Universitat de les Illes Balears

TRABAJO DE FIN DE MÁSTER

MODELOS DE INTELIGENCIA ARTIFICIAL APLICADOS AL JUEGO “HUNGRY GEESE”

Juan José Martín Miralles

Máster Universitario en Análisis de Datos Masivos en Economía y Empresa

Especialidad de Herramientas de gestión y análisis inteligente de datos

Centro de Estudios de Postgrado

Año Académico 2020 - 2021

MODELOS DE INTELIGENCIA ARTIFICIAL APLICADOS AL JUEGO “HUNGRY GEESE”

Juan José Martín Miralles

Trabajo de Fin de Máster

Centro de Estudios de Postgrado

Universidad de las Illes Balears

Año Académico 2020 - 2021

Palabras clave del trabajo:

Hungry Geese, Monte Carlo Tree Search, Deep Learning, Convolutional Neural Networks, Reinforcement Learning, Actor-Critic Learning, Fictitious Self-play

Tutor:

Dr. Gabriel Moyà Alcover

Modelos de Inteligencia Artificial aplicados al juego "Hungry Geese"

Juan José Martín Miralles
Tutor: Dr. Gabriel Moyà Alcover
Trabajo de fin de Máster Universitario en Análisis de Datos Masivos en Economía y Empresa (MADM)
Universitat de les Illes Balears
Palma, Mallorca

Año Académico 2020 - 2021

Resumen

El objetivo de este trabajo de fin de máster es el análisis y la comparación de diferentes modelos de inteligencia artificial aplicados al juego "Hungry Geese", una versión del mítico juego "Snake", pero con gansos, de manera que múltiples agentes compiten entre sí con el objetivo de sobrevivir más tiempo. Se han desarrollado varios agentes con diversas metodologías con el fin de explicarlos y analizarlos. Los agentes implementados son un modelo determinista, un modelo basado en *Monte Carlo Tree Search (MCTS)*, otro creado con redes neuronales convolucionales mediante un entrenamiento supervisado y, por último, un agente entrenado mediante la técnica de *Reinforcement Learning* con *actor-critic learning*, cuyos resultados no han sido satisfactorios. Finalmente, se exponen los resultados obtenidos por cada uno de los agentes implementados con el fin de detectar cuál es la técnica que permite obtener una mejor puntuación en el juego.

Abstract

This master's thesis aims to analyze and compare different artificial intelligence models applied to the game "Hungry Geese", a version of the mythical game "Snake", but with geese, in such a way that multiple agents compete with each other in order to survive longer. Various agents have been developed with various techniques in order to explain and analyze them. The implemented agents are a deterministic model, a model based on *Monte Carlo Tree Search (MCTS)*, another created with convolutional neural networks through supervised training and an agent trained using the technique of *Reinforcement Learning* with *actor-critic learning*, whose results have not been satisfactory. Finally, the results obtained by each of the implemented agents are exposed in order to detect which is the technique that allows obtaining a better score in the game.

Palabras clave: Hungry Geese, Monte Carlo Tree Search, Deep Learning, Convolutional Neural Networks, Reinforcement Learning, Actor-Critic Learning, Fictitious Self-play

1. Introducción

En los últimos años, la investigación en los campos del aprendizaje automático y la inteligencia artificial han crecido de manera drástica. Muchas empresas dedican gran parte de su capital en realizar proyectos dentro de estas áreas. Por ejemplo, en 2014, la empresa DeepMind, dedicada a la inteligencia artificial, fue adquirida por Google después de desarrollar una inteligencia artificial capaz de jugar a los videojuegos de Atari como lo haría un humano [4]. Esta misma empresa desarrolló AlphaGo, una inteligencia artificial para jugar al juego de Go que, en 2015, se convirtió en la primera máquina en ganar a un jugador profesional de Go en un tablero de 19x19 [25].

El objetivo de este TFM es analizar y comparar distintas metodologías para crear una inteligencia artificial que sea capaz de jugar a Hungry Geese. Los modelos y técnicas implementados en este proyecto sirven como base inicial para la aplicación en otros juegos que requieran inteligencia artificial, reconocimiento facial, segmentación de clientes en marketing, modelos de finanzas, etc. Se ha desarrollado un agente determinista, otro basado en *Monte Carlo Tree Search (MCTS)* y otro mediante redes neuronales convolucionales con entrenamiento supervisado. Por último, con el fin de mejorar este último agente, se ha llevado a cabo un análisis e implementación de un modelo basado en *Reinforcement Learning* mediante *actor-critic learning* [26, 23, 17], una clase de algoritmo de políticas de gradiente (*policy gradient algorithms*) que está compuesto por dos componentes principales: una política y una función de valor estimado. El método de *reinforcement learning* no ha dado los resultados esperados en la fase de entrenamiento y, por lo tanto, no se ha podido probar en partidas reales de Hungry Geese.

Hungry Geese es una versión del mítico juego "Snake", pero con gansos, de manera que múltiples agentes compiten entre sí con el objetivo de sobrevivir más tiempo. Este juego nace en una competición de *Kaggle* [10], desarrollada entre el 26 de enero de 2021 y el 10 de agosto de 2021. En este juego, los gansos no deben morir de hambre ni impactar contra los oponentes ni contra sí mismos. Una de las diferencias con el juego de "Snake" es que no hay límites por los bordes del tablero, por lo que cuando el ganso sale por uno de los 4

laterales, aparece por el contrario.

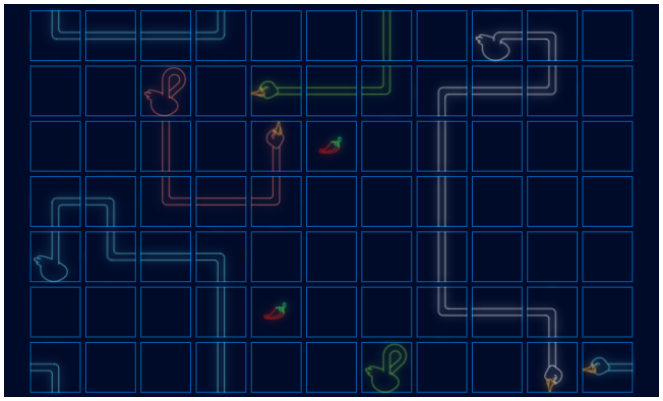


Figura 1: Ejemplo del juego Hungry Geese.

La figura 2, creada por uno de los mejores participantes en la competición [21], muestra la distribución de puntuaciones de los agentes publicados. Las líneas verticales son los rangos de puntuación de los agentes que han conseguido la medalla de bronce, plata u oro en la competición. Los agentes que han conseguido una puntuación de entre 1050 y 1080, aproximadamente, han conseguido la medalla de bronce; entre 1080 y 1210, han conseguido la medalla de plata; a partir de 1210, la medalla de oro. El agente que ha conseguido el primer puesto ha obtenido una puntuación de 1312.

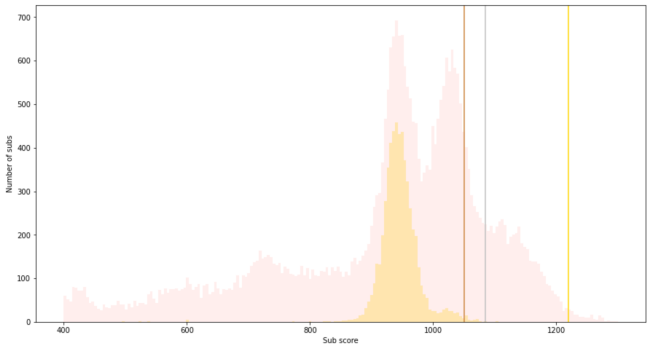


Figura 2: Distribución de puntuaciones de los agentes publicados.

La figura 2 muestra dos picos máximos bien diferenciados: aproximadamente, entre los 900 y 1000 puntos y entre los 1000 y 1100 puntos, respectivamente. En [21], se ha analizado que el 17.5 % de los agentes publicados que están en el primer pico son copias de un agente basado en redes neuronales que se hizo muy popular al ser uno de los primeros publicados, llamado *PubHRL* [30]. Estos agentes se muestran en la figura en color amarillo y algunos de ellos han entrado en bronce realizando pequeñas mejoras. Sobre el segundo pico, la mayoría de los agentes involucrados corresponden a copias de otro agente popular publicado en la mitad de la competición, llamado *AlphaGeese* [24]. Este agente usa como base *PubHRL* añadiendo un algoritmo de *Monte Carlo Tree Search*, de forma similar a como se hace en *AlphaGo* [25].

En la sección de redes neuronales convolucionales con entrenamiento supervisado se ha hecho un estudio entrenando una red neuronal con los movimientos realizados en partidas reales por agentes que se encuentran en diferentes rangos de puntuación. Las partidas se han obtenido mediante *scraping* al conjunto de metadatos de *Kaggle* [10].

