2/1/2023

Desarrollo de un motor de ajedrez empleando redes neuronales y Python

Jorge García Martín
KSCHOOL DATA SCIENCE MASTER'S DEGREE

Contents

Introducción

Los motores de ajedrez son programas de ordenador que, entre otras cosas, ayudan a los jugadores a mejorar su juego y analizar partidas. Desde su aparición en la década de 1970, han experimentado un increíble avance tanto en términos de velocidad como de precisión. Con la evolución de la tecnología, han sido capaces de utilizar algoritmos cada vez más complejos y sofisticados para analizar y evaluar posiciones en el tablero.

En la actualidad, los motores de ajedrez utilizan una amplia gama de técnicas de inteligencia artificial, incluyendo redes neuronales, para mejorar sus habilidades de juego y análisis. Este proyecto se enfoca en el desarrollo de un motor de ajedrez en Python utilizando redes neuronales.

A diferencia de los motores de ajedrez convencionales, que se basan en reglas y heurísticas predefinidas, los motores basados en redes neuronales se benefician de un enfoque más flexible y adaptable. La utilización de redes neuronales permite aprender de experiencias previas y descubrir patrones que ayudan a la evaluación de posiciones de una forma menos predecible y más natural.

Objetivo del Proyecto

El objetivo principal de este proyecto es explorar la aplicabilidad de las técnicas de inteligencia artificial, específicamente las redes neuronales, en el desarrollo de motores de ajedrez. Este proyecto busca entender mejor cómo las redes neuronales pueden ser utilizadas para mejorar la capacidad para analizar y evaluar posiciones en el tablero.

Este proyecto se lleva a cabo con un enfoque explorador, con el objetivo de aprender más sobre cómo funcionan las redes neuronales y cómo se pueden aplicar en este campo específico. A lo largo del proceso, se han evaluado diferentes enfoques y se han realizado ajustes para mejorar la eficacia del motor.

Descripción inicial

1. Instrucciones de uso

Para replicar el proyecto es necesario seguir los siguientes pasos:

- Clonar el repositorio del siguiente enlace. Repositorio GitHub.
- Crear un entorno nuevo importando el archivo chess_env.yml o bien creando un entorno manualmente e instalando el archivo requirements.txt. Ambos archivos se encuentran en el directorio base del repositorio.
- Descarga los datos que se encuentran en la carpeta de drive del proyecto. Carpeta Google Drive.
- Coloca los datos en sus respectivos directorios de acuerdo al documento data_instructions.txt disponible en Google Drive.

El procesado de datos se realiza en los notebooks que se encuentran en la carpeta data/:

- 00_data_sampling.ipynb
- 01_board_generator.ipynb

Los modelos de redes neuronales, su generación y el testeo de los mismos puede encontrarse en los notebooks de la carpeta engine/:

- engine_functions.ipynb
- data_functions.py
- engine_generator.ipynb

El código de la interfaz gráfica puede encontrarse en los scripts dentro de la carpeta chess_game/gui/:

- board.py
- config.py
- constants.py
- engine.py
- game.py
- main.py
- model.py
- mouse.py
- move.py
- piece.py

1.1Instrucciones de uso de la interfaz gráfica.

Para lanzar la interfaz gráfica con éxito deben de estar instaladas todas las herramientas necesarias para la ejecución. Si se ha realizado el apartado anterior con éxito todas deberían de estar instaladas.

Los pasos necesarios para jugar contra el motor son:

1. Abre una ventana de comandos en la terminal.
2. Ve al directorio del proyecto. Por ejemplo:

C:Users\MagnusCarlsen\Desktop\Best_Chess_Engine_TFM

3. Utilizando python, ejecuta el script main.py. Por ejemplo:

python c:/Users/MagnusCarlsen/Desktop/Best_Chess_Engine_TFM/chess_game/gui/main.py

4. Disfruta de la partida.

2. Datos utilizados

Los datos empleados en el proyecto pueden encontrarse al completo en la carpeta de Google Drive indicada anteriormente. Este es un listado de ellos y sus contenidos:

