

# Detección y Clasificación Automática de Tableros de Ajedrez para la Generación de Notación FEN mediante Redes Neuronales Convolucionales

Pablo Gálvez Ortigosa

pablogalor@correo.ugr.es

Eduardo Rodríguez Cao

edurodcao@correo.ugr.es

José Luís Mera Cardoso

jlmera@correo.ugr.es

Mario Megías Mateo

mariomeg@correo.ugr.es

## Abstract

*En este trabajo abordamos el problema de identificar la configuración que siguen las piezas de ajedrez de un tablero (en notación FEN) a partir de una imagen del mismo. A diferencia de otras aproximaciones, en las que se intentaba entrenar un modelo que realizaba todo el proceso, proponemos un modelo que combina técnicas tradicionales de visión por computador con deep learning, con tres etapas: detección del tablero, detección de casillas ocupadas y clasificador de piezas. Entrenaremos y evaluaremos estas componentes utilizando un dataset 3-D generado por Blender con casi 5000 tableros. Los resultados finales son excelentes, consiguiendo predecir todos los tableros del conjunto de test con un error o menos, y obteniendo un número medio de casillas incorrectas por tablero de 0.02.*

## 1. Introducción

En este proyecto, intentamos construir un sistema de visión por computador que permite, a partir de una imagen de una posición física de ajedrez, identificar la configuración del tablero utilizando la notación FEN (Forsyth-Edwards Notation). La notación FEN es una forma compacta de describir la posición de las piezas en un tablero de ajedrez, representando cada pieza por su inicial en inglés (mayúsculas para las piezas blancas y minúsculas para las piezas negras). Existen numerosas aplicaciones que requieren este tipo de sistemas: robots de ajedrez, software de registro de movimientos de ajedrez y análisis de partidas de ajedrez. Actualmente, muchos de estos sistemas requieren introducir, a mano, pieza por pieza en el sistema. Un modelo como el propuesto permite automatizar todo el laborioso proceso. Para lograr este objetivo, proponemos un modelo que internamente está dividido en tres etapas. Primero, se localiza el tablero de juego utilizando un algoritmo basado en RANSAC, calculando una transformación proyectiva u homografía del tablero hacia hacia una malla, obteniendo el

tablero visto desde arriba. Para ello, se utilizarán técnicas geométricas tradicionales de visión por computador. En las siguientes etapas, se empleará una red neuronal convolucional para predecir la ocupación de las casillas en la imagen deformada y, finalmente, empleando otra red neuronal convolucional, se clasificarán las piezas de las casillas ocupadas. Entrenaremos los tres modelos utilizando un dataset 3D generado utilizando Blender con casi 5000 posiciones válidas de ajedrez. En las dos últimas etapas, compararemos el rendimiento de una arquitectura más simple (con 3 capas convolucionales) con ResNet18. Aunque ambos modelos obtienen buenos resultados, comprobamos que ResNet es ligeramente superior.

## 2. Estado del arte

Destacamos que este trabajo se centra en la generación de notación FEN a partir de un tablero físico, y no de capturas de pantalla de juegos de ajedrez online. Cada uno de estos problemas se aborda de una forma diferente, ya que en el ajedrez online la geometría de los tableros sigue un estándar bastante extendido, mientras que en nuestro caso tenemos que lidiar con una representación física del tablero, en la cual existe más ruido debido a los cambios de iluminación, de orientación, etc, como se muestra en [8, 9, 16]. La conversión a FEN a partir de imágenes de tableros de ajedrez se ha abordado con propuestas que datan del 2012, cuando Bennet et al. [3] utilizaron algoritmos matemáticos para diseñar un modelo que extrae los bordes de las casillas del tablero. Existen nuevos enfoques que usan redes neuronales convolucionales para entrenar un modelo end-to-end que se encargue tanto del reconocimiento del tablero como de la clasificación de las piezas para la generación de la notación FEN [6, 13]. Sin embargo, la propuesta de Wölflein et al. [18] de un modelo de tres etapas (detección de casillas, determinación de la ocupación y clasificación de piezas) y que tratamos aquí, se diferencia con el resto en que usa modelos open-source y ajusta cada etapa por separado antes de integrarlas en un único modelo final. Esta

