index = True)


      "se entrena con cada movimiento"
      self.model.fit(X_train, y_train, epochs=100, verbose=0)

      self.rootstate.play(move)
```

Para que los movimientos los aprenda bien es necesario realizar un data augmentation en este caso generando más instancias de todos los movimientos. En este caso se ha elegido reproducir el mismo movimiento 100 veces. Esto hará que la red se aprenda mejor los movimientos del jugador con mayor facilidad. El modelo corre el riesgo de aprender de un jugador que no sepa jugar y eso hará que el modelo no juegue a un nivel alto.

Una característica propia de las redes de neuronas es la posibilidad de entrenar todas las veces que se quiera. Eso permite que la red pueda aprender de todos los movimientos que hayas hecho y de todos los que vayas a hacer.

5.1.3 Generar un movimiento usando el agente

Para generar un movimiento lo único que se necesita es decirle al modelo que lo prediga utilizando nada más que como dato de entrada el propio tablero. Esto no garantiza que el resultado sea aceptable, es posible que genere movimientos que estén fuera del tablero o que intente poner en casillas que estén ocupadas, aunque con un buen entrenamiento no tiene porqué pasar siempre hay el riesgo de que pase. Para solucionar este problema se realizan una serie de seguros. Para que el movimiento predicho no se salga del tablero por arriba se realiza la operación resto (% en Python) respecto del tamaño del lado del tablero a ambas coordenadas, aunque es capaz de poner el resultado en los valores permitidos solo con los valores de entrenamiento sin necesidad de ponerle un seguro y por debajo no se va a salir nunca porque la función de activación RELU lo impide. Para evitar que elija una casilla que ya está ocupada es importante hacerle entender al modelo que no lo puede hacer. Por ello cada vez que lo haga se seleccionara una casilla al azar de las disponibles y se le entrenara al modelo con ella. Lo habitual en caso de sacar una casilla ocupada es elegir una casilla adyacente que este libre, pero en esta versión se ha decidido penalizarlo eligiendo una casilla al azar.

```
  def best_move( self):
      """
predice un movimiento usando el modelo y lo devuelve.
"""
      if (self.rootstate.winner() != gamestate.PLAYERS["none"]):
          "cuando termina el juego guarda los pesos de las neuronas"
          self.model.save_weights( 'model' + str( self.dim))
          return gamestate.GAMEOVER


      board = [self.anterior.board.flatten().astype( int).tolist()]
      "selecciona el movimiento"
      bestmove = self.model.predict( board)
      "nos aseguramos que los resultados esten en los valores
aceptables"
      resul = ( bestmove[0][0].astype( int) % self.size,
bestmove[0][1].astype( int) % self.size)
      if( resul in self.rootstate.moves()):
          return resul
      else:
          """para evitar que seleccione una casilla elegida cada vez
que lo haga le
          penalizamos seleccionando una casilla al azar.
          este movimiento al azar tambien formara parte de su
entrenamiento
          para que aprenda a evitar casillas incorrectas """
```