1.1. Reglas del juego

El objetivo del juego es sobrevivir el mayor número de turnos alimentándose para mantenerse vivo y sin colisionar con tu propio ganso o el de otros agentes. La reglas del juego son las siguientes:

- Cuatro agentes compiten en un tablero de 11x7 celdas indicando en cada turno una de las siguientes acciones: **NORTH**, **SOUTH**, **EAST** o **WEST**.
- No está permitido hacer giros de 180º, es decir, volver a la casilla que se ocupaba en el turno anterior.
- Cada partida consta de 200 episodios. El agente con la valoración más alta o el último que sobreviva gana la partida.
- Si un agente se come una pieza de comida, se le añade un segmento a su cuerpo.
- Cada 40 episodios, el ganso pierde un segmento, independientemente de si ha comido o no con anterioridad.
- Los agentes deben tomar una decisión de movimiento en menos de 1 segundo. Si se sobrepasa, el tiempo de diferencia se añade a un contador. Si este contador llega a 60 segundos, el agente es eliminado de la partida.

Los detalles sobre la información recibida por los agentes se puede encontrar en [10], donde están explicadas las descripciones, el tipo de dato, el valor por defecto y los elementos que puede contener cada variable.

1.2. Objetivo del estudio

El objetivo del estudio es analizar y comparar distintos modelos de inteligencia artificial que puedan ser mejores que ciertos algoritmos tradicionales o deterministas para jugar a Hungry Geese.

1.3. Estructura del documento

Este documento se estructura en 4 secciones. En la primera de ellas, se expone la metodología empleada para llevar a cabo el proyecto donde se explican los diferentes modelos implementados. La segunda sección contiene los experimentos realizados sobre los agentes. En la tercera, se analizan los resultados de los experimentos para explicar los comportamientos de los distintos agentes. Por último, se exponen las conclusiones finales del TFM.

2. Metodología

Se han analizado varios agentes con diversas metodologías con el fin de compararlos y ver cuál es mejor. Los agentes que se han comparado son:

- **Agente aleatorio**: Es un agente cuyos movimientos son totalmente aleatorios, sin ningún tipo de lógica y sin comprobar si existen colisiones con segmentos ocupados por otro agente o por sí mismo. Únicamente se le impide a este agente volver a la casilla que ocupaba en el anterior turno.
- **Agente determinista**: Este agente se mueve en base a acciones predefinidas durante su desarrollo en base al conocimiento que se ha adquirido observando las técnicas que aplican otros agentes.
- **Agente basado en *Monte Carlo Tree Search***: Un agente basado en MCTS permite crear una forma de evaluar el estado del juego sin la necesidad de tener ningún conocimiento previo. MCTS pretende simular movimientos aleatorios para estimar lo buena o mala que puede ser una acción [3]. En la implementación realizada para Hungry Geese no se usa el método *Upper Confidence bounds applied to Trees (UCT)*, introducido en [11], y usado en [22], [18] y [2], sino que el método de selección del nodo del árbol es aleatorio entre los posibles movimientos *válidos* que puede realizar el agente.
- **Redes neuronales convolucionales con entrenamiento supervisado**: Aplicar redes neuronales convolucionales es el siguiente paso para poder crear una inteligencia artificial que sea capaz de competir mejor que el agente basado en MCTS. Para este caso, se ha utilizado un entrenamiento supervisado obteniendo los conjuntos de datos a partir de partidas simuladas mediante el algoritmo MCTS implementado y de partidas reales extraídas mediante *scraping*.
- ***Reinforcement Learning* mediante *actor-critic learning***: Con el objetivo de mejorar el agente basado en redes neuronales con entrenamiento supervisado, se ha llevado a cabo un análisis e implementación de un modelo basado en *Reinforcement Learning* mediante *actor-critic learning* [26, 23, 17]. La metodología de entrenamiento usado es *self-play* [7, 8], donde un mismo agente juega partidas contra copias de sí mismo, obteniendo experiencia de sus movimientos dado los estados del tablero. Este modelo se explica en la sección 5 debido a que se deja fuera de los experimentos y del análisis de los resultados por no haber dado el resultado esperado en la fase de entrenamiento y no haberlo podido probar sobre partidas reales de Hungry Geese.

2.1. Agente aleatorio

El agente más básico posible que se puede implementar es aquel cuyos movimientos son totalmente aleatorios, sin ningún tipo de lógica y sin comprobar si existen colisiones con segmentos ocupados por otro agente o por sí mismo. Además, cualquier pieza de comida que se coma es por mera casualidad. Lo único que se tiene en cuenta es que no vuelva a la celda que ocupaba en el paso anterior debido a que es una norma del juego. Crear un agente con estas características sirve para construir otros agentes más inteligentes como, por ejemplo, uno basado en *Monte Carlo Tree Search* cuyos movimientos son seleccionados aleatoriamente hasta llegar a una cierta profundidad para escoger la mejor acción.

2.2. Agente determinista

Este agente se mueve en base a acciones predefinidas. Normalmente, hacer un agente con estas características sirve para adquirir conocimiento sobre el funcionamiento del juego y poner una línea roja sobre la cual todos los demás agentes que se creen deberán ser mejores.

En este caso, dado que uno de los principios básicos del juego es sobrevivir el mayor tiempo posible, se comprueba en todo momento que el siguiente movimiento que se realice sea válido y no colisione. Además, para no construir un árbol de búsqueda en profundidad, lo cual sería útil para poder seleccionar el mejor camino a tomar pero tendría un coste computacional demasiado alto por la cantidad de posibilidades que hay, solo se comprueba cuál es el camino con más profundidad que el ganso podría coger, limitado a una profundidad de la mitad de la longitud del ganso. Esto disminuye las posibilidades de que entre en un camino sin salida. Además, el ganso solo va a por una pieza de comida si es el agente que está más cerca de dicha pieza. En caso de no ser el que está más cerca, la ignora y simplemente se mueve a una posición válida.

Además, otro aspecto a tener en cuenta son las casillas a las que nuestro agente puede acceder pero que tienen la cabeza de un contrincante adyacente a ella, como se muestra en la figura 3. En estos casos, el agente determinista ignora la casilla debido a que hay un alto porcentaje de que acaben chocando, ya que no se conoce cómo está implementado el otro agente. Podría ser que también fuera un agente determinista y se dirigiera a la comida sin importar si colisiona, o que fuera un agente entrenado con modelos de inteligencia artificial y detecte que es mejor correr el riesgo de ir a por la comida, o que, simplemente, no haya entrenado lo suficiente como para detectar estos casos.

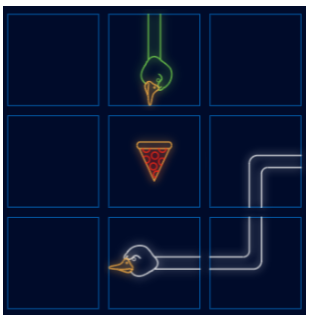


Figura 3: Casillas con comida y 2 cabezas adyacentes.

Un recurso que han utilizado otros usuarios que han creado agentes para Hungry Geese es el de envolverse sobre sí mismo, es decir, que la cabeza de un agente persiga su propia cola en casillas adyacentes. Esta técnica sirve para protegerse en el caso de estar rodeado por varios agentes más grandes, como se muestra en la figura 4 para el agente de color verde.

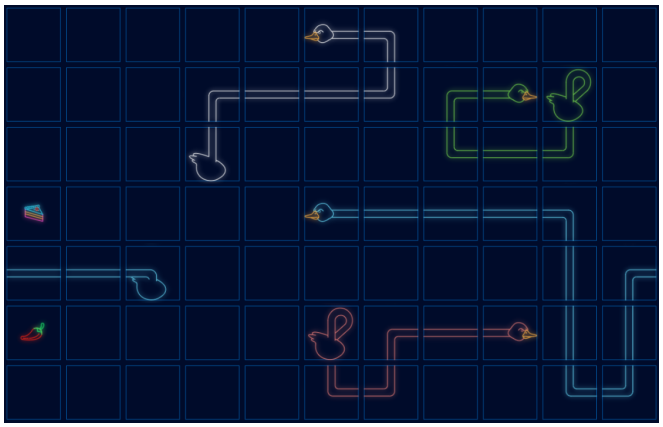


Figura 4: Agente de color verde – Recurso de protección donde la cabeza persigue a la cola.

2.3. Agente basado en *Monte Carlo Tree Search*

Así como para crear un agente determinista que sea competitivo se necesita tener un alto conocimiento del juego y tener en cuenta múltiples casuísticas, un agente basado en *Monte Carlo Tree Search* permite crear una forma de evaluar el estado del juego sin la necesidad de tener ningún conocimiento previo. Este método pretende estimar lo buena o mala que es una acción mediante el resultado de múltiples simulaciones aleatorias. MCTS apareció en artículos científicos en 2006 [11, 3] y dio muy buenos resultados en el juego de Go [2].

MCTS fue usado en [22] para un juego similar a Hungry Geese, llamado **Tron**, donde dos agentes, de manera simultánea, se mueven por un tablero de *M x N* en cuatro direcciones: norte, sur, este y oeste. Los autores aplicaron el método de *Upper Confidence bounds applied to Trees (UCT)* para seleccionar el nodo del árbol que se debe comprobar en la siguiente simulación. En [18] se comparan distintas estrategias de selección en MCTS para el juego de Tron, siendo una política de selección determinista aplicada a UCT la que mejor resultado da. No obstante, la ventaja del juego de Tron con respecto a Hungry Geese es que únicamente juegan dos jugadores y solo deben tener en cuenta que no tiene que colisionar entre sí o contra las paredes, por lo que no existe la necesidad de tener que comer para ganar o sobrevivir. Es decir, el árbol de búsqueda no es tan grande y es posible ir a una mayor profundidad o hacerlo más amplio, sobre todo si no existe un tiempo máximo de movimiento, como sí hay en Hungry Geese. Por este motivo, para el juego de Hungry Geese, la fase de selección del nodo por el que empezar una nueva simulación se hace de manera aleatoria uniforme entre los posibles movimientos *válidos* que puede realizar el agente. De esta forma, no se utiliza tiempo para buscar el mejor nodo, sino en realizar más simulaciones o ir a una mayor profundidad.