separación en tres etapas, en lugar de directamente clasificar las piezas tras detectar las esquinas del tablero, permite reducir en gran medida el número de falsos positivos, es decir, casillas vacías para las que se predice una pieza. Para cada una de las etapas, se han adoptado diferentes enfoques en la literatura. Para la detección de casillas, Czyzewski et al. [6] utilizan un detector de líneas rectas y puntos de red para segmentar el tablero de ajedrez en sus 64 casillas. Sin embargo, este enfoque en concreto puede verse limitado por cambios en la iluminación y en el ángulo de visión, y generalmente son lentos [6]. Se han realizado intentos para mejorar estos aspectos, como los realizados por Abeles en [1] y por Chen et al. en [5]. La mayoría de las propuestas utilizan alguna variación de un detector de líneas basado en la transformada de Hough o algún tipo de detector de esquinas, como Harris [2]. En general, los algoritmos de detección de esquinas son incapaces de detectar de forma precisa las esquinas del tablero, debido a que en muchas ocasiones algunas piezas las tapan. Es por ello que la solución preferible es usar un detector de líneas. El problema de la detección de la ocupación se plantea como un problema de clasificación a partir de una casilla de ajedrez recortada, permitiendo el uso de modelos como ResNet18, que ha demostrado un buen rendimiento en ImageNet [7, 11]. La clasificación de las piezas se formula también como un problema de clasificación, con una clase para cada combinación pieza-color, por lo que se pueden usar las mismas arquitecturas utilizadas en la detección de objetos. Existen ejemplos de modelos de dos y tres etapas que han tenido éxito en este problema, como se expone en Quintana et al. [15]. Sin embargo, Wölflein propone un método novedoso, ya que esos modelos previos no usan arquitecturas de deep learning en ninguna de sus etapas. Otra de las diferencias clave es que algunos de los enfoques actuales intentan predecir el estado más probable del tablero teniendo conocimiento previo sobre cómo pueden colocarse las piezas y sobre cuáles son las configuraciones de tableros más populares [6]. No obstante, en la propuesta de Wölflein no se utiliza información previa para la toma de decisiones del modelo, confiando únicamente en la comprensión del propio modelo sobre las ubicaciones de las casillas y el reconocimiento de piezas, lo cual demuestra su versatilidad para poder aplicarse en entornos reales.

### 3. Dataset

Hemos hecho uso del dataset llamado “Rendered Chess Game State Images”, que podemos encontrar en la web de Open Science Framework [17]. Este dataset cuenta con un total de 4888 imágenes con formato 1200 x 800 y se dividen en un 90% de entrenamiento, 3% de validación y 7% de test. Cada una de las imágenes recrea un escenario de juego real en partidas de Magnus Carlsen haciendo uso del software Blender. Una gran ventaja que presenta este dataset es que

ha sido creado haciendo uso de distintas orientaciones de cámara e iluminación con tal de aportar una mayor robustez al modelo. Cada una de las imágenes de tableros cuentan con un JSON asociado en el que se indican algunos datos importantes como puede ser la orientación de la cámara, la notación FEN, coordenadas del tablero y además información de las piezas. Esta información se da como una etiqueta en la que se indica el tipo de pieza, el color, la casilla en la que se encuentra y una tupla de datos que representa la bounding box de la pieza 1.

En total tenemos 104893 casillas ocupadas y 207939 libres. Respecto al número de piezas, la más frecuente de todas es el peón negro que aparece un total de 27076 veces y la menos es la reina negra que está presente en 3133 ocasiones.

Otro detalle importante a explicar es la diferencia que hay en los recortes realizados en el apartado encargado de detectar si una casilla está ocupada y en la clasificación de piezas. Para la tarea de ocupación, el modelo necesita saber simplemente si la casilla está ocupada por lo que con un simple recorte de la casilla será suficiente (partiendo de la vista aérea obtenida en el detector del tablero). Como se puede observar en la Figura 3, el recorte no se realiza de forma ajustada, sino dejando cierto margen en cada lado de la casilla. De esta forma, se proporciona cierta información contextual, y experimentalmente, se obtienen mejores resultados. Sin embargo, en la clasificación de piezas este recorte no es suficiente, ya que muchas piezas no aparecerían completas en las imágenes. Consideraremos por ejemplo el caso del rey y de la reina, en las que recortar la zona de arriba de la pieza (la corona) las haría prácticamente indistinguibles. Debido a ello, cuando nos encargamos de la tarea de clasificación, hacemos uso de una heurística que aumenta la bounding box de la pieza tanto en vertical como en horizontal dependiendo de la casilla en la que se encuentre colocada la pieza. Sumado a esto, con el fin de facilitar al clasificador su tarea, se hace una transformación en la imagen para llevar la casilla en la que se encuentra la pieza a clasificar al extremo inferior izquierdo para eliminar cualquier posible duda a la hora de qué pieza clasificar sobre todo en imágenes en las que la bounding box de la pieza sea muy alargada. Un ejemplo de las imágenes usadas en el clasificador de piezas es la Figura 4.