- lichess_db_standard_rated_2020-02.pgn.bz2
    - Descargado de lichess.org. Contiene las partidas jugadas en febrero de 2020 en lichess en formato png. Ocupa 13GB por lo que no se ha subido a drive. Está disponible públicamente en este enlace.
- lichess_db_standard_rated_2016-06.pgn.zst
    - descargado de lichess.org. Contiene las partidas jugadas en febrero de 2020 en lichess en formato png.
- games_sample.csv
    - Contiene una muestra de 100 partidas aproximadamente.
- games_processed_sample.csv
    - Contiene las partidas filtradas y procesadas de games_sample.csv
- random_boards_d3.npz
    - Archivo comprimido de numpy. Contiene 1.5 millones de posiciones y su evaluación con stockfish a profundidad de cálculo 3.
- random_boards_d6.npz

    - Archivo comprimido de numpy. Contiene 1.5 millones de posiciones y su evaluación con stockfish a profundidad de cálculo 6.
- random_boards_d10.npz
    - Archivo comprimido de numpy. Contiene 1.5 millones de posiciones y su evaluación con stockfish a profundidad de cálculo 10.

Desarrollo

1. Extracción de los datos

En primer lugar, se necesita obtener una base de datos de partidas evaluadas para poder tener un punto de partida sobre el que entrenar a la red neuronal.

Se barajan varias opciones sobre la fuente de los mismos, siendo las principales:

- Base de datos pública de lichess.org en la que se pueden encontrar todas las partidas jugadas dentro de la plataforma desde 2013.
- Scrapeo web de las partidas públicas de los grandes maestros que se encuentran en chess.com

Debido a la facilidad de acceso y al formato en el que se encuentran se elige empezar por la base de datos pública de lichess.org y dejar para una segunda iteración el scrapeo web de las partidas de los grandes maestros.

La lista con los distintos archivos disponibles en lichess en el momento del comienzo del proyecto se encuentran alojados en el archivo de texto *'data/DatabaseFileList.txt'*

En función de los resultados en la primera iteración se decidirá si merece la pena invertir esfuerzo en el scrapeo y limpieza de estos datos para su utilización en la red neuronal.

2. Limpieza de los datos

Este apartado puede seguirse en el notebook *'00_data_sampling.ipynb'* guardado dentro de la carpeta *'data/'*.

Los datos obtenidos de la base de datos de lichess.org están guardados en un formato estándar de ajedrez llamado con extensión .png. En este formato las partidas se almacenan junto con sus metadatos en líneas independientes. Cada una de las partidas puede diferenciarse de la anterior por una línea en blanco que las separa. Este formato de archivo puede leerse de múltiples maneras, incluyendo todas las que traten con archivos de texto.

Un ejemplo de partida que podemos encontrar es el siguiente:

Imagen 1: Ejemplo de partida

Cabe destacar que sólo un 6% de las partidas almacenadas en lichess han sido evaluadas utilizando un motor de código abierto llamado stockfish, por lo que es necesario desechar todas las partidas no evaluadas y guardar las válidas en un archivo aparte.

Con este objetivo en mente en un primer momento se desarrolló un método para leer los archivos **.png** empleando la librería *pandas* y la función *read_csv*. Una vez leído el archivo y sabiendo que las partidas ocupaban 18 líneas cada una se intentó leer en chunks de 18 categorizando las partidas como evaluadas y almacenándolas en archivos aparte para su posterior uso.

El principal problema de este método de lectura era el tiempo de carga de las líneas y que en un 1% de las ocasiones, cuando en una partida participaban grandes maestros, se añadían excepcionalmente 2 líneas adicionales indicando el título del jugador por lo que había que añadir otra condición que ralentizaba aún más la lectura.

Debido a esto se optó por una segunda aproximación al problema de lectura de los archivos desarrollando una función que leyera línea a línea el archivo **.png**, identificara el tipo de línea ante el que se encontrara y la categorizara, saltando de partida ca vez que encontrara una línea que describiera los movimientos de la partida. Este método, aunque muy efectivo, seguía siendo demasiado lento y tras muchas horas de limpieza de los datos se decidió desechar.

La base de datos de lichess tiene un total de 1TB de datos alojados en formato **.png** y comprimidos en **.zst**. Los archivos comprimidos en este formato cuando se descomprimen ocupan aproximadamente siete veces más por lo que estamos ante 7TB de partidas.

Para procesar la base de datos al completo se habrían necesitado meses de continuo procesado y para procesar la cantidad mínima necesaria para realizar el entrenamiento el tiempo estimado eran semanas por lo que de nuevo había que investigar otra aproximación al problema.