```
            print( "casilla ya seleccionada")
            print( "seleccionando casilla aleatoria de las disponibles")
            resul = random.choice( self.rootstate.moves())
            return resul
```

Este método de Python para generar una solución también tiene la posibilidad de guardar los pesos de la red de neuronas tras todo el entrenamiento en un fichero. Para ello se debe intentar generar un movimiento una vez que la partida tiene un ganador. La intención con esto es guardar la red únicamente cuando se ha realizado una partida valida, es decir, una partida con un ganador.

5.1.4 El agente

En definitiva, este es un agente para el juego Hex adaptado para poder usarse con el protocolo hex GTP con un motor de estrategia que consiste en una red de neuronas inspirada en aprendizaje por imitación.

5.1.5 Justificación del modelo

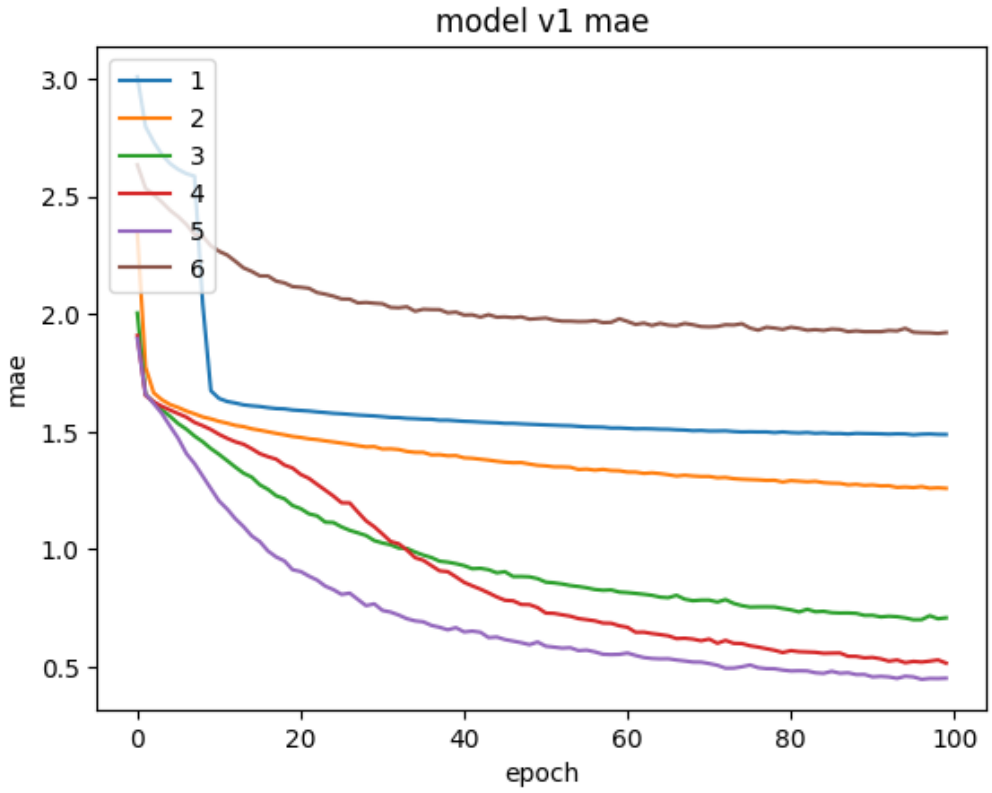
Una vez visto el modelo es necesario justificarlo. Se considera que hay que ver cómo funciona el modelo primero para entender por qué se ha elegido. Para la elección de este modelo se realizaron pruebas utilizando una serie de criterios que los pusieran en las mismas condiciones. Todos los modelos se han entrenado en 5 partidas simuladas usando el algoritmo de Monte Carlo con un data augmentation de 1000. Las diferencias entre los modelos se observan en el número de capas ocultas siendo 1 el mínimo y 4 el máximo y en el número de neuronas en esas capas siendo estas 8, 16 y 32. Se han probado dos combinaciones de modelos por cada número de capas. Los pesos iniciales en las neuronas han sido los creados por defecto. Los criterios para elegir han sido: La capacidad para no elegir una casilla ocupada y el accuracy.

Finalmente se ha elegido una red de dos capas ocultas, la primera de 16 y la segunda de 8 que han dado un accuracy superior al resto y su porcentaje de casillas ya ocupadas era lo suficientemente bajo como para seleccionarlo.

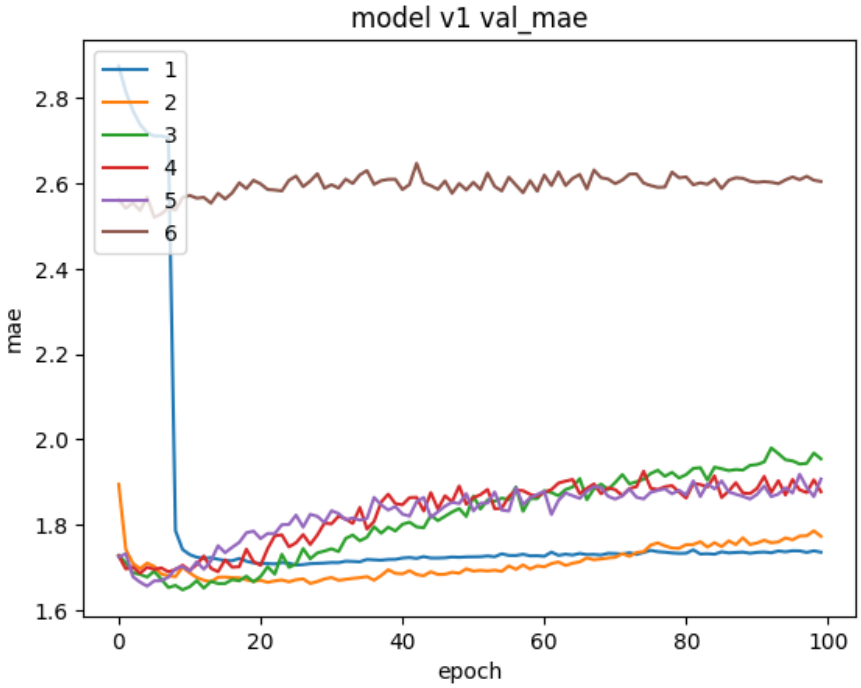
Al ver que el modelo no era lo suficientemente complejo y bueno utilizando otras métricas como el error absoluto medio se han realizado mas pruebas en busca de encontrar un modelo mas apropiado. Tras varias comparativas se ha llegado a la conclusión que un modelo de 5 capas ocultas tiene la complejidad mas apropiada para solucionar este problema introduciendo un MAE inferior a 1 y un MAE con los datos de validación inferior a 2.

El numero de neuronas a sido elegido en una comparativa tras elegir el numero de capas incrementando las neuronas de las capas según un esquema escalonado de tal forma que la siguiente capa tenga igual o menos que la anterior y limitando la primera capa al tamaño del tablero. El numero de neuronas en cada capa no ha generado modelos muy distintos y por lo tanto se ha dejado la configuración utilizada en la comparativa anterior que consistía en una capa de entrada del tamaño del tablero, una capa de salida de 2 neuronas y tres capas de 64 seguidas de una de 16 y una de 8.

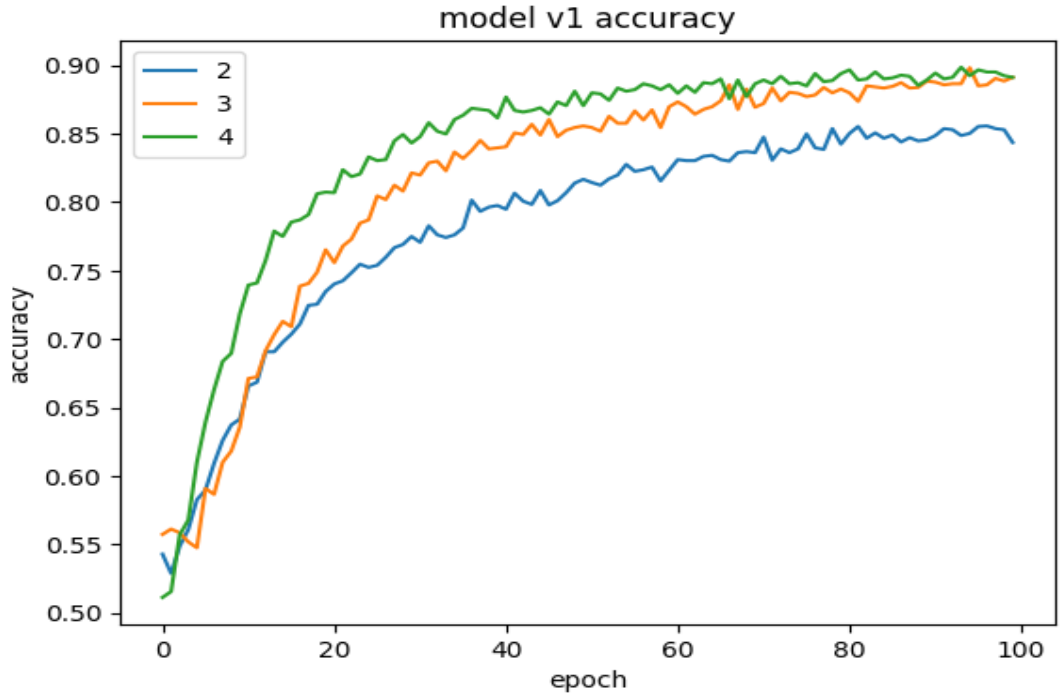
Grafica mostrando los resultados del MAE a lo largo de 100 épocas dependiendo del número de capas ocultas:



Grafica mostrando los resultados del MAE con los datos de validación a lo largo de 100 épocas dependiendo del número de capas ocultas.



Grafica mostrando los resultados del accuracy en un periodo de 100 épocas para los modelos de 3,4 y 5 capas ocultas:



Entre los posibles modelos hay que destacar que con más de 5 capas el MAE sube demasiado. Los modelos de 1 y 2 capas ocultas tienen un MAE más elevado con los

datos de entrenamiento y más bajo con los datos de validación lo que hace que imite peor, pero generalice mejor. Los modelos de 3,4 y 5 capas ocultas han demostrado bajar de 1 en el MAE con datos de entrenamiento por lo que son los mejores para imitar, pero no son tan buenos para generalizar. El objetivo de este proyecto está más enfocado en la imitación por eso se han elegido los modelos de 3,4 y 5 capas.

5.2 La segunda versión

5.2.1 La red

Para este modelo se han usado dos redes idénticas. Las redes contienen una capa de entrada con un parámetro de entrada para cada casilla del tablero. 3 capas ocultas con una configuración elegida tras varias pruebas. Y como capa de salida una neurona con función de activación RELU.

Ejemplo de una red de neuronas para un tablero de 8x8