### 4. Métodos

El modelo propuesto por Wölflein [18] y que proponemos nosotros tiene tres etapas: detección del tablero, detección de casillas ocupadas y clasificador de piezas. Vamos a describir cada etapa detalladamente, visualizando todo el proceso con el siguiente ejemplo de tablero 6.



Figure 1. Tablero con la visualización de los datos del JSON

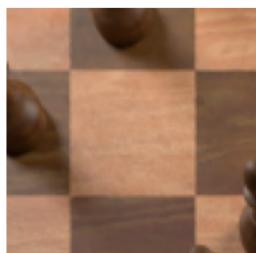


Figure 2. Ejemplo de casilla vacía

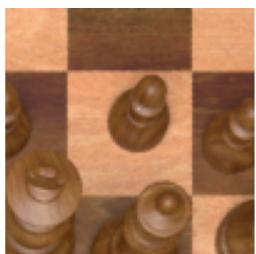


Figure 3. Ejemplo casilla ocupada



Figure 4. Ejemplo de alfil negro



Figure 5. Ejemplo de peón blanco



Figure 6. Ejemplo de tablero que usamos

#### 4.1. Detector del tablero

El objetivo del detector del tablero es encontrar las cuatro esquinas de la cuadrícula del tablero. Una vez que las tenemos, obtenemos las casillas de inmediato con una homografía de la cuadrícula a una malla  $8 \times 8$ . Para encontrar las esquinas, nos centramos primero en la detección de las 9 rectas verticales y las 9 rectas horizontales que subdividen la cuadrícula en casillas. Para eso empezamos aplicando el detector de bordes **Canny**. Como podemos ver, Canny detecta bastante bien los bordes, aunque está claro que algunas rectas no se detectarán bien porque aparecen troceadas.

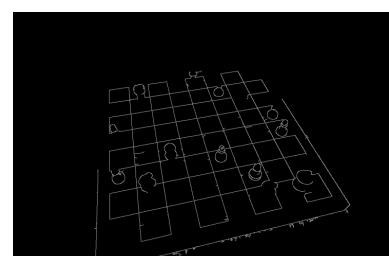


Figure 7. Detector de bordes Canny

Ahora aplicamos **transformada de Hough** para detectar líneas rectas. Como la parametrización de la recta  $y = mx + c$  tiene el problema de que la pendiente  $m$  toma valores reales, trabajaremos con las rectas en forma normal de Hesse, es decir parametrizadas como  $\rho = x \cos \theta + y \sin \theta$ .

con  $(\rho, \theta)$  parámetros que representan la distancia al origen y el ángulo del vector normal con el eje  $X$  respectivamente. Podemos ver como Hough consigue detectar muchas rectas, tanto horizontales como verticales, pero desgraciadamente también detecta rectas que no pertenecen a la cuadrícula, como las rectas que delimitan el propio tablero 8. Este problema se solucionará más adelante.

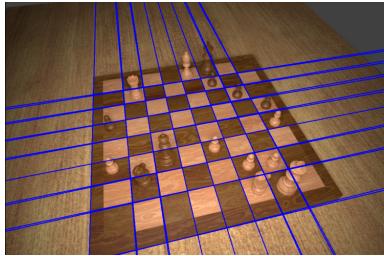


Figure 8. Rectas detectadas por la transformada de Hough

El siguiente paso será distinguir cuáles, de las rectas identificadas, son verticales y cuáles horizontales. Debido a los ángulos tan diferentes de la cámara en las distintas imágenes, establecer umbrales para las direcciones de las distintas rectas no es suficiente para clasificar las líneas de forma robusta. Es por ello que utilizaremos un algoritmo de **clustering aglomerativo**, utilizando como métrica de distancia el menor ángulo entre cada dos rectas. Este algoritmo nos permitirá crear una estructura jerárquica de abajo a arriba, de manera que cada línea comenzará en su propio clúster y se irán uniendo clústers de manera que se minimice la varianza intra-clúster. Por último, el ángulo medio de cada clúster de líneas con el eje Y determinará qué clúster corresponde a las líneas horizontales y cuál a las verticales.