Monte Carlo Tree Search se divide en cuatro pasos principales para cada iteración hasta que el juego finaliza [5]:

1. **Selección:** este paso consiste en seleccionar un nodo en el árbol desde el cual comenzar una nueva simulación. En [18] se comparan empíricamente las políticas de selección deterministas (*UCB1*, *UCB1-Tuned*, *UCB-V*, *UCB-Minimal*, *OMC-Deterministic* y *MOSS*) y estocásticas (e_n*-greedy*, *EXP3*, *Thompson Sampling*, *OMC-Stochastic* y *PBBM*). No obstante, tal y como se ha explicado anteriormente, la metodología elegida para seleccionar un nodo del árbol es totalmente aleatoria.
2. **Expansión:** si el nodo seleccionado no es el final de la partida, se añade un nuevo nodo a este y se selecciona este nuevo nodo. Es decir, se escoge el primer movimiento que realiza el agente desde el nodo seleccionado.
3. **Simulación:** este paso consiste en simular una partida desde el nodo seleccionado, realizando movimientos aleatorios, con el método *self-play* [7], hasta el final del juego para devolver la recompensa que ha obtenido el agente en esa simulación.
4. **Propagación:** este paso consiste en propagar el resultado de la simulación (la recompensa obtenida) desde el nuevo nodo creado hasta la raíz del árbol.

Este agente selecciona aleatoriamente una de las acciones que el ganso puede realizar, es decir, escoge el primer movimiento del recorrido que va a realizar. A partir de aquí, se simulan movimientos aleatorios tanto de nuestro agente como de los rivales hasta una profundidad especificada o hasta que la partida llegue a su fin. Esto da una puntuación a nuestro agente que, en ocasiones morirá, otras cogerá comida y otras, simplemente, avanzará una casilla sin colisionar. Para conocer cada una de estas posibilidades, es necesario informar al agente sobre cuál es la recompensa final que obtendrá dada la acción inicial. La función para calcular la recompensa es la siguiente:

$$r = \begin{cases} 0 & \text{if } agent\_length = 0 \\ (step + 1) + agent\_length^5 & \text{otherwise} \end{cases}$$

donde *step* es el número de pasos que ha realizado el agente desde que inició la primera acción hasta que acaba la simulación de ese recorrido y *agent_length* es la longitud que tiene el agente cuando acaba la simulación de ese recorrido. De esta forma, si el agente muere en la simulación de una acción inicial, la recompensa para esa acción será 0. Por el contrario, si el agente avanza sin aumentar su longitud tendrá menos recompensa que si la aumenta, es decir, se le da más prioridad a las acciones que conlleven coger comida.

Estas simulaciones se realizan un número finito de veces y con una profundidad predefinida para cada recorrido. Finalmente, se hace una media de las recompensas de cada recorrido realizado dada la acción inicial seleccionada por el agente,

quedándose con aquella que más puntuación haya obtenido. No obstante, el problema de este método es que puede generar descompensación sobre el número de veces que simula el juego en una acción u otra, por lo que la media de puntuaciones se verá afectada. Además, se puede llegar a simular múltiples veces el mismo camino, ya que no se guardan las trazas de lo que se ha analizado con anterioridad. Otro problema que genera esta función es que no tiene en cuenta el camino más óptimo para llegar a coger un alimento, por lo que es posible que se detecten múltiples rutas por las que llegar a una misma casilla y que puedan ser buenas posibilidades en el paso actual pero malas en el siguiente paso.

2.4. Redes neuronales convolucionales con entrenamiento supervisado

Una red neuronal convolucional es una red neuronal formada por una o más capas convolucionales y se usan, principalmente, para procesamiento de imágenes, clasificación, segmentación y para otros tipos de datos correlacionados.

Al ser una red neuronal con entrenamiento supervisado y, por lo tanto, no estar implementada con *reinforcement learning*, es necesario obtener conjuntos de datos de entrenamiento para tener una etiqueta con la que pueda aprender qué acción realizar para cada estado del tablero. Por lo tanto, en la primera parte de esta sección, se explica cómo se han obtenido dichos conjuntos de datos. A continuación, se describe cómo es la entrada de los datos a la red neuronal. Por último, se explica la arquitectura implementada y el optimizador del modelo.

Obtención del conjuntos de datos de entrenamiento

Los modelos de inteligencia artificial supervisados requieren de un conjunto de datos de entrenamiento que contenga una "etiqueta" que indique cuál debería ser el resultado dadas unas variables. En el caso del juego Hungry Geese, estas variables son el estado actual del tablero, es decir, las posiciones ocupadas por los cuerpos de cada agente, las posiciones ocupadas por comida o cualquier otro dato que dé información acerca del tablero. La etiqueta es la acción (*NORTH*, *SOUTH*, *EAST* o *WEST*) que se debería realizar dado un estado del tablero.

Hungry Geese es un juego en el que todos los jugadores deben moverse a la vez, a diferencia de otros juegos donde cada jugador tiene su turno para decidir un movimiento. Esto provoca que cada jugador deba decidir al mismo tiempo cuál es la mejor acción que ha de realizar sin saber cuál será la acción que el contrario realizará al mismo momento. Esto da la posibilidad de que en un mismo turno se puedan guardar hasta 4 estados del tablero con su respectiva etiqueta, es decir, uno por cada jugador, por lo que se pueden crear conjuntos de datos más grandes en menos partidas y, por lo tanto, en menos tiempo. En las dos formas que se han implementado para obtener un conjunto de datos de entrenamiento, se ha guardado el estado actual del tablero y el movimiento que ha realizado cada agente para que, en el momento de codificar la entrada para la red neuronal, haya cuatro visiones y cuatro decisiones distintas para un mismo estado, es decir, una por cada agente.

Las dos soluciones que se han implementado para obtener conjuntos de datos son las siguientes:

- **Simular partidas completas mediante la implementación realizada con *Monte Carlo Tree Search*.** Mediante la simulación de partidas con el método de MCTS es posible obtener un gran conjunto de datos de entrenamiento pudiendo especificar el número de simulaciones que se deben hacer y la profundidad a la que se debe llegar. En este contexto ya no existe el inconveniente de que se debe dar una solución en menos de 1 segundo ya que no se realizan partidas reales, sino simuladas. Por lo tanto, si se quiere una mayor precisión sobre la acción emitida por MCTS, se deben aumentar el número de simulaciones a realizar y la profundidad.
- **Extracción de partidas de los mejores agentes del juego.** Mediante *scraping* al conjunto de metadatos de *Kaggle*, se han obtenido las partidas de los agentes publicados en Hungry Geese para crear un conjunto de datos con los estados del tablero que han tenido en cada turno y la acción que han decidido realizar. Mediante esta técnica, se pueden obtener conjuntos de datos más fiables descargando las partidas de los mejores agentes (los de puntuación superior a 1200, los cuales han entrado en posiciones para obtener el oro) debido a que se presupone que sus acciones son las más correctas. No obstante, no son agentes perfectos y también pierden partidas o realizan jugadas que no son óptimas, por lo que en el conjunto de datos también se encuentran movimientos con acciones que no son 100 % válidas y que el agente aprenderá de manera errónea.

Entrada y salida de la red neuronal

Los datos de entrada a la red neuronal contienen el estado actual del tablero, el cual se ha codificado en una matriz binaria de tamaño $(11, 7, 17)$, es decir, 11 columnas, 7 filas y 17 canales. Esto quiere decir que cada celda del tablero de tamaño 11 x 7 está representada por 17 variables binarias. Cada canal de la matriz codifica una parte distinta de información del tablero.

El valor en la matriz, para todos sus canales, es 1 si la posición en el tablero se corresponde a la celda de la matriz que se quiere representar en el canal. El valor es 0 si no se quiere representar nada en esa posición de la matriz. Si un agente está muerto, los canales que representen la posición actual en el tablero tendrán todas sus celdas con el valor 0 (Figura 5).

- Los primeros cuatro canales codifican las posiciones de las cabezas de cada agente, con un canal por agente. Cada canal es *one-hot* ya que hay, como mucho, un valor que no es 0.
- Los canales del 5 al 8 indican la posición de la cola de cada agente. Cada canal es *one-hot*.

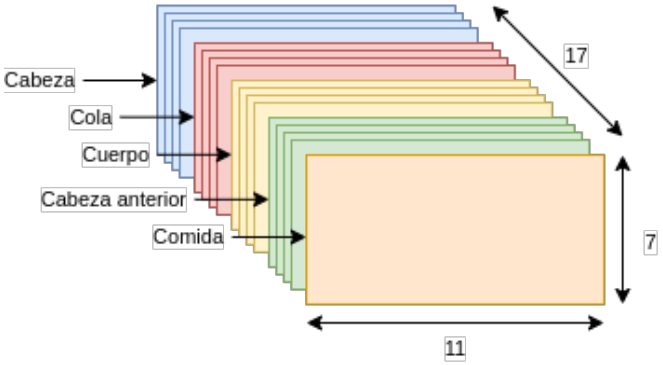


Figura 5: Diagrama de entrada de la red neuronal.