```
Model: "sequential_30"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_194 (Dense)           (None, 64)                4160

 dense_195 (Dense)           (None, 64)                4160

 dense_196 (Dense)           (None, 16)                1040

 dense_197 (Dense)           (None, 8)                 136

 dense_198 (Dense)           (None, 1)                 9

=================================================================
Total params: 9505 (37.13 KB)
Trainable params: 9505 (37.13 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

5.2.2 El entrenamiento

El entrenamiento de este modelo es similar al anterior. Al igual que al anterior se ha llegado a la conclusión de aplicar un data augmentation de 100 y también aprende de los movimientos de su rival y los suyos propios.

5.2.3 Generar un movimiento usando el agente

Hay ligeras modificaciones en este apartado respecto del modelo anterior. En caso de elegir una casilla que este ocupada elige una de las casillas libres adyacentes. Esto es una práctica muy común para cualquier modelo de inteligencia artificial para teoría de juegos. Esto genera un modelo mucho mas defensivo ya que al imitar al rival suele apuntar a las casillas adyacentes lo cual es una practica muy recomendada en hex.

Ahora también guarda los pesos de la red cada vez que se entrena independientemente de si la partida es valida o no lo que le permite aprender más.

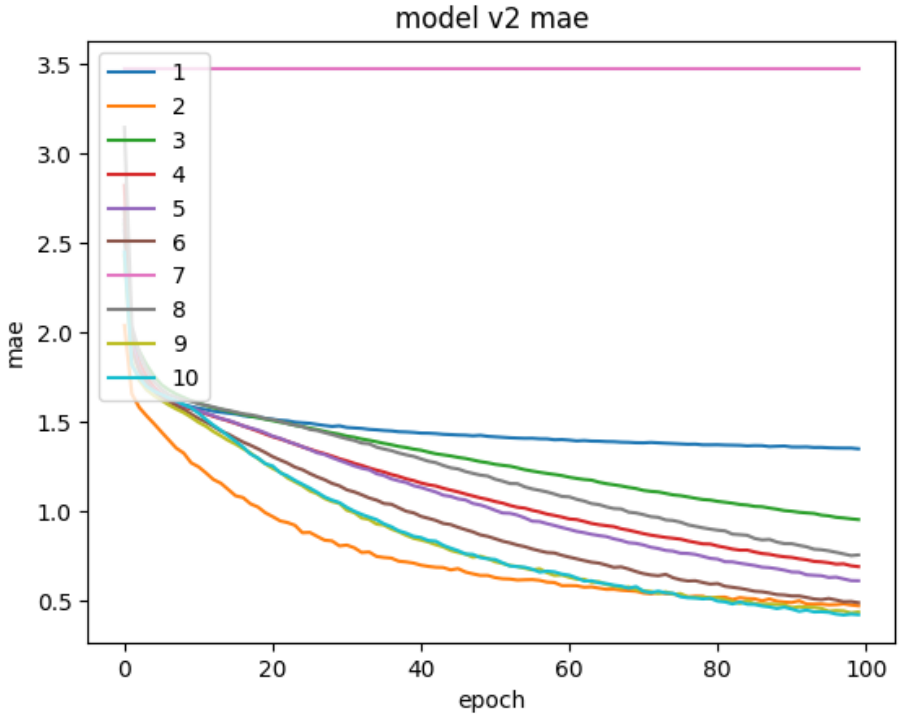
5.2.4 El agente

Este agente es mas avanzado y complejo que el anterior mas pensada para mejorar en la propia partida que tras varias partidas. Este modelo es peor imitando una estrategia en comparación con el anterior, pero es mejor para adaptarse al nivel del rival lo que lo hace un buen modelo defensivo para este juego.