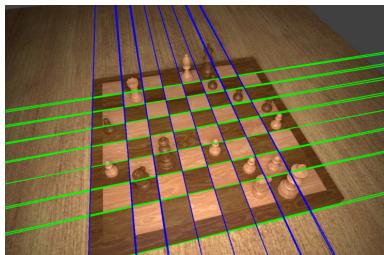


Figure 9. Distinción entre rectas horizontales y verticales (verdes y azules) con clústering aglomerativo

La mayoría de las líneas detectadas son muy similares entre ellas. Para eliminar líneas horizontales similares utilizaremos otro algoritmo de clústering, en este caso, **DBSCAN**. En primer lugar, determinaremos la línea vertical media del clúster vertical. Calcularemos entonces los puntos de intersección del clúster horizontal con esta recta. No sabemos con exactitud ni el número de clústers ni la forma de los clústers, y además todos los clústers están bien sepa-

rados por regiones de baja densidad (en las zonas de intersección hay muchos puntos alejados de la siguiente zona de intersección), por lo que es un caso perfecto para aplicar DBSCAN. El algoritmo de clústering agrupará las líneas similares basándose en estos puntos de intersección. Una vez tenemos los clústers, elegiremos una línea representante de cada clúster. En concreto, nos quedaremos con la mediana (ordenando las rectas según  $\rho$ ). Para eliminar líneas verticales similares se realizará el mismo proceso, pero usando la línea horizontal media del clúster horizontal y calculando los puntos de intersección con el clúster vertical. Con las líneas obtenidas, calculamos los puntos de intersección.

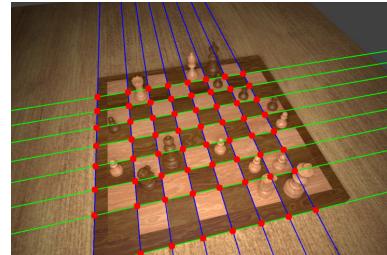


Figure 10. Eliminación de líneas similares con DBSCAN y cálculo de puntos de intersección (en rojo)

Una vez llegados a este punto, en la última imagen podemos observar que detectamos un total de 8 líneas tanto horizontales como verticales, cuando deberíamos detectar 9. También tenemos el problema de que detectamos líneas que no pertenecen a la cuadrícula, en el caso de nuestro ejemplo, detectamos rectas del borde del tablero físico. Para solucionar el primer problema, tenemos que determinar si las líneas a añadir van a estar arriba o abajo (si son horizontales) o bien a la derecha o a la izquierda si son verticales. Para ello, en lugar de buscar las líneas candidatas en la imagen original, lo que hacemos es una transformación de la imagen para que los puntos de intersección formen una malla de puntos que coincida con la vista del tablero desde arriba. Esto lo vamos a conseguir calculando una matriz de homografía  $H$ , que vamos a obtener mediante un algoritmo basado en **RANSAC**, que es robusto tanto a la falta de líneas como a la presencia de rectas outliers. Dicho algoritmo consiste en los siguientes pasos:

1. Se muestran de forma aleatoria cuatro puntos de intersección que se encuentren en dos líneas horizontales y verticales distintas. Estos puntos describen un rectángulo en el tablero de ajedrez.
2. Se calcula la matriz de homografía  $H$  que mapea estos cuatro puntos a un rectángulo de ancho  $s_x = 1$  y altura  $s_y = 1$ . Aquí,  $s_x$  y  $s_y$  son los factores de escala horizontal y vertical.
3. Se proyectan todos los demás puntos de intersección

usando  $H$  y se contabiliza el número de inliers, que son los puntos explicados por la homografía salvo una pequeña tolerancia  $\gamma$  (es decir, la distancia euclídea desde un punto proyectado  $(x, y)$  hasta el punto  $(\text{round}(x), \text{round}(y))$  es menor que  $\gamma$ ).

4. Si el tamaño del conjunto de inliers es mayor que el de la iteración anterior, se guarda este conjunto de inliers y la matriz de homografía  $H$  en su lugar.
5. Se repite desde el paso 1 para  $s_x = 2, 3, \dots, 8$  y  $s_y = 2, 3, \dots, 8$  con el objetivo de determinar cuántos casillas de ajedrez abarca el rectángulo seleccionado.
6. Se repite desde el paso 1 hasta que al menos la mitad de los puntos de intersección sean inliers.
7. Se vuelve a calcular la solución con el menor error cuadrático usando la matriz de homografía  $H$  con todos los inliers identificados.