Por último, se intentó parsear las partidas con una librería que leyera el formato estándar y categorizara cada partida, guardando las evaluadas en un fichero aparte. Este método resultó ser más lento que el anterior.

Ante esta situación se decidió probar con alternativas fuera de las bases de datos públicas. Esto se decidió por la similitud de las bases de datos entre sí, pues todas seguían una lógica parecida y presentaban el mismo problema ante su lectura.

3. Generación de datos de forma alternativa

Este apartado puede seguirse en el notebook '*01_board_generator.ipynb'* guardado dentro de la carpeta '*data/'*

Debido a la imposibilidad en un tiempo prudencial de leer y categorizar las partidas necesarias para obtener las evaluaciones para entrenar la red neuronal se optó por una aproximación alternativa, generar localmente las posiciones y evaluaciones necesarias para entrenar a la red neuronal.

Para este cometido es necesaria la descarga de motor mencionado anteriormente, stockfish.

Se puede obtener del siguiente enlace:

https://stockfishchess.org/download/

Para replicar el proyecto es necesaria la descarga del stockfish adecuado al sistema operativo en el que vamos a ejecutarlo, en el caso de este proyecto se descargó la versión para Windows, disponible también en la carpeta de Google Drive asociada.

En primer lugar, se necesita generar los tableros que posteriormente serán evaluados por el motor.

Para ello, se emplea una función que tiene como objetivo generar tableros de ajedrez aleatorios utilizando la librería de Python '*chess'*, que en su clase *Board* representa el estado de una partida de ajedrez.

Se inicializa un tablero vacío y luego se realiza una cantidad aleatoria de movimientos dentro del rango 0 a '*max_depth'* definido. Cada movimiento es elegido al azar entre todos los movimientos legales en el tablero actual. Si la partida termina, la función deja de realizar movimientos. Al terminar, se devuelve el tablero resultante.

En segundo lugar, se evalúa cada tablero obtenido con stockfish siempre desde la perspectiva de las blancas para tener consistencia en el signo de la medida.

Para almacenar las posiciones de tal forma que tenga sentido para una red neuronal se debe de guardar en formato matricial toda la información relevante de cada posición que generemos.

Primero se declara una función que tiene como objetivo convertir las coordenadas de un cuadrado de ajedrez a un índice de matriz en dos dimensiones empleando un diccionario que relaciona las letras de las filas con su correspondiente índice en la matriz.

Después mediante otra función, se genera una matriz tridimensional que represente un tablero de ajedrez. En ella, se agrega la ubicación de las piezas, tanto de un color como de otro, y se incluyen también matrices con las jugadas legales y los ataques posibles para cada posición.

Para lograr esto, se crea una matriz de ceros de 14 x 8 x 8 y se agrega una vista para cada tipo de pieza. Se recorren las piezas blancas y las negras y se actualiza la matriz con la ubicación de cada una, obteniendo 6 matrices por cada jugador. Además, se añaden dos matrices más con la información sobre los movimientos legales y los ataques posibles para cada uno de los jugadores, lo que permite a la red neuronal conocer el estado actual de la partida con mayor precisión.

Con esto se consigue convertir un tablero de ajedrez en una representación numérica que pueda ser utilizada por una red neuronal para evaluar la mejor jugada.

Los resultados se guardan en la carpeta '*random_generated/*'.

Se guardan 3 archivos **.npz** compuestos por 1,5 millones de posiciones evaluadas con profundidades de cálculo 3, 6 y 10 respectivamente.

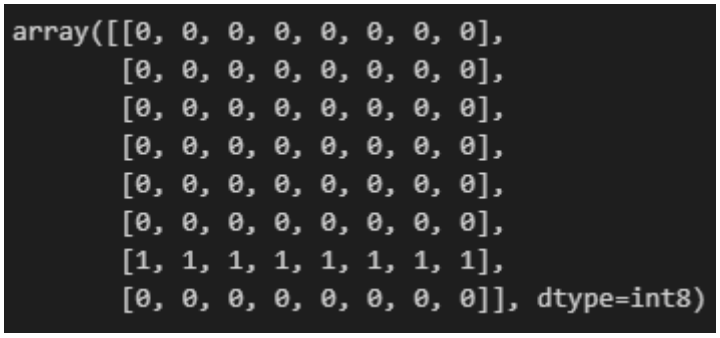

Imagen 2: ejemplo de formato de la matriz de 3 dimensiones. Posición de los peones blancos

4. Desarrollo de un motor de evaluación

Este apartado se puede seguir mediante los notebooks '*engine_generator.ipynb'* y '*engine_functions.ipynb'*, disponibles en la ubicación '*engine/'*.