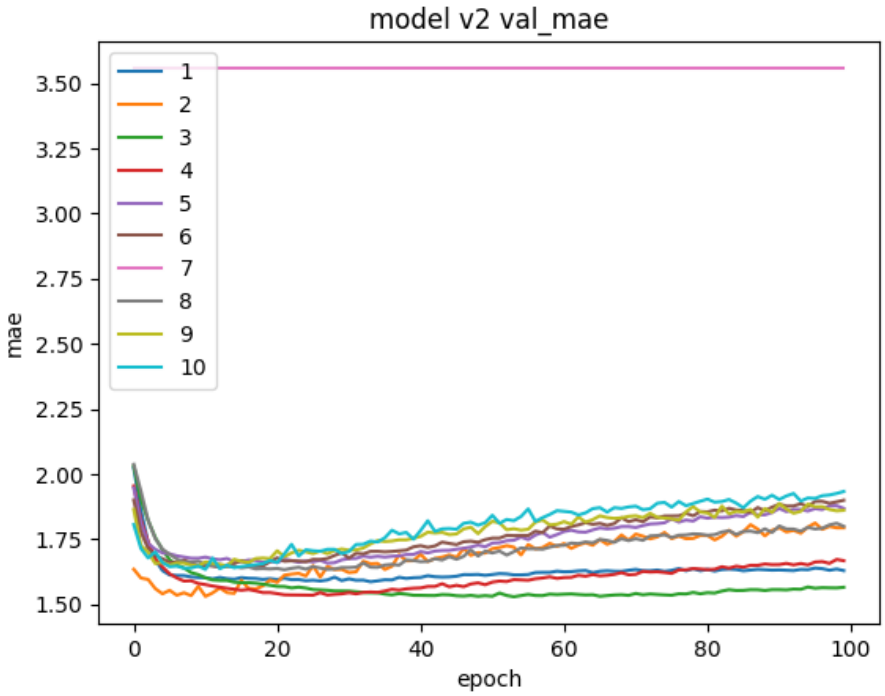
5.2.5 Justificación del modelo

Para la elección de este modelo se realizaron pruebas utilizando una serie de criterios que los pusieran en las mismas condiciones. En esta decisión se ha utilizado como métrica el MAE ya que se ha llegado a la conclusión que es una métrica mas apropiada para este modelo.

Grafica que muestra el MAE a lo largo de las épocas en función del número de capas:



Grafica que muestra el MAE de con los datos de validación a lo largo de las épocas en función del número de capas:



Al realizar las comparaciones de los distintos posibles modelos se ha llegado a la conclusión que el modelo de 3 capas ocultas es el mas adecuado ya que posee el MAE mas bajo tanto con los datos de entrenamiento como los datos de validación.

6 Resultados

6.1 Versión 1

Para demostrar que este modelo puede ser viable se han realizado distintos entrenamientos y se ha tratado de competir contra ellos utilizando la propia habilidad como jugador de varias personas humanas para evaluarlo tanto empezando primero como empezando en segundo lugar. Y luego se ha comparado los resultados con el método de Monte Carlo de mopyhex.

6.1.1 Auto entrenamiento

La primera prueba a la que fue sometido fue la del auto entrenamiento. En esta prueba todos los movimientos que aprendió la red eran generados por la propia red. En este experimento se simularon cien partidas (15 minutos por partida) de entrenamiento con un data augmentation de 10 por movimiento lo que da un data set de 1000 movimientos.

En la evaluación final con jugadores humanos cometía errores estratégicos como empezar por los laterales y no poner la ficha ganadora cuando tenía oportunidad. En esta evaluación no gano ninguna partida.

Este método de entrenamiento a probado no ser apto para este modelo.

6.1.2 Entrenamiento con jugador

Esta prueba es la principal ya que el objetivo real de este proyecto es ver si este modelo es capaz de imitar a los humanos. En este experimento se han realizado 25 partidas (5 minutos por partida de media) con un data augmentation de 40 lo que nos da un tamaño de 1000 en el data set.

Durante el experimento he podido ver como la red no cometía los errores del experimento anterior con la misma frecuencia a medida que avanzaban las pruebas.

En la evaluación final no terminaba de realizar los movimientos correctos, pero ya parecían mejor calculados. Pero no ganó ninguna partida.

Este método de entrenamiento es insuficiente debido a que necesita realizar muchísimas mas partidas para entrenar el modelo. Se recomienda utilizar un modelo ya entrenado para que se noten mejor los resultados.

6.1.3 Entrenamiento con mopyhex

Se supuso que le faltaban partidas para poder imitar correctamente a otro jugador por lo que se puso una simulación de mopyhex de 100 partidas con un data augmentation de 100. Lo que da un data set de 20000. Se quería que la red supiera comportarse de la misma forma que mopyhex.