- Los canales del 9 al 12 codifican las posiciones de los cuerpos enteros de cada agente. Estos canales no son *one-hot*.
- Los canales de 13 al 16 indican las posiciones que ocupaban las cabezas de los agentes en el anterior estado del tablero. Cada celda es *one-hot*.
- El canal 17 indica todas las posiciones que ocupa la comida, pudiendo haber una o dos piezas de comida a la vez.

Con este tipo de codificación, el modelo puede inferir la longitud y forma del agente desde la posición de la cabeza, cuerpo y cola. Además, la dirección que ha tomado con respecto a la última posición de la cabeza y la posición de los demás agentes para evitar colisiones.

La salida de la red neuronal es una matriz de $1x4$ con las probabilidades de realizar cada una de las 4 acciones posibles que tiene el juego (*NORTH*, *SOUTH*, *EAST* o *WEST*). La acción con la probabilidad más alta es la que finalmente se realiza dado el estado actual del tablero en cada turno. Para la fase de entrenamiento, se utiliza una matriz $1x4$ *one-hot* con la acción realizada durante la obtención del conjunto de datos dado un estado del tablero.

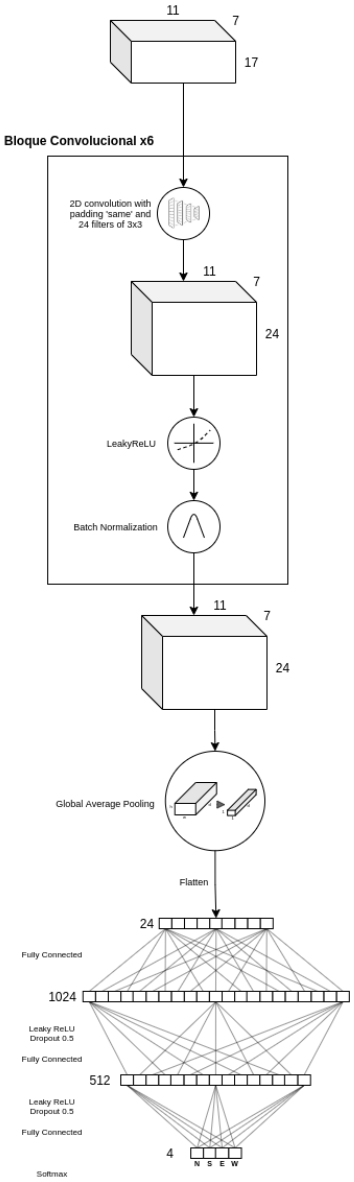


Figura 6: Arquitectura de la Red Neuronal.

Arquitectura de la red neuronal

La red neuronal implementada consiste en un primer bloque, llamado *bloque convolucional*, que se repite 6 veces, seguido de una capa *Global Average Pooling*, para minimizar la posibilidad de *overfitting* reduciendo el número de parámetros en el modelo, y finalizando con dos capas de neuronas completamente conectadas de 1024 y 512 neuronas, respectivamente, cuya función de activación es *Leaky ReLU*. La capa de salida consta de 4 neuronas, que son las cuatro posibles acciones que puede realizar el agente. La arquitectura completa de la red neuronal se puede ver en el figura 6.

El *bloque convolucional* consiste en una capa convolucional de dos dimensiones, con un *padding* automático que permite que la entrada y salida de esta capa sean iguales en altura y anchura, y 24 filtros de 3x3. Como se ha explicado anteriormente, el tamaño de la entrada del modelo es $(11, 7, 17)$, por lo que al aplicar *padding* automático y 24 filtros de 3x3, la salida de la capa convolucional será de $(11, 7, 24)$. A continuación, se aplica la función de activación *Leaky ReLU*. Por último, se usa una capa *Batch Normalization* para estandarizar la salida de la capa convolucional hacia la entrada de la siguiente capa, transformando los valores a media 0 y desviación estándar 1. *Batch Normalization* permite usar un *learning rate* más alto sin que haya un riesgo de divergencia, además de regularizar el modelo y reducir la necesidad de usar *Dropout* [27, 9]. Este mismo bloque se repite 6 veces más, obteniendo un salida final de tamaño $(11, 7, 24)$.

Global Average Pooling se aplica para reducir la dimensionalidad espacial de *(h, w, d)* a *(1, 1, d)*. El principio básico que sigue esta capa es la de reducir cada parámetro *(h, w)* a un único valor aplicando una media de todos los valores de *hw*.

El primer artículo donde se propuso esta capa es [13], donde aplican una capa de *Max Pooling* seguida de una capa *GAP* para producir un vector con una sola entrada para cada objeto posible en el proceso de clasificación y, finalmente, una capa densamente conexa.

Finalmente, se aplican dos capas ocultas de neuronas completamente conexas, de 1024 y 512 neuronas, respectivamente, con una función de activación *Leaky ReLU* y un *Dropout* de 0.5.

Optimizador del modelo

El optimizador usado en el modelo es *Stochastic Gradient Descent con Momentum (SGDM)*. Este optimizador es una extensión de *SGD* y ha sido ampliamente usado en modelos de inteligencia artificial con redes neuronales, obteniendo buenos resultados empíricos. La idea detrás de SGDM se origina en [19] con el método llamado *heavy-ball* de Polyak donde, intuitivamente, una pelota más pesada rebota menos en las paredes y se mueve más rápido en regiones con baja curvatura, en comparación a una pelota menos pesada, debido al impulso. De esta forma, los cambios de dirección son más pequeños y nos acerca más rápidamente a la convergencia. En [15] y [16] se analiza el comportamiento de SGDM en mínimos cuadrados de regresiones y se establece una convergencia lineal. En [14] se demuestra como SGDM converge tan rápido como SGD para entornos fuertemente convexos y no convexos.

Los parámetros que se deben ajustar en SGDM son *learning rate* y *momentum*. En los modelos de redes neuronales, SGDM se aplica con distintas metodologías que ajustan estos parámetros y ayudan a mejorar la eficiencia del entrenamiento. Una de las más populares es la aplicada en [6], donde se empieza el entrenamiento con un *learning rate* fijo que decrece por un factor constante cuando no hay una mejora en el modelo en un número determinado de *epochs*, mientras que el parámetro *momentum* se mantiene constante durante todo el entrenamiento a 0.9.

Se ha usado este mismo método para entrenar la red neuronal de la figura 6. El *learning rate* inicial es de 0.01 y decrementa por un factor de 0.3 cuando el modelo no mejora en su conjunto de datos de validación durante 5 *epochs* hasta un mínimo de 0.0001.

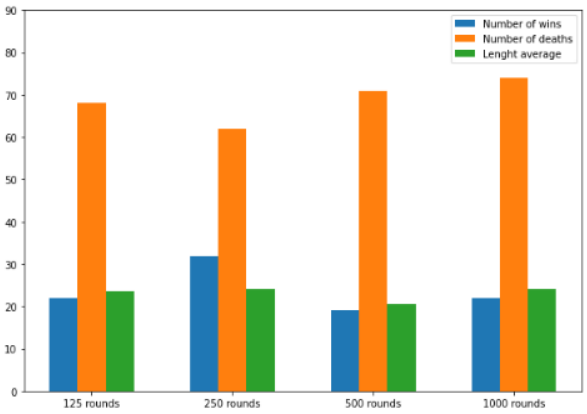
3. Experimentos realizados

En esta sección se evalúan los modelos implementados en tres experimentos distintos:

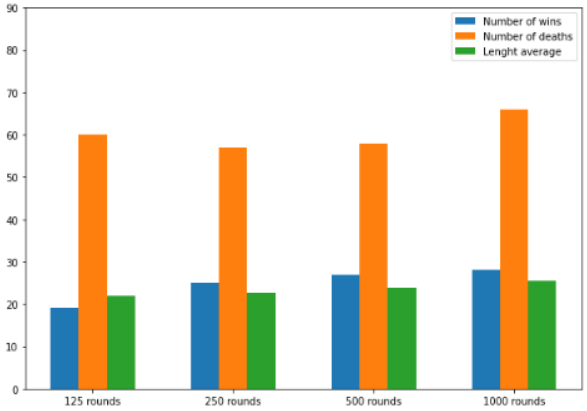
- Comparación de MCTS con distintos valores en los parámetros de *número de simulaciones* y *profundidad*.
- Comparación de los conjuntos de datos de entrenamiento de la red neuronal para observar cuál es la afectación que tiene cada uno de ellos sobre las predicciones que realiza.
- Comparación de los diferentes modelos implementados usando los agentes que han dado mejor resultado: determinista, mejor modelo de MCTS, mejor red neuronal entrenada con datos generados por partidas simuladas con MCTS y mejor red neuronal entrenada con datos de partidas reales.