De esta forma, RANSAC no solo nos calcula la mejor homografía, sino que además nos ayuda a localizar los outliers, resolviendo el problema de rectas incorrectamente detectadas. Como ejemplo, vamos a visualizar una iteración de RANSAC con un muestreo malo y el resultado final del bucle con el muestreo bueno. Como podemos observar en la figura 11, 2 de los 4 puntos muestreados se encuentran en una recta fuera de la cuadrícula, de modo que la homografía calculada obtiene 0 inliers.

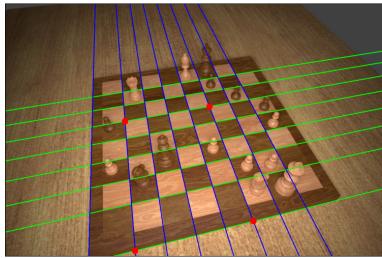


Figure 11. Ejemplo de muestreo malo con 0 inliers

En cambio, en 12 tenemos los 4 puntos muestreadados pertenecientes a la cuadrícula, de modo que la homografía calculada obtiene 49 inliers, que en nuestro ejemplo es el máximo posible.

Disponiendo ya de la matriz de homografía  $H$  correcta, el siguiente paso a tomar es la transformación de la imagen y de los puntos de intersección usando dicha  $H$ . Los puntos de intersección se calculan como números enteros porque tras transformar la imagen, cada casilla tiene una longitud de una unidad. Llamamos  $x_{min}$  y  $x_{max}$  a los valores extremos de las coordenadas transformadas en el eje horizontal e  $y_{min}$   $y_{max}$  a las del eje vertical. A todo lo que obtenemos en el algoritmo previo lo llamamos como mejor configuración.

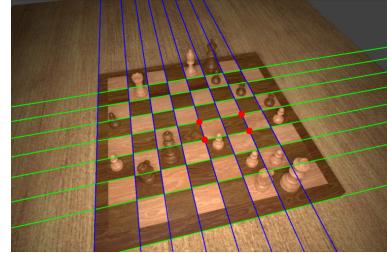


Figure 12. Ejemplo de muestreo bueno con 49 inliers

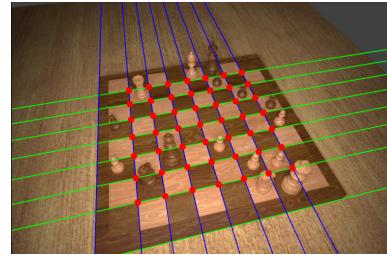


Figure 13. Mejor resultado obtenido por RANSAC (los puntos inliers en rojo)

Finalmente, tras obtener la mejor configuración nos queda por resolver el problema que detectamos al principio: no siempre se detectan todas las rectas.

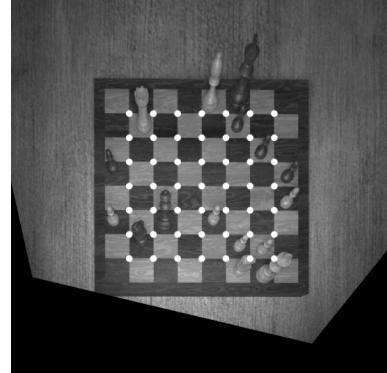


Figure 14. Vista desde arriba tras homografía (con puntos inliers)

Como anteriormente hemos eliminado los outliers, ya no tenemos rectas que no pertenezcan a la cuadrícula, pero sí nos faltan rectas en los bordes de esta. Nuestro objetivo ahora es expandir la cuadrícula que tenemos hasta que sea  $8 \times 8$ . Es decir, queremos que  $x_{max} - x_{min} = y_{max} - y_{min} = 8$ . Es importante subrayar que solo nos quedan rectas por detectar en los bordes, incluso si no hemos detectado al principio una recta dentro de la cuadrícula, esta se obtiene directamente una vez que tengamos las cuatro esquinas. Luego nos centramos en expandir la cuadrícula en las cuatro direcciones posibles hasta que la completemos. Estudiamos el caso de rectas verticales, es lo mismo para

las horizontales:

1.  $x_{max} - x_{min} = 8$ : Tenemos todas las rectas verticales.
2.  $x_{max} - x_{min} > 8$ : Hemos detectado un tablero demasiado grande, algo que podría suceder si se detecta una línea fuera del tablero a una distancia proporcional a las dimensiones de las casillas. Procedemos a quitar rectas por los dos lados hasta quedarnos en el caso 1 o 3.
3.  $x_{max} - x_{min} < 8$ : Nos faltan rectas por detectar en los bordes, para saber en qué dirección ampliar la cuadrícula aplicamos filtro **Sobel** horizontal y después detector de bordes **Canny**. Nos esperamos encontrar las rectas faltantes en la dirección de mayor ruido, luego sumamos las intensidades en bandas centradas en las hipotéticas rectas y comparamos el lado izquierdo y derecho, expandiendo la cuadrícula por el lado que sume más.