Durante el periodo de entrenamiento se pudo ver las estrategias propias del algoritmo de Monte Carlo. Lo que permitirá comprobar si realmente ha aprendido algo. Se debe decir que el algoritmo de Monte Carlo es bastante rápido ya que puede ganar contra sí mismo en 3 minutos tardando 10 segundos en realizar un movimiento mientras que el modelo de aprendizaje por imitación con auto entrenamiento podía realizar un movimiento entre 1 o 2 segundos. El modelo era capaz de jugar sin cometer muchos errores.

En la evaluación final se ha podido probar que la red de neuronas ha aumentado su nivel de juego ganando en un par de ocasiones. Sin embargo, aun cometía muchos errores sobre todo cuando los agentes humanos intentaban cosas que no encajaban dentro de las estrategias que la red conocía.

Este método de entrenamiento vuelve a este modelo bueno cuando se enfrenta al algoritmo de Monte Carlo ya que sus jugadas ya las conoce, pero es vulnerable a movimientos desconocidos.

6.1.4 Imitation learning vs monte carlo

En este último experimento se quería ver la eficacia del anterior modelo que imitaba al algoritmo de Monte Carlo contra el algoritmo verdadero de Monte Carlo. Para ello se ha utilizado el propio mopyhex.

Durante las simulaciones se pudo comprobar el algoritmo de Monte Carlo da mejores resultados, pero el algoritmo de imitación es capaz de ganar en contadas ocasiones.

6.1.5 Evaluación

La técnica utilizada para evaluar este modelo es el MAE. El cuál es el grado de error de la predicción respecto al resultado real. El MAE está sujeto a la interpretación del observador y cuanto más bajo mejor. En este caso se ha interpretado que un MAE inferior a 2 es aceptable y si es inferior a 1 es sobresaliente.

Aquí se puede ver la fórmula del MAE. Donde n es el número de resultados, Yi es la predicción y la $\hat{Y}$i es la predicción:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |Y_i - \hat{Y}_i|$$

Estas evaluaciones que se han realizado han tomado el modelo entrenado haciendo uso de mopyhex. Es decir, se han simulado 1000 partidas de hex usando el modelo de Monte Carlo en ambos jugadores tardando un total de 4 días y se han guardado para su

uso en Google Colab donde se ha aplicado un data augmentation basado en rotar el tablero.

Una vez recolectados los datos se ha procedido a evaluar el MAE. El resultado del MAE en este modelo ha sido de 0.5 aproximadamente con los datos de entrenamiento lo que quiere decir que sus resultados se desvían en menos de 1 casilla en las predicciones que conoce y de 1.8 aproximadamente con los datos de validación lo que quiere decir que, aunque haya casos que no ha visto seria capaz de acercarse a la solución de mopyhex en menos de 2 casillas.

Tras evaluar el modelo de distintas formas se ha llegado a la conclusión de que este modelo es capaz de imitar los movimientos al agente de mopyhex sin alejarse demasiado.

6.2 Versión 2

6.2.1 Entrenamiento con mopyhex

Ya que para el modelo anterior se ha observado que el mejor método para entrenarlo era automatizarlo con partidas simuladas de MOPYHEX no se ha intentado ningún otro método de entrenamiento.

Durante el entrenamiento de este modelo he podido comprobar que este modelo tiene un nivel de imitación muy parecido al anterior. Sin embargo, era mucho más capaz de realizar movimientos adecuados ya que los fallos los colocaba en posiciones adyacentes.

6.2.2 Auto juego

Tras haberlo entrenado se ha considerado que como método de evaluación se debería comprobar como juega consigo mismo. Para ello se han simulado una serie de partidas entre el modelo.

Se ha podido comprobar que este modelo es muy defensivo y siempre trata de poner cerca del último movimiento del rival.

6.2.3 Evaluación

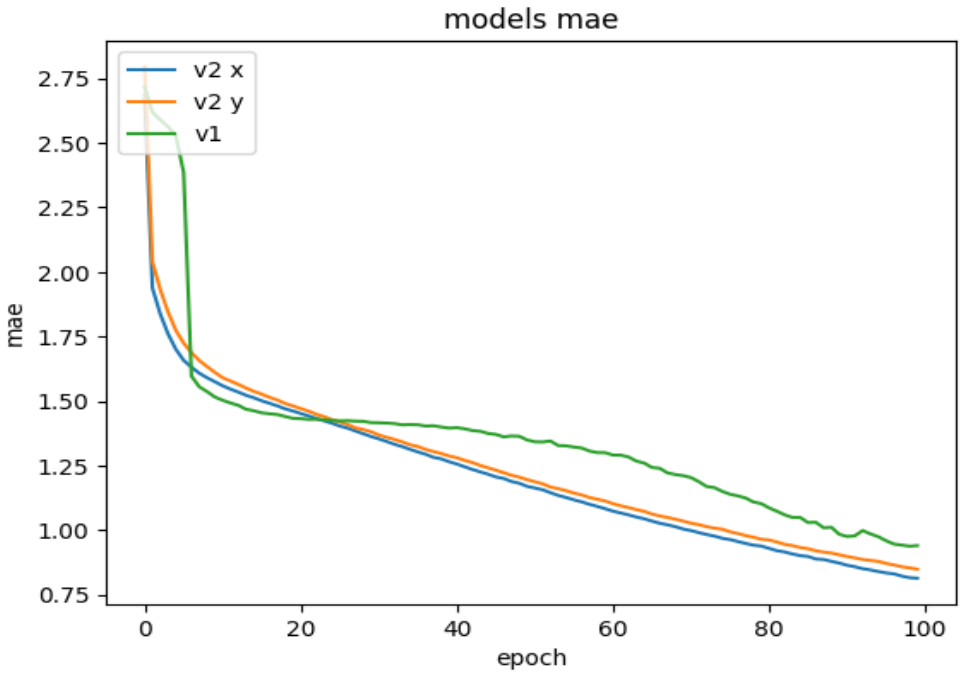
Durante esta evaluación también se ha hecho uso del MAE como métrica para decidir con que grado de aprendizaje esta imitando este modelo. Se ha utilizado el mismo dataset que con el modelo anterior y el mismo tipo de data augmentation con la intención de replicar las mismas condiciones y facilitar la comparación.