3.1. *Monte Carlo Tree Search* con distinto *número de simulaciones* y *profundidad*

Dos parámetros que se deben modificar para obtener mejores resultados con el método de MCTS son el *número de simulaciones* y la *profundidad*. El primero indica cuántas simulaciones debe realizar el agente desde su posición actual por los caminos que tiene disponible para poder decidir cuál de ellos le dará una mayor puntuación. El segundo parámetro indica el número de movimientos que debe realizar el agente por un camino. Cuanto mayor sea la profundidad, más amplia es la exploración que el agente realiza de su entorno dado que puede ir a una distancia más larga por un camino. No obstante, también se requiere un mayor número de simulaciones para poder analizar todos los posibles caminos que se descubren a mayor profundidad. Por este motivo, es necesario encontrar el equilibrio entre *número de simulaciones* y *profundidad* que permita al agente obtener un mejor resultado.



(a) Simulaciones con profundidad 4.



(b) Simulaciones con profundidad 6.

Figura 7: Competición de 100 partidas con diferente número de simulaciones de MCTS.

Se han realizado dos competiciones, de 100 partidas cada una, entre 4 agentes con el modelo de MCTS implementado. La configuración de los agentes de la primera competición tiene una profundidad de 4 y 125 simulaciones por movimiento para el primer agente, 150 simulaciones para el segundo, 500 para el tercero y 1000 para el cuarto agente. La configuración de los agentes de la segunda competición es la misma pero a una profundidad de 6. El resultado se muestra en la figura 7.

Por último, se ha realizado una competición más con los agentes de 125 y 250 simulaciones a profundidad 4, y los agentes de 500 y 1000 simulaciones a profundidad 6. El resultado se puede ver en la figura 8.

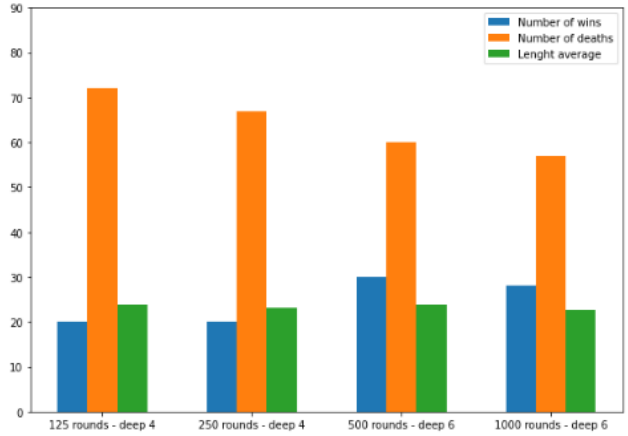


Figura 8: Competición de 100 partidas con los agentes de 125 y 250 simulaciones a profundidad 4 (los dos vencedores de la figura 7a), y los agentes de 500 y 1000 simulaciones a profundidad 6 (los dos vencedores de la figura 7b).

El mayor problema de MCTS es el tiempo computacional que emplea para las simulaciones. Cuantas más simulaciones y más profundidad, más tiempo tarda el agente en dar un resultado final. En Hungry Geese, los agentes tienen 1 segundo para emitir una decisión y, si se pasan, se añade la diferencia a un contador que al llegar a 60 segundos provoca la eliminación del agente. En la tabla 1 se pueden observar los tiempos que tardan de media cada uno de los agentes para dar una respuesta sobre qué acción realizar.

3.2. Conjunto de datos de entrenamiento de la red neuronal convolucional

Los conjuntos de datos son una de las partes más importantes para poder entrenar un modelo de inteligencia artificial supervisado. En este TFM se han implementado dos formas de obtener conjuntos de datos: mediante partidas simuladas con MCTS y mediante partidas reales obtenidas mediante *scraping*. En esta sección se expone un experimento para cada una en los que compiten varios agentes entrenados con conjuntos de datos obtenidos con diferentes parámetros y contextos.

Agente MCTS	segundos/iteración
125 sim., 4 prof.	0.115
250 sim., 4 prof.	0.216
500 sim., 4 prof.	0.452
1000 sim., 4 prof.	0.867
125 sim., 6 prof.	0.148
250 sim., 6 prof.	0.282
500 sim., 6 prof.	0.583
1000 sim., 6 prof.	1.121

Cuadro 1: Media del tiempo en la que los agentes dan una respuesta.

Experimento con partidas simuladas

En la figura 9 se muestra una competición de 100 partidas de cuatro agentes usando la red neuronal convolucional explicada en la sección 2.4. Los agentes han sido entrenados a partir de partidas simuladas con MCTS con distinto número de simulaciones y profundidad 6. El número de simulaciones escogidas para entrenar cada red neuronal son 125, 250, 500 y 1000, respectivamente. Para el entrenamiento se ha reservado un 20 % del conjunto de datos para la fase de validación del modelo. La precisión máxima alcanzada sobre el conjunto de datos de validación ha sido de 0.655, 0.694, 0.709 y 0.736, respectivamente.

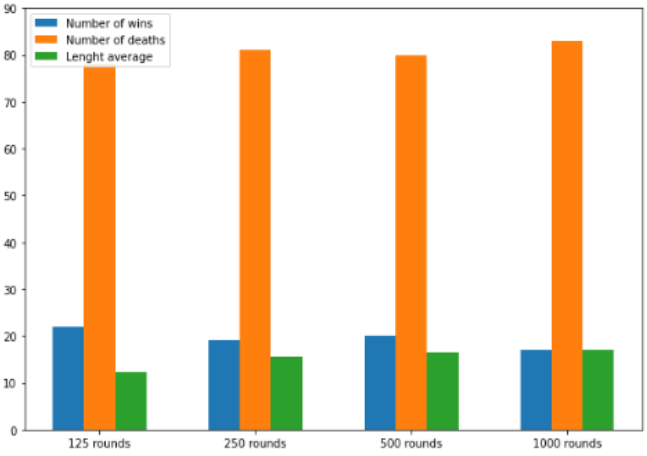


Figura 9: Competición de 100 partidas con redes neuronales entrenadas con partidas simuladas con MCTS con distinto número de simulaciones y profundidad 6.

Experimento con partidas reales

En la figura 10 se muestra una competición de 100 partidas de cuatro agentes usando la red neuronal convolucional explicada en la sección 2.4. Los agentes han sido entrenados a partir de partidas reales de otros agentes que se encuentran en diferentes rangos de puntuación. De esta forma, se puede

estudiar la afectación que tiene entrenar una red neuronal con las partidas de agentes que, supuestamente, son mejores que otros, según la tabla de clasificación. Las puntuaciones escogidas para entrenar cada red neuronal son entre 900 y 1000 (sin premio según su posición en la clasificación), entre 1001 y 1100 (con el premio de bronce), entre 1101 y 1200 (con el premio de plata) y mayores a 1200 (con el premio de oro). Para el entrenamiento se ha reservado un 20 % del conjunto de datos para la fase de validación del modelo. La precisión máxima alcanzada sobre el conjunto de datos de validación ha sido de 0.782, 0.752, 0.755 y 0.775, respectivamente.

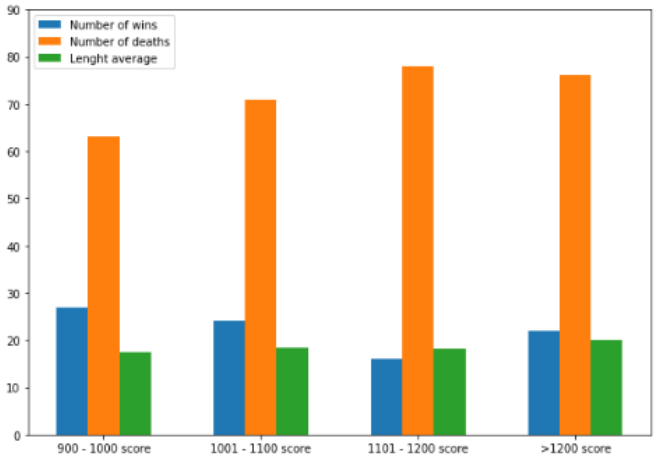


Figura 10: Competición de 100 partidas con redes neuronales entrenadas con partidas reales de agentes que se encuentran en diferentes rangos de puntuación.

3.3. Comparación de los agentes implementados

Para saber qué agente es más válido para competir en Hungry Geese, tanto por elección de las mejoras acciones como por rapidez en decidir qué acción tomar, se ha realizado un experimento en el que compiten el agente determinista, el mejor agente de MCTS (1000 simulaciones a profundidad 6), la mejor red neuronal entrenada con datos generados por partidas simuladas con MCTS (125 simulaciones a profundidad 6) y la mejor red neuronal entrenada con datos de partidas reales (puntuación de 900 a 1000). El resultado se puede ver en la figura 11.

4. Análisis de los resultados

En esta sección se analizan los resultados obtenidos por los experimentos realizados y se detallan los motivos por los cuales una metodología funciona mejor o peor que otra. De manera individual, se analiza el método de MCTS y el método de entrenamiento de la red neuronal convolucional. Por último, se expone un análisis final de todos los modelos implementados en conjunto. En la tabla 2 pueden verse los resultados obtenidos en las diferentes competiciones realizadas en los experimentos.