Para el desarrollo del motor se ha optado por construir una red neuronal que consta de una entrada de 3 dimensiones, en la que se representa un tablero de ajedrez. La entrada tiene la forma de 14 × 8 × 8 vista en el apartado anterior.

Las redes neuronales han sido desarrolladas y ejecutadas usando la librería tensorflow-gpu, que optimiza los recursos de la GPU disponible para la ejecución de los modelos.

Para replicabilidad en ordenadores que no dispongan de GPU se recomienda instalar la librería intel-tensorflow que permite utilizar modelos optimizados para GPU gracias a su integración con las CPU gráficas de Intel. Este requisito está incorporado en el '*requirements.txt'* que se incluye en el repositorio.

La construcción del modelo de red neuronal se realiza mediante la función '*build_model'*. Primero se aplican varias capas de convolución 2D a la entrada para aprender las características y patrones presentes en el tablero. Las capas de convolución utilizan un kernel de tamaño 3 × 3 y la estrategia de padding *'same'* para mantener el tamaño de la salida igual a la entrada. La activación de ReLU se aplica a la salida de cada capa de

convolución. La función utiliza un bucle para aplicar tantas capas de convolución con la misma configuración como se haya especificado en la definición del modelo.

Después de aplicar todas las capas de convolución, la salida se aplana utilizando la función Flatten. Finalmente, se aplican dos capas densas para reducir la dimensionalidad de la entrada y producir una salida única. La primera capa densa tiene 64 unidades y utiliza una activación ReLU. La segunda capa densa tiene una sola unidad y utiliza una activación sigmoide, lo que significa que la salida será un número comprendido entre 0 y 1. Esta capa produce una probabilidad para la jugada de ajedrez que está siendo evaluada.

En general, se han escogido estas capas y esta arquitectura para que la red neuronal pueda aprender patrones complejos y características en el tablero de ajedrez. Las capas de convolución se utilizan para aprender características locales en el tablero, mientras que las capas densas se utilizan para aprender patrones a nivel global y producir una salida única. La activación ReLU se utiliza para asegurarse de que la red neuronal no produzca valores negativos en la salida, y la activación sigmoide se utiliza para producir una salida que sea una probabilidad.

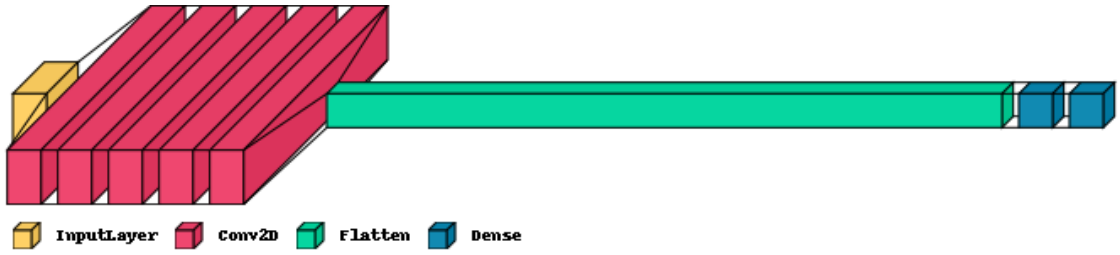

Imagen 3: Representación gráfica de la red neuronal

4.1 Redes Neuronales Residuales

Adicionalmente y como expansión sobre el desarrollo de la red neuronal, se ha trabajado en un modelo que utilice las conexiones residuales para mejorar la precisión de las predicciones.

Una conexión residual es un tipo de conexión en una red neuronal que permite a la información pasar directamente desde una capa anterior a una capa posterior, sin ser procesada por la capa intermedia. Esto se logra mediante la adición de la entrada de la

capa anterior a la salida de la capa intermedia. La idea detrás de las conexiones residuales es permitir que la red tenga acceso a información más antigua y original, lo que ayuda a mejorar la capacidad de la red para aprender y resolver problemas más complejos. Además, también ayuda a prevenir el problema del desvanecimiento del gradiente que puede ser un obstáculo para el aprendizaje efectivo.