El resultado del MAE en este modelo ha sido de 0.5 aproximadamente con los datos de entrenamiento lo que quiere decir que sus resultados se desvían en menos de 1 casilla en las predicciones que conoce y de 1.75 aproximadamente con los datos de validación lo que quiere decir que, aunque haya casos que no ha visto sería capaz de acercarse a la solución de mopyhex en menos de 2 casillas.

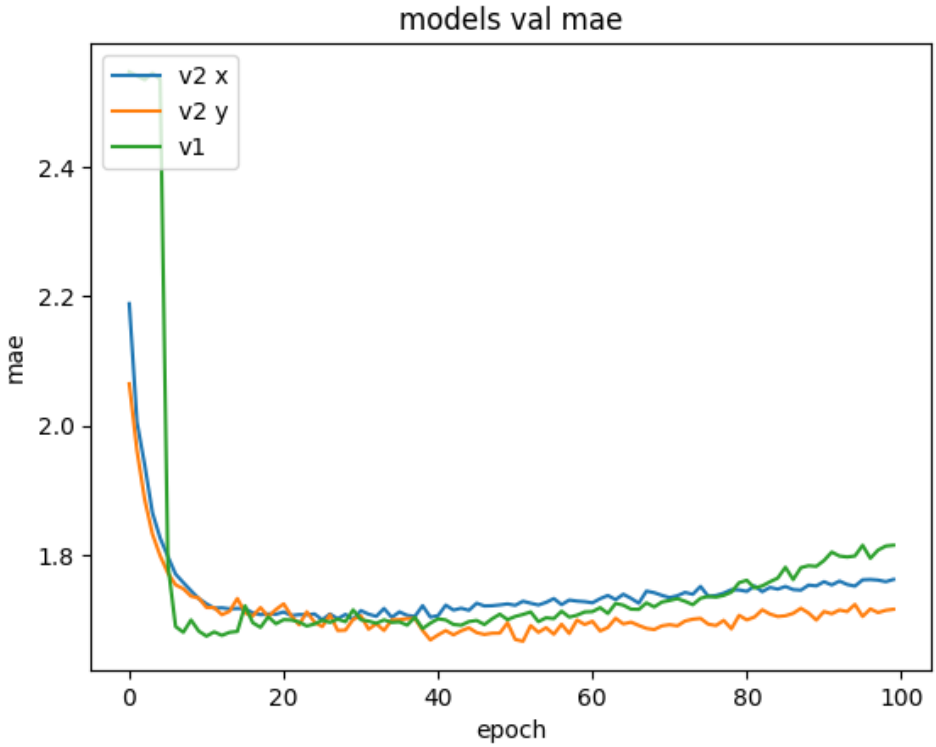
6.3 Comparación de versiones

Para esta comparativa se ha cogido como referencia el error absoluto medio.

Grafica que muestra el MAE de con los datos de entrenamiento a lo largo de las épocas en función de las dos versiones:



Grafica que muestra el MAE de con los datos de validación a lo largo de las épocas:



Tras observar los resultados del MAE de ambos modelos se ha llegado a la conclusión de que el modelo de la versión 2 es mejor dando como resultado un MAE inferior a 1 con los datos de entrenamiento y un MAE inferior a 1.8 con los datos de validación. Mientras que la versión 1 esta bastante cerca en sus resultados sin llegar a superarlo.

6.4 Comparación de modelos

Para la comparación y análisis de los distintos modelos que se han visto en este trabajo se ha evaluado el comportamiento en usa serie de partidas simuladas entre ellos en tableros de 8x8.

Los modelos evaluados han sido:

- Katahex: Con una arquitectura basada en KataGo combina redes convolucionales con los árboles de búsqueda de Montecarlo.
- Mopyhex: implementación en Python del método de árboles de búsqueda de Montecarlo.

- Neurohex: Con una arquitectura basada en AlphaGo con 10 capas convolucionales terminadas en una capa una neurona y combinado con los árboles de búsqueda de Montecarlo. Este modelo es capaz de ganar a mopyhex con la desventaja de empezar segundo e incluso permitirse cometer errores en las ultimas etapas de la partida.
- imitationagent ver. 1: modelo de redes de neuronas entrenado para imitar el método de árboles de búsqueda de Montecarlo.
- imitationagent ver. 2: modelo con dos redes de neuronas idénticas para seleccionar la fila y la columna por separado. Capaz de imitar la jugada del rival con la intención de bloquearle.
- agente humano: Todos estos modelos también han sido probados contra un jugador humano.