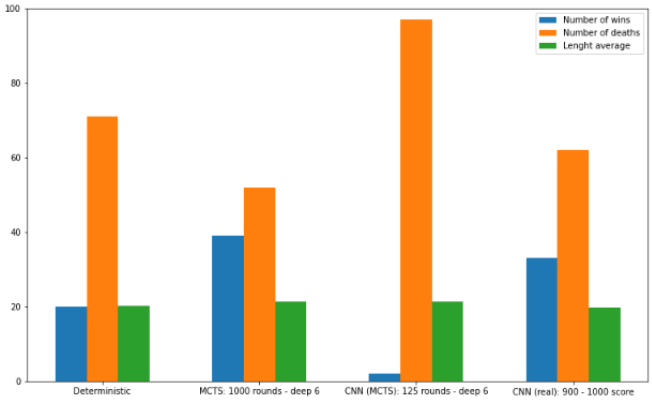


Figura 11: Competición de 100 partidas con el agente determinista, el mejor agente de MCTS, la mejor red neuronal entrenada con datos generados por partidas simuladas con MCTS y la mejor red neuronal entrenada con datos de partidas reales.

4.1. Análisis de *Monte Carlo Tree Search*

En la figura 7a, se puede observar que los agentes con mayor número de victorias y con menor número de muertes son los agentes con 125 y 250 simulaciones. Este hecho, que parece contradictorio porque la teoría dice que cuantas más simulaciones de MCTS, más preciso es el resultado, puede deberse a una particularidad táctica del juego. Si bien es cierto que el agente que consigue ser más largo y sobrevivir hasta el final, es el que gana, es un hecho que puede ir en contra en las primeras etapas de una partida. La explicación es que cuanto más largo sea un agente, más dificultades tiene para escapar de un espacio pequeño y, por el contrario, cuanto más corto sea, más facilidad tiene. La función empleada para calcular la recompensa que obtiene un agente por ir por un camino da más peso a la longitud del agente, provocando que siempre que encuentre una pieza de comida, vaya a por ella porque le hace crecer, según la precisión que tenga (número de simulaciones realizadas). Por lo tanto, lo que ocurre, es que los agentes con 500 y 1000 simulaciones son más precisos a una **profundidad de 4** para ir por un camino con el objetivo de hacerse más grandes, lo cual les penaliza haciendo que no sepan detectar que, realmente, están entrando en un camino sin salida (figura 12) y mueran antes que los agentes con 125 o 250 simulaciones, que son más cortos por tener menos precisión con ese número de simulaciones y profundidad. Cuando los agentes con 125 y 250 simulaciones se encuentran solos en el tablero, ya tienen la posibilidad de crecer más fácilmente. En la figura 7, esto se refleja en la métrica *lenght average*, que indica la longitud media que alcanza cada agente solo en las partidas donde sobreviven hasta el final. Podemos ver que, para los agente con 250 y 1000 simulaciones, la longitud media es muy semejante, pero el primero ha sobrevivido más veces que el segundo.

Competición MCTS Prof. 4 – Figura 7a			
Agente	**# victorias**	**# muertes**	**Longitud media**
125 simulaciones	22	69	24
250 simulaciones	33	61	26
500 simulaciones	19	73	22
1000 simulaciones	22	73	27
Competición MCTS Prof. 6 – Figura 7b			
Agente	**# victorias**	**# muertes**	**Longitud media**
125 simulaciones	19	60	23
250 simulaciones	27	56	23
500 simulaciones	29	57	24
1000 simulaciones	30	65	26
Competición MCTS Prof. 4 vs Prof. 6 – Figura 8			
Agente	**# victorias**	**# muertes**	**Longitud media**
125 simulaciones, profundidad 4	20	73	24
250 simulaciones, profundidad 4	21	68	23
500 simulaciones, profundidad 6	32	60	24
1000 simulaciones, profundidad 6	30	57	23
Competición NN (datos MCTS) – Figura 9			
Agente	**# victorias**	**# muertes**	**Longitud media**
125 simulaciones, profundidad 6	22	78	12
250 simulaciones, profundidad 6	18	82	15
500 simulaciones, profundidad 6	19	81	16
1000 simulaciones, profundidad 6	16	84	16
Competición NN (datos reales) – Figura 10			
Agente	**# victorias**	**# muertes**	**Longitud media**
900 - 1000 puntuación	28	63	18
1001 - 1100 puntuación	27	72	20
1101 - 1200 puntuación	18	79	20
>1200 puntuación	25	76	22
Competición final – Figura 11			
Agente	**# victorias**	**# muertes**	**Longitud media**
Determinista	20	77	20
MCTS: 1000 sim., prof. 6	40	58	21
CNN (MCTS): 125 sim., prof. 6	2	98	21
CNN (real): 900 - 1000 puntuac.	33	63	20

Cuadro 2: Resultados obtenidos en las diferentes competiciones realizadas en los experimentos.

Este hecho también se muestra en las gráficas de la segunda competición (figura 7b), pero únicamente en el número de muertes. En este caso, el agente con 1000 simulaciones es el que más veces muere pero, a la vez, es el que más partidas ha ganado. La explicación de este hecho se debe a lo expuesto en el párrafo anterior, es decir, cuanto más largo sea el agente, más difícil es que sobreviva. No obstante, gana más partidas porque a una mayor profundidad es capaz de ser más preciso sobre lo que puede ocurrir en los próximos turnos y, por lo tanto, evitar un mayor número de caminos que no tienen salida.

En la competición realizada con los agentes de 125 y 250 simulaciones a profundidad 4 (los dos vencedores de la figura 7a), y los agentes de 500 y 1000 simulaciones a profundidad 6 (los dos vencedores de la figura 7b), se puede observar que los agentes que más partidas han ganado han sido los de 500 y 1000 simulaciones a una profundidad de 6. Por lo tanto, se puede concluir que un agente basado en MCTS da mejores resultados cuanto mayor sea la profundidad a la que puede ir. Si bien es cierto que un mayor número de simulaciones hace que el agente sea más preciso, la profundidad es lo que provoca que pueda detectar mejores posibilidades en los siguientes turnos.

4.2. Análisis de la red neuronal convolucional

En la figura 9, la cual muestra el resultado de la competición cuyos agentes han sido entrenados a partir de partidas simuladas con MCTS, se puede observar como los agentes han muerto en la mayoría de partidas y, cuando no lo han hecho,

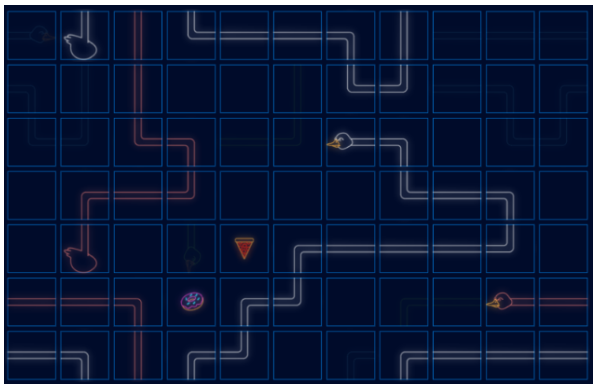


Figura 12: El agente de color rojo, con MCTS de 1000 simulaciones a profundidad de 4, no es capaz de detectar que se está metiendo por un camino sin salida y, finalmente, acaba colisionando.

han sido los ganadores. La explicación a esta casuística es que el número de victorias y el número de muertes de cada agente suma el total de partidas realizadas, 100. Este hecho implica que ninguna partida ha llegado a los 200 pasos con más de 1 agente vivo, es decir, todos han muerto antes o solo uno ha quedado con vida. Esto demuestra que la metodología de usar partidas simuladas por MCTS no da un buen resultado, debiéndose a que el conjunto de datos obtenido contiene acciones que se han calculado mediante la aleatoriedad del modelo de MCTS, por lo que un porcentaje de valores es incorrecto, afectando en el entrenamiento de la red neuronal.

En la figura 10, la cual muestra el resultado de la competición cuyos agentes han sido entrenados a partir de partidas reales, se puede observar un resultado totalmente distinto al esperado, ya que un entrenamiento usando movimientos de agentes con mayor puntuación no da los mejores resultados. La figura 10 muestra como la red neuronal entrenada con movimientos de agentes con menos puntuación es la que mayor número de veces ha ganado y menor número de veces ha muerto. Este mismo experimento se ha realizado tres veces más, obteniendo el mismo resultado, por si distintas casuísticas de las partidas pudieran haber llevado a esos valores.

Uno de los mayores defectos que tiene este modelo, al igual que el de MCTS, es que no sabe predecir si está entrando en un camino sin salida. Dado que no estamos en un entorno de *reinforcement learning*, no se especifica si la acción realizada por el agente ha sido correcta o no. Además, en la entrada de la red neuronal no se infieren los parámetros globales de la partida que hacen referencia al número de turnos que han transcurrido o a los turnos que quedan para que se elimine un segmento de cada agente. Esto provoca que, en el entrenamiento de la red neuronal, se detecte, sobre todo, que los mejores agentes van en dirección a una pieza de comida. No obstante, la red neuronal no es capaz de inferir en qué momento los mejores agentes no van hacia la comida. Por el contrario, los agentes con menor puntuación no suelen ir directamente a por una pieza de comida, sino más bien intentan reducir el riesgo de morir. Eso puede deberse a que la mayoría de los agentes de ese rango, posiblemente, son deterministas o tienen algoritmos clásicos como, por ejemplo, *minimax*, el cual intenta minimizar la pérdida máxima esperada, es decir, interpretan que ir a por comida es lo mejor, pero ante ciertas situaciones minimizan el riesgo de morir con el simple hecho de sobrevivir. Es posible que debido a este motivo, la red neuronal con mayor precisión sobre su conjunto de datos de validación haya sido la entrenada con los movimientos de los agentes con menor puntuación, ya que recurren a patrones más comunes.

En la figura 13 se puede observar como el agente de color rojo, el cual se ha entrenado con los movimientos de agentes con puntuación mayor a 1200, se ha encerrado sobre sí mismo con el fin de ir a por la pieza de comida, mientras que los agentes de color blanco y azul, que han sido entrenados con los movimientos de agentes con puntuación más baja, se están protegiendo dando vueltas sobre sí mismos.

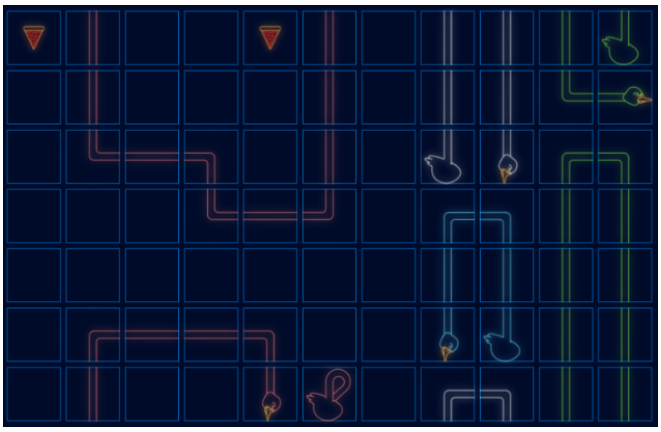


Figura 13: Competición de redes neuronal con entrenamiento de partidas reales con diferentes rangos de puntuación. Blanco 900 – 1000, azul 1001 – 1100, verde 1101 – 1200, rojo >1200.

4.3. Análisis comparativo de los agentes implementados

En la figura 11 se puede observar el resultado de 100 partidas de los agentes implementados. El agente que peor resultado ha obtenido ha sido la red neuronal entrenada con datos generados por partidas simuladas con MCTS debido a que solo ha conseguido ganar 3 veces y ha muerto en 97 partidas. Como hemos podido analizar en la figura 7, ninguno de los agentes entrenados con esta metodología obtenía un buen resultado y esta es la razón por la cual este agente ha sido el peor de los cuatro. El agente determinista ha quedado en tercera posición, ganando 20 partidas y muriendo en 77, por lo que en 3 partidas no ha ganado ni perdido, pero ha llegado hasta el máximo de 200 turnos. El problema de esta implementación, y de que no se haya creado un espacio de búsqueda en profundidad que compruebe cuál es el mejor camino, es que comete errores cuando todos los agentes tienen una longitud considerable, lo cual provoca que haya menos movimientos válidos y

más caminos sin salida.

Por otro lado, los agentes de MCTS y la red neuronal entrenada con datos de partidas reales han obtenido, aproximadamente, el mismo número de victorias, aunque la red neuronal ha muerto 5 veces más que el agente de MCTS. Ambos modelos tienen el mismo problema de no saber identificar que están entrando en un camino sin salida. Además, en la mayoría de partidas, estos dos agentes compiten entre sí hasta el final de la partida, sin que haya ningún contrincante más por haber muerto en turnos anteriores. Esto provoca que tengan más libertad de movimiento y, por lo tanto, que puedan llegar a las piezas de comida más fácilmente. No obstante, a medida que crecen, hay mayor probabilidad de que entren en un espacio del cual no sepan salir.

5. *Reinforcement Learning* mediante *actor-critic learning*

Con el objetivo de mejorar el agente basado en redes neuronales con entrenamiento supervisado, se ha llevado a cabo un análisis e implementación de un modelo basado en *Reinforcement Learning* mediante *actor-critic learning*. *Actor-critic* es una clase de algoritmo de políticas de gradiente (*policy gradient algorithms*) que está compuesto por dos componentes principales: una política y una función de valor estimado. La estructura de la política es conocida como *actor*, porque es la encarga de seleccionar las acciones, y la función de valor estimado es conocida como *critic*, debido a que "critica" las acciones hechas por el actor. Este método, introducido en [1], [20] y [29], ha sido extendido al aprendizaje profundo en [26], [23] y [12]. En [17] se ha realizado una combinación de redes neuronales y múltiples actores distribuidos, dando buenos resultados en Atari.

Esta sección describe el modelo implementado mediante *reinforcement learning* con *actor-critic learning* usando la metodología de entrenamiento *self-play*. Para este modelo se usa el concepto de "ventaja" (*advantage*) [28], que es la diferencia entre el resultado real de la partida y el resultado esperado, para mejorar el proceso de entrenamiento. De esta forma, se estima cómo una decisión particular contribuye al resultado final. Para calcular la ventaja, primero es necesario un estimador del valor del estado (*state-value*), el cual se denota como $V(s)$. Este es el retorno esperado del agente, es decir, indica si la posición del tablero es buena o mala para un agente. Si $V(s)$ es un valor cercano a 1, el agente se encuentra en una posición favorable, y si está cercano a -1, el agente se encuentra en una posición desfavorable. La función para el parámetro ventaja dada una acción a es un estado s es

$$A(a, s) = R - V(s)$$

donde R es una estimación de $Q(a, s)$ que hace referencia a la recompensa obtenida por el agente al finalizar la partida. El parámetro $V(s)$ no se puede obtener de forma exacta debido a la gran cantidad de posibles estados del juego. Por lo tanto, se usa una red neuronal para aproximar dicho valor.

5.1. Metodología de entrenamiento

La metodología de entrenamiento usada para este modelo es *self-play*. Este método, llamado más concretamente *Fictitious Self-Play (FSP)*, consiste en que un mismo agente juegue partidas repetidamente contra copias de sí mismo y guarde la experiencia de sus movimientos, junto con el estado del tablero y un indicador que señale si ha sido un movimiento que le ha llevado a ganar la partida o a perderla. Fue introducido en [7] y proviene del modelo de aprendizaje en juegos basado en *teoría de juegos*, creado por George Brown en 1951. Más concretamente, en esta implementación se ha usado el método *Neural Fictitious Self-Play (NFSP)*, que combina *FSP* con redes neuronales. Cada agente de una partida es controlado por una misma red neuronal, que juega contra sí misma, guardando la experiencia mencionada para, posteriormente, usarla como entrada en un entrenamiento supervisado de esa misma red neuronal para mejorarla. *NFSP* ha sido aplicado en [8], donde se ha demostrado que es capaz de converger de una forma fiable para aproximarse al equilibrio de Nash en juegos de *información imperfecta*.

El ciclo completo para entrenar el modelo mediante *self-play* es:

1. Inicializar el modelo con los pesos de la red neuronal generados de forma aleatoria.
2. Realizar un número determinado de partidas donde el modelo juegue contra copias de sí mismo. De esta forma se genera un conjunto de datos de entrenamiento.
3. Una vez realizadas las partidas, se debe entrenar el modelo con el conjunto de datos obtenido en 1 solo *epoch*, ya que no se puede entrenar más sobre datos que aún no son suficientemente consistentes debido a la falta de entrenamiento del modelo.
4. Al acabar un *epoch*, se debe comparar el modelo con su versión previa para comprobar si ha mejorado. Para ello, se genera un número determinado de partidas y, si la nueva versión gana más de la mitad de las partidas, el modelo ha mejorado y, por lo tanto, se realiza un nuevo ciclo de entrenamiento desde el punto 2 con la nueva versión del modelo. Por el contrario, si no gana más de la mitad de las partidas, se genera un nuevo conjunto de entrenamiento, se realiza un nuevo *epoch* sobre la versión anterior con ambos conjuntos de entrenamiento y se vuelve a evaluar el modelo.

5.2. Arquitectura del modelo

La figura 14 muestra la arquitectura creada para el modelo con *reinforcement learning*. Se puede observar como la parte del *bloque convolucional* es el mismo que el de la figura 6. No obstante, la diferencia empieza a partir de la segunda capa densa, la cual se divide en dos capas de 512 neuronas para

poder emitir dos salidas del modelo. Una red neuronal con dos salidas conlleva a que cada salida sirve como regulador de la otra salida, por lo que ayuda a minimizar el *overfitting*.

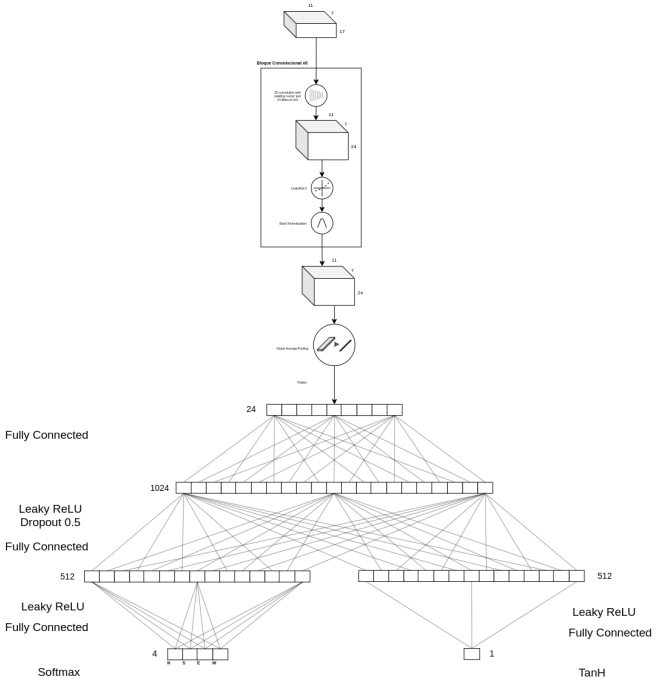


Figura 14: Arquitectura de la Red Neuronal con *Reinforcement Learning*.