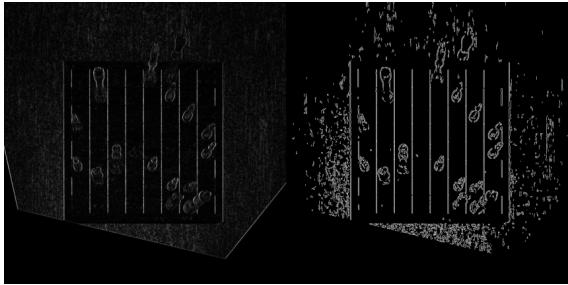


Figure 15. Refinamiento de bordes verticales con Sobel y Canny

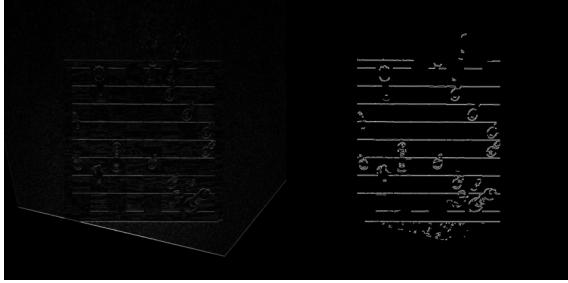


Figure 16. Refinamiento de bordes horizontales con Sobel y Canny

De esta forma obtenemos las rectas faltantes y a partir de esas las cuatro esquinas del tablero que lo determinan completamente. Con esto concluye la etapa de detección del tablero.

## 4.2. Clasificación de casillas por ocupación

Recortamos cada una de las 64 casillas del tablero y las pasamos al clasificador de ocupación que determina si la



Figure 17. Esquinas encontradas en el tablero

casilla está ocupada o no. Para el modelo de clasificación, usamos una CNN básica con 3 capas de convolución, 3 de pooling y 3 densas y una ResNet18 preentrenada con ImageNet. La CNN se entrenará desde cero mientras que con la ResNet18 se hará fine-tuning sobre nuestro dataset. La cabecera final tendrá 2 salidas, una ocupada y otra vacía, y en el caso de la ResNet18 usaremos la cabecera por defecto de FastAI (intentando reducir así el sobreajuste de la red).

## 4.3. Clasificación de casillas por pieza

Si la casilla se detecta como ocupada, el clasificador de piezas predice qué pieza (incluyendo el color) hay en la casilla. Se utilizarán las dos arquitecturas que se usaron para ocupación, adaptando la cabecera final para que la última capa tenga 12 salidas (una por cada pieza y color), y añadiendo cierta regularización en el caso de ResNet (utilizando la cabecera por defecto de FastAI).

Una vez obtenido, para cada casilla del tablero, si está ocupada o no, y en su caso, qué pieza hay en la casilla, se calcula la FEN del tablero.

## 4.4. Detalles técnicos e implementación

Para la implementación de algoritmos de clústering, se han utilizado las librerías de Scikit-Learn [14]. Para la implementación de los diversos algoritmos de Visión por Computador, utilizados en el detector del tablero, hemos usado las librerías de OpenCV [4]. Los parámetros de los algoritmos tales como Canny y Hough se han encontrado haciendo grid search sobre un conjunto de parámetros razonable en un subconjunto pequeño del dataset. El entrenamiento y evaluación de los modelos utilizados tanto en ocupación como en la clasificación de piezas se ha realizado utilizando FastAI [12]. Por comodidad, se ha usado la librería chess de Python [10] para trabajar con los tableros de ajedrez.

## 5. Experimentos

### 5.1. Clasificación de la ocupación

Entrenamos el clasificador de la ocupación en el dataset generado mediante Blender durante 5 épocas y un tamaño de batch de 32. Utilizamos la función de pérdida cross

entropy loss para dos categorías (ocupado o vacío) y el optimizador Adam. Creamos dos modelos de ocupación, cada uno con una arquitectura CNN diferente descrita en la sección de métodos (Convnet de 3 capas y ResNet18 preentrenada en ImageNet). Para la Convnet usamos el learning rate que proporciona el finder de fastai, y 0.01 para ResNet18. El tiempo de entrenamiento para estos dos clasificadores fue de 60 minutos y 1.3 horas respectivamente, usando una GPU NVIDIA T4.