La tabla de resultados muestra el número de veces que el agente manejando las piezas blancas (empezando primero) ha ganado un encuentro:

blancas\negras	katahex	mopyhex	neurohex	imitationagent ver.1	imitationagent ver. 2	agente humano
katahex	6	10	7	10	9	10
mopyhex	0	8	0	7	6	10
neurohex	7	10	3	9	7	10
imitationagent ver. 1	0	4	0	10	5	7
imitationagent ver. 2	5	6	4	7	6	9
agente humano	2	1	2	6	3	10

Tras esta evaluación se ha llegado a la conclusión de que los árboles de búsqueda de Monte Carlo son la solución para hacer un agente para jugar a hex y que suponga un reto para jugadores profesionales. Especialmente si se combina con redes convolucionales. Sin embargo, tienen la desventaja de que son muy lentos ya que tarda 10 segundos en elegir un movimiento. Los agentes de imitación están mejor adaptados para jugar partidas rápidas y si están entrenados con partidas generadas con los arboles de Monte Carlo pueden ser un reto mas apto para partidas en las que el tiempo por turno sea limitado.

7 conclusión

Recordando la pregunta planteada en la introducción: ¿Se podría desarrollar una inteligencia artificial capaz de aprender de las partidas que tiene con el jugador? La respuesta que he hallado realizando este proyecto es que sí, sí se puede. Pero para hacerlo competente se necesita entrenarlo con una gran cantidad de jugadas lo que conlleva una gran cantidad de tiempo y recursos. Si realmente se quiere crear un algoritmo de imitación que sea capaz de ganar en los más altos niveles debe ser entrenado por sus propios rivales antes de enfrentarse a ellos. Para entrenarlo con un agente humano es necesario saber que la curva de aprendizaje de estos modelos es muy larga así que se recomienda más para gente que quiera aprender a jugar con un rival que se vuelva más difícil con cada partida porque su nivel de juego va a evolucionar con el jugador.

Se considera que, para futuros trabajos, alguien investigue más sobre los algoritmos de aprendizaje por imitación porque podrían tener mucho futuro. En cuanto a los algoritmos de Hex no se cree que los algoritmos de imitación puedan igualar en calidad de juego a los basados en Monte Carlo, pero se podría entrenar un modelo que pueda simular un algoritmo de Monte Carlo y eso seria incluso mejor ya que el algoritmo de Monte Carlo tarda demasiado en seleccionar una ficha y eso afecta también a modelos de Deep learning que lo usen como katahex o neurohex.

Se debe terminar este capítulo mencionando que el código de este proyecto esta publicado en GitHub, pero no se hará público hasta que hayan pasado las evaluaciones de los trabajos de fin de curso.

8 Referencias

8.1 Webgrafia

Código utilizado para realizar las pruebas:

https://github.com/migueltellogomez/tfg

HexWiki; (2023); Main Page

https://www.hexwiki.net/index.php/Main_Page

Autor desconocido, fecha desconocida, Imitation Learning; deepai.org;

https://deepai.org/machine-learning-glossary-and-terms/imitation-learning

Young.K; (2016); NeuroHex; github

https://github.com/kenjyoung/Neurohex

Anderson.B; fecha desconocida; HexGUI; github

https://github.com/selinger/hexgui

HZY; (2022); KataHex; github

https://github.com/selinger/katahex

Varios autores; (2022); imitation; github

https://github.com/HumanCompatibleAI/imitation

Anaconda; (2024); about us

https://www.anaconda.com/about-us

Keras; (2024); Keras

https://keras.io/

Varios autores; (2022); GTP

https://www.hexwiki.net/index.php/GTP

UNESCO; fecha desconocida; Ética de la inteligencia artificial

https://www.unesco.org/es/artificial-intelligence/recommendation-ethics#:~:text=La%20Recomendaci%C3%B3n%20interpreta%20la%20IA,similar%20a%20un%20comportamiento%20inteligente.

cs677; (2020); Imitation Learning; GitHub

https://pantelis.github.io/cs677/docs/common/lectures/autonomous-cars/imitation-learning/

8.2 Bibliografía

Varios autores, (2012),Reinforcement Learning; Springer Berlin, Heidelberg

https://link.springer.com/chapter/10.1007/978-3-642-27645-3_1

Richard E. Bellman; (1957); A Markovian Decision Process

https://apps.dtic.mil/sti/tr/pdf/AD0606367.pdf

Ronald A. Howard; (1960); Dynamic Programming and Markov Processes

https://gwern.net/doc/statistics/decision/1960-howard-dynamicprogrammingmarkovprocesses.pdf

Qiong Liu, Ying Wu; (2015); Supervised Learning

https://www.researchgate.net/publication/229031588_Supervised_Learning

Varios autores, (1997), Neural Systems for Control; Elsevier

https://www.sciencedirect.com/science/article/abs/pii/B9780125264303500039

Giacomo Bonanno; (2018); GAME THEORY

https://faculty.econ.ucdavis.edu/faculty/bonanno/PDF/GT_book.pdf

Nishanth Kumar; (2022); The Past and Present of Imitation Learning

A Citation Chain Study

https://arxiv.org/pdf/2001.02328.pdf

Varios autores; (2012); A Survey of Monte Carlo Tree Search Methods

https://www.lamsade.dauphine.fr/~cazenave/A+Survey+of+Monte+Carlo+Tree+Search+Methods.pdf

Cazebave.T; Saffidine.A, fecha desconocida, Monte-Carlo Hex; Francia

https://www.lamsade.dauphine.fr/~cazenave/papers/hex.pdf

Chalup. S; Mellor.D; Rosamond.F; (2005); The Machine Intelligence Hex Project; Australia

https://www.researchgate.net/publication/228909062_The_machine_intelligence_Hex_project