La función que construye el modelo de red neuronal que incluye estas características puede encontrarse bajo la función *'build_model_residual'*.

En la función *build_model_residual*, también se utilizan capas convolucionales 2D, pero se agrega una conexión residual después de cada pareja de estas capas. La conexión residual permite que la información a través de la red sea retenida y agregada a la salida de la capa siguiente. Esto puede ayudar a prevenir el problema de "pérdida de información" en el modelo, lo que puede mejorar su capacidad para aprender patrones complejos y mejorar su capacidad generalizada. Adicionalmente, después de cada par de capas convolucionales y la correspondiente conexión residual, se agrega una capa de normalización batch y una activación ReLU.

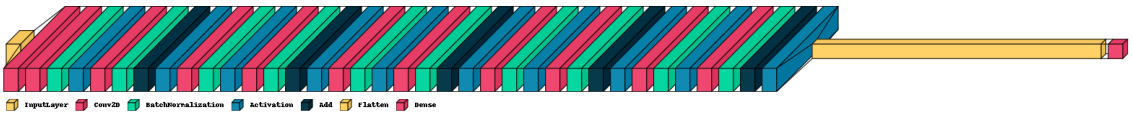

Imagen 4: Representación gráfica de la red neuronal con residuales

4.2 Ejecución de los modelos

Para el proyecto se han entrenado 4 modelos. 3 modelos variando el *dataset* de referencia entre los 3 *datasets* previamente generados y 1 modelo con el *dataset* más complejo de los anteriores y empleando la función que incluye el uso de residuales explicado en el apartado 4.1.

Para todos los modelos el tamaño de las capas convolucionales escogidas ('*conv_size*') es de 64 y la profundidad de capa escogida ('*conv_depth*') es de 5.

Los modelos se compilan con un optimizador Adam y como función de pérdida se escoge el error cuadrático medio (mse). El optimizador Adam combina aspectos de los optimizadores gradient descent y root mean squared propagation (RMSProp). Es efectivo en situaciones en las que los datos de entrenamiento tienen ruido o tienen una

distribución irregular y es una opción popular para ajustar los parámetros en modelos de *deep learning*.

Tras esto, el modelo en cuestión se entrena con el *dataset* seleccionado durante un máximo de 1000 épocas. Durante el entrenamiento, se utilizan dos *callbacks* para mejorar la eficiencia del entrenamiento: *ReduceLROnPlateau* y *EarlyStopping*. Ambos se emplean para mejorar el entrenamiento y prevenir el sobreajuste. *ReduceLROnPlateau* reduce la tasa de aprendizaje cuando la pérdida no disminuye después de una cantidad determinada de épocas. *EarlyStopping* detiene el entrenamiento temprano si la pérdida no disminuye después de una cantidad determinada de épocas o si la disminución es demasiado pequeña.

Todos los modelos, incluido es que emplea residuales, terminan su ejecución entorno a las 60 épocas. Los modelos sencillos tienen un tiempo de ejecución de 15 minutos y el modelo complejo tarda 4 veces más, llegando a la hora de entrenamiento.

5. Desarrollo de los algoritmos necesarios para la interacción entre el jugador y el motor

Para un motor de ajedrez la evaluación de posiciones es solo la mitad del trabajo. Una vez evaluadas las posibles jugadas el motor debe de escoger cual es la mejor de todas y ellas. Para esto se necesitan algoritmos de búsqueda. En este apartado se detallan los utilizados en el proyecto.

Los algoritmos detallados, así como las funciones que los implementan pueden encontrarse tanto en el notebook almacenado en '*engine/engine_functions.ipynb'* como en el script empleado en la gui que se encuentra en *'chess_game/gui/engine.py'*

5.1 Algoritmo Minimax

El algoritmo minimax es un algoritmo de búsqueda de decisión utilizado en juegos en los que dos jugadores compiten por el mejor resultado. Este algoritmo trabaja asumiendo que el oponente siempre jugará de la manera más eficiente para ganar, es decir, jugará de la manera que minimice la máxima ganancia del jugador.

Minimax se basa en una estructura de árbol, donde cada nodo representa un estado del juego y las hojas representan las evaluaciones de los estados finales. Cada nodo es evaluado como el máximo (para el jugador a maximizar) o el mínimo (para el jugador a minimizar) de los valores de sus hijos. El algoritmo busca la hoja con la mejor evaluación y devuelve su valor.

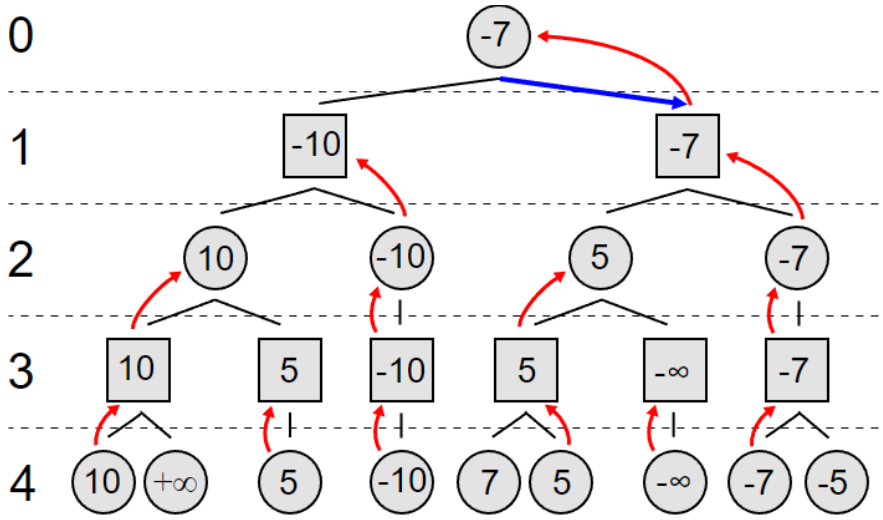

Imagen 5: Ejemplo de árbol de decisión y algoritmo minimax

5.2 Pruning

El pruning es una técnica que se utiliza para reducir el número de nodos que a explorar en el árbol generado por minimax. El objetivo es evitar la exploración de rutas innecesarias que no afectan el resultado final. En el proyecto se utiliza la técnica conocida como alpha-beta pruning, donde se mantienen dos valores, alpha y beta, que representan el valor máximo y mínimo que se espera obtener en un camino determinado. Si el valor de beta es menor o igual al valor de alpha, se detiene la exploración en ese camino. De esta manera, se ahorra tiempo al no tener que explorar todo el árbol para alcanzar el mejor resultado.

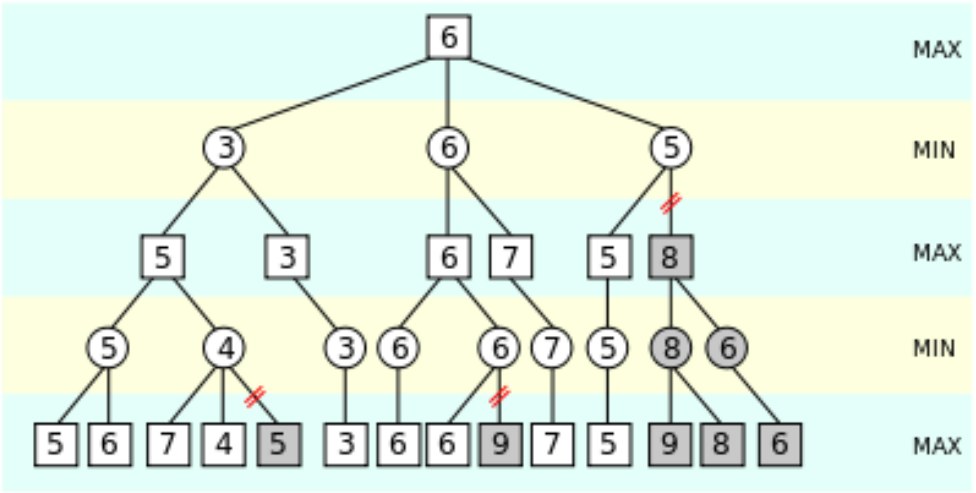

Imagen 6: Ejemplo de árbol de decisión, algoritmo minimax y Alpha-beta pruning

5.3 Tablas de transposición

Las tablas de transposición se pueden usar para almacenar resultados previos de la evaluación de una posición y reutilizarlos en lugar de volver a calcular la evaluación en cada búsqueda.

Esta técnica es similar al apha-beta pruning, pero utiliza una tabla de valores de historial para almacenar las valoraciones de los nodos previamente visitados.

La idea detrás del uso de tablas es que, si un nodo ha sido previamente evaluado y su valoración es lo suficientemente bueno o lo suficientemente malo, es posible que no sea necesario explorar ese nodo en profundidad en el futuro.

Para implementar esta técnica, primero se necesita agregar una tabla de valores de historial para cada nodo. Luego, antes de explorar un nodo, se debe verificar si el valor de ese nodo se encuentra en la tabla de valores de historial. Si es así, el valor previamente almacenado se usa para el cálculo de alpha o beta. Si no, el nodo se explora en profundidad como se hace normalmente.

6. Desarrollo de una interfaz gráfica de juego (GUI)

Para demostrar las capacidades del motor y ayudar a la visualización de los resultados es necesario una forma de comunicar gráficamente los movimientos escogidos al motor y una forma de transcribir los movimientos escogidos por dicho motor en una visualización que el jugador entienda con facilidad.

Para este apartado se ha desarrollado íntegramente empleando la librería *pygame* una interfaz gráfica que trabaja como enlace entre el jugador y el motor de juego.

En desarrollo de este apartado se realizado íntegramente con programación orientada a objetos. Los scripts de Python relativos a este apartado se pueden encontrar bajo la ubicación '*chess_game/gui'*.

En la siguiente figura se detallan las clases utilizadas, la relación entre ellas, los atributos dentro de cada clase y las funciones declaradas en cada una de ellas.

Los scripts *engine.py* y *model.py* son una adaptación del notebook *engine_functions.ipynb* y el script data_functions.py que se pueden encontrar bajo la ubicación '*engine/*'

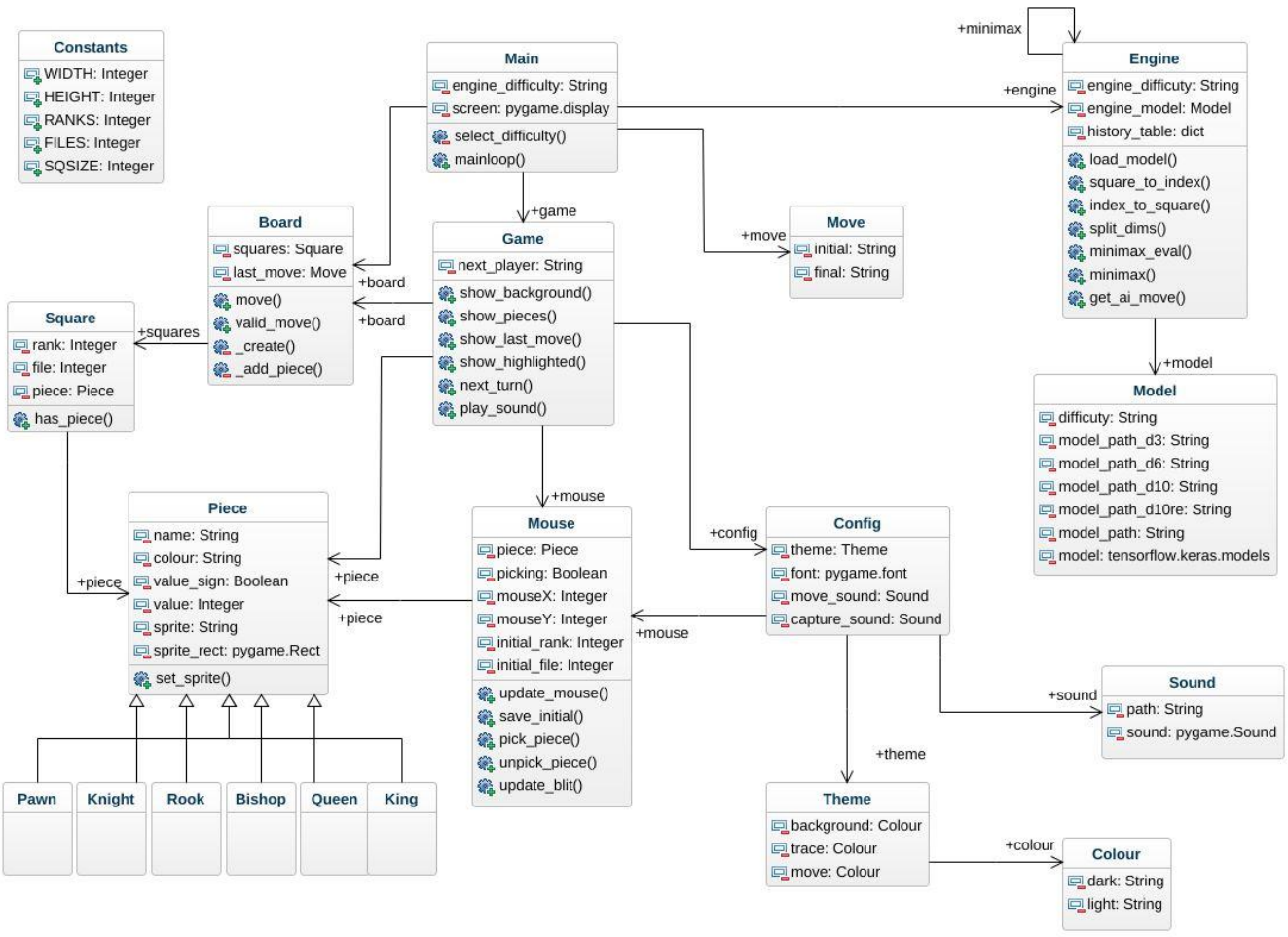

Imagen 7: Diagrama de clases pertenecientes a la interfaz gráfica

Para la interfaz gráfica se han desarrollado 10 scripts de *python* que se pueden encontrar en la ubicación *'chess_game/gui'* y que contienen 18 clases que componen la interfaz al completo.

Los recursos multimedia utilizados como las imágenes o los sonidos empleados pueden encontrarse en la ubicación *'chess_game/gui/resources'*

7. Puntos de mejora

Una vez terminado el proyecto se han detectado algunos puntos sobre los que se podría mejorar en siguientes iteraciones.

7.1 Mejora de las técnicas de selección/evaluación de posiciones para aumentar la velocidad de cálculo.

- Aumentar la profundidad de búsqueda del algoritmo minimax: A medida que aumenta la profundidad de búsqueda, la precisión de la evaluación de posiciones mejora, pero también aumenta el tiempo de ejecución.

- Emplear mejores algoritmos de búsqueda como por ejemplo el Monte Carlo tree search.

- Uso de técnicas de poda adicionales: Además de Alpha-Beta pruning, otras técnicas de poda pueden ser implementadas para reducir la cantidad de nodos que se deben evaluar en la búsqueda.

- Mejora de la función de evaluación: Una función de evaluación eficiente puede ayudar a reducir el tiempo de búsqueda, ya que permite una selección temprana de las mejores opciones. Se podría desarrollar un *dataset* más grande o con más profundidad y entrenar un modelo con esos datos.

7.2 Exploración de distintas alternativas a la clásica red neuronal evaluadora

Como punto de mejora ideal, es posible utilizar otras técnicas como por ejemplo algoritmos basados en aprendizaje por refuerzo.

En este enfoque, el motor juega muchas partidas contra sí mismo o contra otros motores y evalúa el resultado de cada juego. En función de esa evaluación, ajusta su estrategia y tácticas, en un proceso de aprendizaje continuo. Este proceso se lleva a cabo mediante el uso de una función de recompensa, que le da al motor una puntuación o valoración por cada juego ganado o perdido. El motor utiliza esta información para actualizar sus parámetros, como la evaluación de las posiciones o la importancia de diferentes tácticas, y para mejorar su desempeño en el futuro.

El problema de este enfoque es la capacidad computacional requerida y la dificultad exponencial para ajustar el modelo adecuadamente.

7.3 Extracción de datos más naturales de jugadores humanos

Las posiciones evaluadas son artificiales pues ningún jugador que merezca la pena ser estudiado juega aleatoriamente. Una mejora clara incluiría obtener datos de las posiciones más comunes alcanzadas de manera orgánica un mayor número de veces y entrenar los modelos con ellas. Esto implicaría obtener enormes cantidades de datos evaluados de partidas de jugadores y clasificar por frecuencia las posiciones alcanzadas durante las partidas. Seleccionar el top 1 millón de posiciones y evaluarlas. A partir del dataset obtenido se entrenaría al modelo de igual forma que se ha hecho en el proyecto.

7.4 Inclusión de un libro de aperturas

Los primeros movimientos en ajedrez están estudiados en profundidad. Una mejora podría ser incluir un libro con las aperturas más comunes para no tener que evaluar las posiciones durante los primeros movimientos de una partida y poder agilizar el proceso de juego al menos en los primeros movimientos.