Las dos salidas del modelo son el valor del *actor* (*actor-value*), el cual representa una distribución de probabilidad sobre las posibles acciones a realizar por el agente, y el valor del estado (*state-value*) que, tal y como se ha explicado anteriormente, es $V(s)$ y es el retorno esperado del agente dentro del rango -1 y 1, es decir, indica si la posición del tablero es buena o mala para un agente. En la primera salida se usa la función de activación *softmax* para que la suma de las probabilidades de cada acción sea 1, mientras que en la segunda salida se usa la función de activación *tanh*. Para la fase de entrenamiento, los valores de las etiquetas de la primera salida, en lugar de ser *one-hot*, son el parámetro "ventaja" únicamente para la acción realizada dado un estado del tablero. Para el resto de acciones, el valor es 0.

La entrada del modelo tiene la misma estructura que la figura 5.

5.3. Resultados obtenidos en *reinforcement learning*

Los resultados obtenidos mediante este modelo no han sido satisfactorios en la fase de entrenamiento. Este hecho se debe a que no se ha realizado un entrenamiento con todo el tiempo que requiere un modelo de estas características. Hay que tener en cuenta que las primeras fases se realizan con una red neuronal cuyos pesos han sido inicializados de manera aleatoria, por lo que sus movimientos en las partidas para generar un conjunto de datos son aleatorios. Esto da como resultado un conjunto de datos de entrenamiento desbalanceado donde la mayoría de las acciones son "negativas" y, en muy pocas ocasiones, hay acciones "positivas" obtenidas de manera casual. A medida que avanza la fase de entrenamiento, el conjunto de datos se balancea debido a que el agente va aprendiendo qué acciones son positivas o negativas. No obstante, para llegar a ese punto se requiere de un volumen alto de tiempo y recursos. Por ejemplo, el ganador de la competición compartió (`https://www.kaggle.com/c/hungry-geese/discussion/263279`) que para entrenar su modelo necesitó tres semanas de entrenamiento mediante *self-play*, generando dieciséis millones de partidas sobre una máquina con 1 GPU y 64 *workers*. Además, para terminar de perfeccionar el entrenamiento, entrenó cuatro días más, generando tres millones de partidas contra otros agentes ya entrenados, sobre una máquina con 1 GPU y 288 *workers*. Debido a que el modelo explicado en este documento ha entrenado un total de 9 horas seguidas y con una memoria insuficiente en las máquinas gratuitas que ofrece Kaggle, no se ha llegado a obtener un resultado satisfactorio. También se intentó entrenar en dos tandas de distintos días durante 9 horas cada día, pero tampoco fue suficiente. Además, dado que es necesario generar una alta cantidad de partidas y realizar múltiples iteraciones para poder ver un mínimo resultado de entrenamiento, el tiempo dedicado no ha sido suficiente y, por lo tanto, no se obtiene un resultado concluyente sobre este modelo.

6. Conclusión

Los modelos de inteligencia artificial usados para jugar a Hungry Geese sirven como base inicial para la aplicación en múltiples ámbitos donde se necesite inteligencia artificial como, por ejemplo, reconocimiento facial o de objetos, segmentación de clientes en marketing, modelos de finanzas, etc. Tanto el modelo basado en Monte Carlo como la red neuronal con entrenamiento supervisado como el modelo de *reinforcement learning* mediante *actor-critic learning* se han usado en multitud de proyectos y todos con necesidades distintas, como se puede ver en la bibliografía presentada. El objetivo de este proyecto era analizar y comparar estos modelos de inteligencia artificial con el fin de saber cuál es la que tiene mejor aplicación en el juego de Hungry Geese.

Se ha observado que el modelo de *Monte Carlo Tree Search* permite competir y ganar partidas reales, pero con muchas carencias que se agravan en los momentos donde no es capaz de detectar que está entrando en un camino sin salida. Este hecho se podría atenuar aumentando el parámetro de profundidad del árbol, el cual se ha observado que tiene más importancia que el número de simulaciones necesarias para poder detectar mejores posibilidades en los siguientes turnos. No obstante, esto no es trivial debido a la limitación de tiempo que hay en cada turno del juego.

El modelo de redes neuronales con entrenamiento super-

visado ha demostrado ser mejor que el de MCTS cuando el conjunto de datos de entrenamiento se ha realizado a partir de partidas reales obtenidas mediante *scraping*, además de dar un resultado mucho más rápido que MCTS. No obstante, habiendo conseguido un 80 % de precisión (*accuracy*) sobre el conjunto de datos de validación del modelo no es suficiente para conseguir un resultado óptimo en el que pueda ganar a otros agentes entrenados mediante *reinforcement learning*.

Por este mismo motivo, se ha decidido investigar un modelo entrenado mediante *reinforcement learning*, el cual no ha dado los resultados esperados en la fase de entrenamiento y no ha sido posible probarlo en partidas reales. El hecho de que la mayoría de los agentes que han conseguido una puntuación alta en el juego se hayan basado en modelos de *reinforcement learning*, hace pensar que es la técnica indicada para conseguir los mejores resultados. No obstante, dedicar más tiempo de entrenamiento y mejorar el modelo en caso necesario se deja como trabajo futuro en el que se pueda obtener un resultado válido y concluyente para poder analizar.

Debido a que este trabajo es un análisis y una aplicación de distintos modelos de inteligencia artificial ya creados con anterior, es decir, no se ha implementado ningún modelo o técnica novedosa, la verdadera importancia que, bajo mi punto de vista, tiene este proyecto es el aprendizaje obtenido sobre los modelos expuestos a partir de la bibliografía aportada. Las referencias mencionadas a lo largo del artículo muestran diferentes aspectos y puntos de vista de los modelos implementados, desde los primeros artículos que fueron publicados sobre ellos hasta las técnicas concretas explicadas en los más recientes de 2019, 2020 e, incluso, 2021. El código fuente con los modelos implementados y los experimentos realizados puede encontrarse en `https://github.com/JoanMartin/hungry-geese-kaggle`.

Referencias

[1] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[2] Guillaume Chaslot, Jos Uiterwijk, Bruno Bouzy, and H. Herik. Monte-carlo strategies for computer go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, 01 2006.

[3] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *International Conference on Computers and Games*, 4630, 05 2006.

[4] DeepMind. Deepmind technologies limited, April 2019. London, United Kingdom.

[5] Guillaume Maurice Jean-Bernard Chaslot. Monte-Carlo Tree Search. In *Ph.D. dissertation, Ph.D. thesis*, pages 19–31, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands, 2010.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *In Proceedings of the 32nd International Conference on Machine Learning*, 07 2015.

[8] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *NIPS Deep Reinforcement Learning Workshop*, 03 2016.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[10] Kaggle. Hungry geese competition, 2021.

[11] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML*, 2006:282–293, 09 2006.

[12] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, Conference Track Proceedings*, San Juan, Puerto Rico, 2016.

[13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *2nd International Conference on Learning Representations, Conference Track Proceedings*, Banff, AB, Canada, 2014.

[14] Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020*, 2020.

[15] Nicolas Loizou and Peter Richtárik. Linearly convergent stochastic heavy ball method for minimizing generalization error. *NIPS-Workshop on Optimization for Machine Learning*, 10 2017.

[16] Nicolas Loizou and Peter Richtárik. Momentum and stochastic momentum for stochastic gradient, newton, proximal point and subspace descent methods. *Computational Optimization and Applications*, 77, 12 2020.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[18] Pierre Perick, David L. St-Pierre, Francis Maes, and Damien Ernst. Comparison of different selection strategies in Monte-Carlo Tree Search for the game of Tron. *IEEE Conference on Computational Intelligence and Games*, pages 242–249, 2012.

[19] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[20] Richard Stuart Sutton. Temporal Credit Assignment in Reinforcement Learning. In *PhD thesis*, University of

Massachusetts Amherst, 1984.

[21] Rogba. Public code and the evil twin effect, July 2021. Kaggle Discussions.

[22] Spyridon Samothrakis, David Robles, and Simon M. Lucas. A UCT Agent for Tron: Initial Investigations. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 365–371, 2010.

[23] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *4th International Conference on Learning Representations, Conference Track Proceedings*, San Juan, Puerto Rico, 2016.

[24] Shoheiazuma. Alphageese baseline, April 2021. Kaggle Notebooks.

[25] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529 (7587):484–489, January 2016.

[26] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, 06 2014.

[27] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[28] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[29] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12, NIPS Conference*, pages 1057–1063, Denver, Colorado, USA, 1999. The MIT Press.

[30] Yuricat. Smart geese trained by reinforcement learning, January 2021. Kaggle Notebooks.