## 5.2. Clasificación de las piezas

De manera similar al detector de ocupación, entrenamos nuestro clasificador de piezas utilizando los datos generados por Blender y los mismos hiperparámetros. Empleamos la función de pérdida cross entropy loss y el optimizador Adam. Los dos clasificadores de piezas son una CNN de 3 capas y ResNet18 preentrenada en ImageNet. Los tiempos de entrenamiento fueron de 22 minutos y 1.13 horas respectivamente, usando una GPU NVIDIA T4.

## 6. Resultados

Para la evaluación del modelo, se han elegido criterios para comprobar el funcionamiento del modelo en cada etapa.

Para la detección del tablero, el archivo JSON correspondiente a cada imagen almacena las coordenadas sobre las esquinas reales del tablero, por lo que podemos medir la distancia en píxeles entre las esquinas predichas y las verdaderas. También usaremos *accuracy*, considerando que el cálculo de esquinas comete un error cuando la diferencia de alguna esquina calculada con la esquina real del tablero es mayor de 1% de la anchura de la imagen.

Los clasificadores tanto de ocupación como de piezas usarán *accuracy* y *F1-weighted*, este último porque el dataset está desbalanceado, en la ocupación hay muchas más casillas vacías que ocupadas, mientras que en las piezas hay más peones que, por ejemplo, reinas.

### 6.1. Detector de tablero

Nuestro detector de tablero evaluado en el conjunto de evaluación obtiene un *accuracy* de 100% y un error en píxeles medio de 1.36 píxeles. Por lo tanto nuestro detector de tablero prácticamente no comete errores.

### 6.2. Clasificador por ocupación

Modelo	Accuracy	F1-weighted
CNN	99.75%	99.75%
Resnet18	100%	100%

Table 1. Evaluación de distintos modelos del clasificador de ocupación en el conjunto de validación

Como podemos ver en Tabla 1, la Resnet18 obtiene resultados prácticamente perfectos y por eso es nuestra elección final, aunque hay que tener en cuenta que la CNN básica también obtiene resultados muy buenos. Evaluada en el conjunto de test, la Resnet18 tiene *accuracy* 99.97% y *F1-weighted* 99.97%, resultado extremadamente bueno. El éxito de los clasificadores de ocupación es debido a que, intuitivamente, parece una tarea bastante simple.

### 6.3. Clasificador de piezas

Modelo	Accuracy	F1-weighted
CNN	95.7%	95.7%
Resnet18	99.94%	99.94%

Table 2. Evaluación de distintos modelos del clasificador de piezas en el conjunto de validación

Como podemos ver en Tabla 2, la Resnet18 obtiene resultados muy buenos y por eso es nuestra elección final, mientras que la CNN básica obtiene resultados que en general se pueden considerar buenos pero lejos de la perfección. Evaluada en el conjunto de test, la Resnet18 tiene *accuracy* 99.97% y *F1-weighted* 99.97%, resultado extremadamente bueno. Aunque esta tarea es más difícil, parece que la ResNet18 sigue obteniendo resultados de calidad.

### 6.4. Pipeline completo

Finalmente, el modelo completo (Detector+Resnet18+Resnet18) se evalúa mediante la comparación de las FENs predichas con las verdaderas configuraciones del tablero (almacenadas en el archivo JSON correspondiente a cada tablero) en el conjunto de validación y de test con las métricas que aparecen en la siguiente Tabla 3. Aquí consideramos como error a una casilla clasificada incorrectamente como ocupada, o una casilla ocupada clasificada como una pieza incorrecta. Para

Métrica	Val	Test
NMCIT	0.01	0.02
Tableros sin errores (%)	99.32	97.66
Tableros con $\leq 1$ errores (%)	100	100
Tasa de error por casilla (%)	0.01	0.04
Accuracy del detector de esquinas (%)	100	100
Accuracy del CO por casilla (%)	100	99.97
Accuracy del CP por casilla (%)	99.99	100

Table 3. Evaluación del pipeline completo en el conjunto de validación y test

acortar NMCIT es el número medio de casillas incorrectas por tablero, CO es el clasificador de ocupación y CP el de piezas. Como podemos ver, el modelo entero obtiene resultados muy buenos, no tenemos tableros con más de 1 error

y casi todos los tableros se han clasificado correctamente tanto en validación como en test, aunque en test el resultado es un poco peor. El NMCIT y la tasa de error por casilla son muy bajos, aunque en test un poco más elevados. También podemos ver qué componente del modelo ha cometido más errores: el detector de bordes es perfecto, en validación el CO es perfecto y el CP comete muy pocos errores, mientras que en test el CP es perfecto y el CO comete más errores, aunque es casi perfecto. Podemos concluir que el modelo obtenido es de alta calidad.

## 7. Conclusiones

El objetivo de nuestro trabajo es automatizar la detección del estado de juego en un tablero de ajedrez a partir de una imagen. Para ello, hemos desarrollado un modelo compuesto por tres etapas principales: detección del tablero, detección de la ocupación de las casillas y clasificación de las piezas en ellas. Nuestra solución combina técnicas tradicionales con métodos de aprendizaje profundo.

En la etapa de detección del tablero, utilizamos técnicas clásicas como el algoritmo de Canny, la transformada de Hough, algoritmos de clustering y homografías para identificar y alinear el tablero en la imagen. Para las etapas de detección de ocupación y clasificación de piezas, entrenamos redes neuronales convolucionales (CNN), incluyendo una CNN básica y una ResNet18, siendo esta última la que obtuvo los mejores resultados.

Finalmente, el modelo completo demostró un excelente desempeño en el conjunto de prueba, clasificando correctamente todos los tableros con un máximo de un error y logrando un promedio de tan solo 0.02 casillas incorrectas por tablero.

Como trabajo futuro, se propone explorar el uso de transfer learning para evaluar cómo nuestro modelo, entrenado en el conjunto de datos sintético generado con Blender, generaliza a ejemplos no vistos previamente. Esto permitiría analizar su capacidad para manejar diversos desafíos, como diferencias en los diseños de las piezas y tableros.

Además, con un mayor volumen de datos y una capacidad de cómputo superior, sería interesante experimentar con arquitecturas avanzadas basadas en transformers, como CLIP o ViT, para comparar su desempeño frente a nuestras CNNs actuales. Estas arquitecturas podrían ofrecer mejoras significativas en la capacidad del modelo para comprender representaciones visuales más complejas.

## References

- [1] Peter Abeles. Pyramidal blur aware x-corner chessboard detector. 2021. [2](#)
- [2] N Banerjee, D Saha, A Singh, and G Sanyal. A simple autonomous chess playing robot for playing chess against any opponent in real time. In *International Conference on Computational Vision and Robotics; Institute for Project Management: Bhubaneshwar, India*, 2012. [2](#)
- [3] Stuart Bennett and Joan Lasenby. Chess - quick and robust detection of chess-board features. 2013. [1](#)
- [4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. [6](#)
- [5] Ben Chen, Caihua Xiong, Quanlin Li, and Zhonghua Wan. Rcdn - robust x-corner detection algorithm based on advanced cnn model. *ArXiv*, abs/2307.03505, 2023. [2](#)
- [6] Maciej A. Czyzewski, Artur Laskowski, and Szymon Wasik. Chessboard and chess piece recognition with the support of neural networks. 2020. [1, 2](#)
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. pages 248–255, 2009. [2](#)
- [8] Jialin Ding. Chessvision : Chess board and piece recognition. 2016. [1](#)
- [9] Xidong Feng, Yicheng Luo, Ziyan Wang, Hongrui Tang, Mengyue Yang, Kun Shao, David Mguni, Yali Du, and Jun Wang. Chessgpt: Bridging policy learning and language modeling. 2023. [1](#)
- [10] Niklas Fiekas. python-chess: A chess library for python, 2017. Accessed on December 25, 2024. [6](#)
- [11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. [2](#)
- [12] Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11(2):108, 2020. [6](#)
- [13] Athanasios Masouris and Jan van Gemert. End-to-end chess recognition. *ArXiv*, abs/2310.04086, 2023. [1](#)
- [14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. [6](#)
- [15] David Mallasén Quintana, Alberto Antonio del Barrio García, and Manuel Prieto Matías. Livechess2fen: a framework for classifying chess pieces based on cnns. 2020. [2](#)
- [16] Soumadeep Saha, Saptarshi Saha, and Utpal Garain. Valued – vision and logical understanding evaluation dataset. 2024. [1](#)
- [17] Georg Wölflein and Ognjen Arandjelović. Dataset of rendered chess game state images. 2021. [2](#)
- [18] Georg Wölflein and Ognjen Arandjelović. Determining chess game state from an image. *Journal of Imaging*, 7(6), 2021. [1, 2](#)