Young.K; Vasan.G; Hayward.R; (2016); NeuroHex: A Deep Q-learning Hex Agent; USA

https://webdocs.cs.ualberta.ca/~hayward/papers/neurohex-paper.pdf

Varios autores; (2022); imitation: Clean Imitation Learning Implementations;

https://arxiv.org/abs/2211.11972

Broderick Arneson, Ryan B. Hayward, Philip Henderson; (2010); Monte Carlo Tree Search in Hex

https://webdocs.cs.ualberta.ca/~hayward/papers/mcts-hex.pdf

Antonio Moreno, Eva Armengol, Javier Béjar, Lluís Belanche, Ulises Cortés, Ricard Gavaldà, Juan Manuel Gimeno, Beatriz López, Mario Martín, Miquel Sànchez; (1994); Aprendizaje automático

https://upcommons.upc.edu/bitstream/handle/2099.3/36157/9788483019962.pdf

Serguei Guerra Fernández, Yasser Lomaña Padilla, Omar Alexander Guzmán Obregón, Yudel Pérez Arbella; (2013); Optimización de la Estimación de DOA en Sistemas de Antenas Inteligentes usando criterios de Redes Neuronales

https://www.researchgate.net/publication/262746657_Optimizacion_de_la_Estimacion_de_DOA_en_Sistemas_de_Antenas_Inteligentes_usando_criterios_de_Redes_Neuronales

Gio Wiederhold, John McCarthy; (1992); Arthur Samuel: Pioneer in Machine Learning

https://www.researchgate.net/publication/224103556_Arthur_Samuel_Pioneer_in_Machine_Learning

Autor desconocido; (2019); Python/Generalidades/Breve historia de Python

https://es.wikibooks.org/wiki/Python/Generalidades/Breve_historia_de_Python

Autor desconocido; fecha desconocida; Strategies of play: Minimax

https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/Minimax.html#:~:text=The%20Minimax%20algorithm%20is%20the,choice%20that%20a%20player%20makes.

Jeffrey Barratt, Chuanbo Pan; fecha desconocida; Deep Imitation Learning for Playing Real Time Strategy Games

https://cs229.stanford.edu/proj2017/final-reports/5244338.pdf

Brian D. Ziebart, J. Andrew Bagnell, Anind K. Dey; fecha desconocida; Modeling Interaction via the Principle of Maximum Causal Entropy

https://www.cs.cmu.edu/~bziebart/publications/maximum-causal-entropy.pdf

9 Índice de figuras

10 Anexo

1. Código que representa el agente de imitación:

```
from gamestate import gamestate
from copy import deepcopy
import keras
from keras.layers import Dense
import pandas as pd
import random



class imitationagent:
    """
   implementacion de un agente que realiza imitation learning para hex.
   """

    def __init__(self, state=gamestate(8)):

        self.model = keras.Sequential()
        self.size = 0
        self.set_gamestate(state)

        "se crea la red de neuronas que se va a utilizar con tantas neuronas en la capa
de entrada como grande sea el tablero"
        self.model.add(keras.Input(shape=(self.dim,)))
        self.model.add(Dense(self.dim, activation='relu'))
        self.model.add(Dense(64, activation='relu'))

        self.model.add(Dense(64, activation='relu'))

        self.model.add(Dense(64, activation='relu'))

        self.model.add(Dense(64, activation='relu'))

        self.model.add(Dense(16, activation='relu'))
        self.model.add(Dense(8, activation='relu'))
        "queremos que la salida sean dos (fila y columna) y con funcion de activacion
relu para sacar numeros enteros enteros positivos"
        self.model.add(Dense(2, activation='relu'))
        self.model.compile(loss='mse', optimizer='adam', metrics = ['accuracy'])
        "cargamos pesos anteriores"
        self.model.load_weights('model' + str( self.dim))

)
        print(self.model.summary())

    def best_move(self):
        """
      predice un movimiento usando el modelo y lo devuelve.
      """
        if (self.rootstate.winner() != gamestate.PLAYERS["none"]):
```

```
            "cuando termina el juego guarda los pesos de las neuronas"
            self.model.save_weights('model' + str( self.dim))
            return gamestate.GAMEOVER


        board = [self.anterior.board.flatten().astype(int).tolist()]
        "selecciona el movimiento"
        bestmove = self.model.predict(board)
        "nos aseguramos que los resultados esten en los valores aceptables"
        resul = (bestmove[0][0].astype(int) % self.size, bestmove[0][1].astype(int) %
self.size)
        if(resul in self.rootstate.moves()):
            return resul
        else:
            """para evitar que seleccione una casilla elegida cada vez que lo haga le
            penalizamos seleccionando una casilla al azar.
            este movimiento al azar tambien formara parte de su entrenamiento
            para que aprenda a evitar casillas incorrectas """
            print("casilla ya seleccionada")
            print("seleccionando casilla aleatoria de las disponibles")
            resul = random.choice(self.rootstate.moves())
            return resul


    def move(self, move):
        """
      mover ficha elegida y entrenamiento
      """

        trainX = self.anterior.board.flatten().astype(int).tolist()
        trainy = move
        self.databaseX = pd.DataFrame([trainX])
        self.databaseY = pd.DataFrame([trainy])

        "se realiza data augmentation"
        for i in range(100):
            X_train = pd.concat([self.databaseX, pd.DataFrame([trainX])], ignore_index =
True)
            y_train = pd.concat([self.databaseY, pd.DataFrame([trainy])], ignore_index =
True)


        "se entrena con cada movimiento"
        self.model.fit(X_train, y_train, epochs=100, verbose=0)

        self.rootstate.play(move)

    def set_gamestate(self, state):
        """
      Actualiza el estado del juego
      """
        self.rootstate = deepcopy(state)
        self.anterior = state
        self.size = state.size
        self.dim = state.size * state.size
```

1. Grafica que muestra las diferencias en el MAE entre redes con distinta configuración de neuronas de más simple a más compleja: