Técnicas de aceleración para el reconocimiento de piezas de ajedrez

Acceleration techniques for chess piece recognition

TRABAJO DE FIN DE GRADO DEL GRADO EN
INGENIERÍA INFORMÁTICA

Curso 2019/2020

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Director: Alberto Antonio del Barrio García
Codirector: Manuel Prieto Matías

David Mallasén Quintana

Madrid, 20 de junio de 2020

Resumen

La digitalización automática de partidas de ajedrez mediante visión artificial es un reto tecnológico significativo. Es imprescindible tanto para los organizadores de torneos como para jugadores amateurs o profesionales de cara a retransmitir en línea o analizar las partidas mediante motores de ajedrez. En este trabajo primero hemos entrenado y comparado diversas redes neuronales convolucionales para la clasificación de piezas de ajedrez. Posteriormente, hemos acelerado sobre una Nvidia Jetson Nano la detección del tablero y la inferencia de estos modelos, necesarios para una completa digitalización. Conseguimos así un framework funcional que digitaliza automáticamente la configuración de un tablero de ajedrez sobre un sistema empotrado en menos de 5 segundos, con una precisión del 92 % al clasificar las piezas y un 95 % al detectar el tablero.

Palabras clave

Aceleración de redes neuronales, Nvidia Jetson Nano, ONNX, TensorRT, Ajedrez, FEN, Visión artificial, Redes neuronales convolucionales, Deep learning, Python.

Abstract

Automatic digitization of chess games by means of computer vision is a significant technological challenge. It is essential both for tournament organizers and amateur or professional players in order to broadcast online or analyze games using chess engines. In this work, we have first trained and compared different convolutional neural networks for chess piece classification. Subsequently, we have accelerated on a Nvidia Jetson Nano the detection of the board and the inference of these models, required for a complete digitization. Thus achieving a functional framework that automatically digitizes the configuration of a chessboard on an embedded system in less than 5 seconds, with an accuracy of 92 % when classifying the pieces and 95 % when detecting the board.

Keywords

Neural network acceleration, Nvidia Jetson Nano, ONNX, TensorRT, Chess, FEN, Computer vision, Convolutional neural networks, Deep learning, Python.

Agradecimientos

En primer lugar, gracias a Alberto y Manuel por pensar en mí para este trabajo y por todo su interés e innumerables correcciones e ideas durante todo este tiempo. Desde las primeras reuniones con un tablero de ajedrez delante, hasta las sesiones virtuales de los últimos meses.

Gracias también a todos los demás profesores que durante estos años habéis hecho que descubra mi vocación y que quiera convertir esta pasión en mi profesión.

Quería agradecer también todo el apoyo desde siempre por parte de mi familia. En este caso especialmente el inconformismo de mi madre, que me empujó e hizo que no haya estudiado "solo" matemáticas.

No querría olvidarme de todos mis amigos, estéis más cerca o más lejos, de todos aquellos con los que he compartido techo e inquietudes, compañeros de clase y personas en general con las que me he cruzado estos años. Sin vosotros no sería ni la mitad de feliz de lo que soy ahora.

Este trabajo ha sido financiado por la Unión Europea (FEDER), el Gobierno de España y la Comunidad de Madrid a través de los proyectos RTI2018-093684-B-I00 y S2018/TCS-4423.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1
Introducción

El reconocimiento de piezas y tableros de ajedrez es un problema de visión artificial que aún no se ha resuelto de manera eficiente. Sin embargo, su solución es crucial para muchos jugadores experimentados que desean competir contra motores de ajedrez y programas especializados, pero que también prefieren tomar decisiones usando un tablero de ajedrez físico. Además, es importante para los organizadores de torneos de ajedrez que desean digitalizar el juego para la retransmisión en línea o para los jugadores aficionados que desean compartir sus partidas con amigos. Por lo general, estas tareas de digitalización son realizadas por humanos o con la ayuda de tableros de ajedrez y piezas especializadas.

Para conseguir digitalizar una partida de forma automática, es necesario ser capaz de reconocer tanto el tablero de ajedrez como posteriormente las posiciones de las piezas. En el escenario de las partidas en vivo, además de la precisión es crítico minimizar el tiempo necesario para realizar dicho reconocimiento, ya que las jugadas pueden sucederse muy velozmente. Este es el caso de las modalidades de juego conocidas como "Blitz" y "Bullet", en las que cada jugador tiene entre 1 y 5 minutos para jugar toda la partida. También ocurre esto mismo en las partidas clásicas cuando queda poco tiempo.

1.1. Antecedentes

Una solución hardware para la digitalización automática de las partidas son los tableros especializados que detectan las piezas físicamente. Ejemplos recientes que se usan actualmente en grandes torneos los podemos encontrar en [Squ] o [DGTa]. Sin embargo, estos tableros son costosos y difícilmente desplegables en muchos ámbitos. A modo de ejemplo, un set de tablero y piezas de DGT (marca usada en torneos oficiales) cuesta desde unos 500€ hasta más de 1000€ [DGTb].

Otras alternativas son las proporcionadas por robots que mueven las piezas de un tablero, como pueden ser [Mat+11] o más recientemente [CW19]. Estos robots se basan en posicionar una cámara cenital sobre el tablero y detectar los diferenciales entre un movimiento y el siguiente. Un inconveniente de esto es que se necesita partir de un estado inicial conocido, no se podría digitalizar de esta forma un tablero con una configuración de piezas genérica. Además, habría que tener en cuenta los fallos, porque se podrían encadenar en cada nueva jugada.

Las soluciones que ofrece la visión artificial son una alternativa a tener en cuenta, ya que proporcionan sistemas más baratos y cada vez más precisos, además de ser un reto tecnológico importante. Como comparación, la plataforma que utilizaremos

para la inferencia, una Nvidia Jetson Nano, cuesta alrededor de 110€ [Nvib] y no se ciñe únicamente a una tarea, podría reutilizarse para otra labor.

Estos procedimientos de visión por ordenador se basan en combinar y adaptar transformaciones y detectores ya conocidos y utilizados en otros ámbitos, como pueden ser el detector de esquinas de Harris y la transformada de Hough [EA10]. Muchos de los métodos suelen asumir grandes simplificaciones, como determinar la posición exacta de la cámara, utilizar tableros diseñados específicamente con marcadores para ayudar en la detección de las esquinas, o directamente mediante la interacción con el usuario [Din16]. Sin embargo, ya existen soluciones genéricas que permiten solventar estas restricciones.

Por ejemplo, hay métodos que permiten clasificar apariciones de varios objetos en lugares arbitrarios utilizando redes neuronales convolucionales (CNNs) [Gao+17], pero no tienen la precisión requerida para obtener su posición exacta. En [Ben+16] se describe un método para detectar objetos que también utiliza CNNs entrenadas mediante supervisión débil. Es decir, basta con tener imágenes etiquetadas de los objetos para entrenar la red, sin necesidad de proporcionar también la posición exacta del objeto en cada imagen de entrenamiento. El problema de esta aproximación es que se necesitan muchas iteraciones para localizar un objeto de forma precisa, lo cual hace que no sea muy práctico en situaciones que requieran respuestas en tiempo real.

En [CLW17] se presenta un método para la detección de tableros que es robusto frente a las condiciones de luz y al ángulo desde el que se toman las imágenes. Además, funciona con la mayoría de estilos de tableros y salva muchas de las debilidades que hemos ido comentando. Es un proceso iterativo en el que se va refinando la posición del tablero en varias fases. Los autores obtienen un 99,5 % de precisión a la hora de detectar las intersecciones de la cuadrícula central del tablero y encuentran su posición completa de forma precisa un 95 % de las veces.

En cuanto a la clasificación de las piezas una vez localizado el tablero, en [Din16] se propone un método que utiliza una máquina de vector soporte (SVM). Esta se entrena a partir de las características extraídas por un descriptor SIFT [Low04], consiguiendo de esta forma una precisión del 85 % a la hora de clasificar las piezas. En [CLW17] afirman conseguir un 95 % de precisión al reconocer las piezas utilizando una red neuronal convolucional (CNN) a la que complementan mediante *clustering* de piezas similares, aprovechando su altura y área, y usando un motor de juego para calcular la probabilidad de configuraciones particulares. El código de estas mejoras no lo han hecho público, así que no hemos podido analizar este enfoque particular.

1.2. Objetivos y plan de trabajo

El procesado de cada fotograma todavía es demasiado lento como para realizar transmisiones en directo. Por tanto, acelerar este cómputo es un reto muy importante. El objetivo final es construir un framework funcional que sea capaz de realizar el

proceso completo de digitalización de una foto de una partida de ajedrez en tiempo real, efectuando todos los cálculos sobre un hardware especializado.

Para ello, partiremos de un método ya implementado para detectar el tablero y sobre el cual realizaremos una serie de cambios. Reordenaremos el código y modificaremos algunos módulos para incluir nuevas posibilidades (véase por ejemplo el apartado 2.2.1 o la comprobación de la posición del tablero en la sección 5.3). Además, optimizaremos el código basándonos en herramientas de análisis de rendimiento y aceleraremos su ejecución sobre nuestro hardware (sección 4.3). Finalmente, construiremos una red neuronal convolucional para clasificar las piezas obtenidas y reduciremos drásticamente el tiempo necesario para la inferencia en el sistema final (sección 4.4).

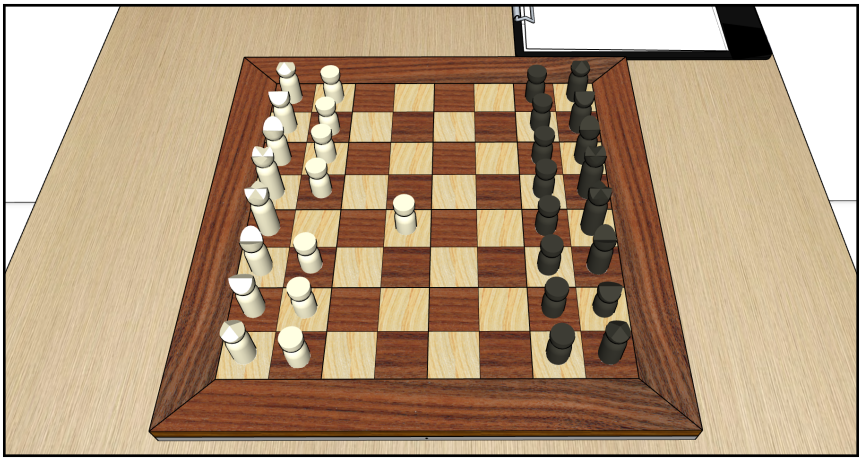


Figura 1.1: Ejemplo de foto tomada por la cámara.

Esto lo realizaremos desde la visión de su uso en el juego aficionado o en torneos, tomando las fotos desde un lateral del tablero cada vez que un jugador pulsa el botón del reloj (Figura 1.1). No obstante, en ocasiones suceden olvidos en los jugadores y no se pulsa el reloj. En estos casos, sería necesario un muestreo periódico en función de la velocidad de procesado de una imagen para tratar de capturar todas las jugadas. Mostramos un esquema de la posición de la cámara en la Figura 1.2.

Figura 1.2: Esquema de la situación de la cámara y del hardware que realizará el cómputo en el uso final.

Capítulo 2
Detección del tablero

Para la detección del tablero nos basamos en el método desarrollado en [CLW17] escrito en Python. En la sección 4.3 veremos las optimizaciones más significativas que hemos realizado y la aceleración sobre el hardware.

Localizar un tablero de ajedrez con piezas ocultando parte de su cuadrícula es un problema de visión artificial complejo. Como además más adelante vamos a querer separar el tablero en sus 64 casillas y de ello dependerá que podamos clasificar las piezas, necesitaremos mucha precisión al localizar las cuatro esquinas. Hacer esto directamente sería muy complejo computacionalmente, así que estos autores proponen un proceso iterativo en el que se va refinando la detección. De esta forma, partiendo de una foto tomada desde las vecindades de un tablero, bastan tres iteraciones para localizarlo con la precisión que necesitamos (Figura 2.1).

2.1. Descripción de las iteraciones

Una iteración del algoritmo se compone de tres pasos, cada uno de los cuales se encarga de identificar nuevos patrones a partir de la información que obtiene de la etapa anterior. El primero de estos pasos, SLID (*Straight LIne Detector*), es la detección de líneas rectas. A continuación, a partir de estas líneas rectas, el siguiente paso, LAPS (*LAttice Points Search*), es la búsqueda de los puntos que puedan formar parte de la cuadrícula del tablero. Finalmente, a partir de estos puntos y las rectas anteriores, el último paso, CPS (*Chessboard Position Search*), se encarga de formar el marco que tenga mayor probabilidad de encuadrar los puntos que forman las esquinas del tablero (Algoritmo 2.1).

Algoritmo 2.1: Descripción de una iteración.

Entrada: Imagen que contiene un tablero.
Salida: Cuatro puntos que encuadran al tablero.

```
1 lineas ← SLID(imagen); // Detección de líneas rectas
2 puntos ← LAPS(imagen, lineas); // Cuadrícula del tablero
3 cuatro_puntos ← CPS(lineas, puntos); // Marco que encuadra al tablero
4 devolver cuatro_puntos
```

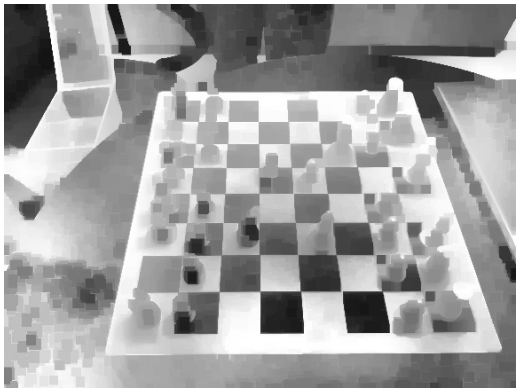
(a) Imagen original.

(b) Resultado tras la primera iteración.

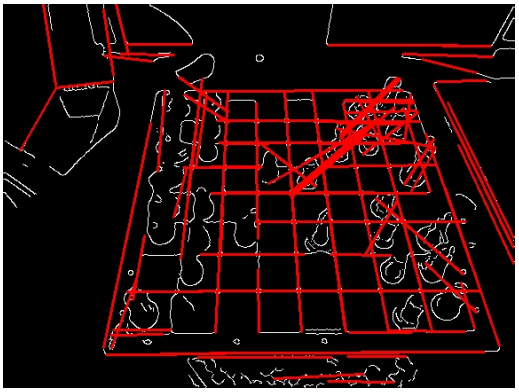
(c) Resultado tras la segunda iteración.

(d) Resultado tras la tercera iteración.

Figura 2.1: Ejemplo de las imágenes obtenidas después de cada iteración.



(a) Imagen transformada.



(b) Segmentos detectados.

Figura 2.2: Ejemplo de dos etapas en la primera fase de detección de líneas.

2.1.1. Detección de líneas rectas

El primer módulo del algoritmo es el encargado de buscar líneas rectas en la imagen inicial y filtrar las que no vayan a necesitarse de cara a encontrar las coordenadas del tablero de ajedrez. También intenta fusionar aquellos segmentos que en realidad formen parte de la misma recta pero que queden separados por la presencia de las piezas (Algoritmo 2.2).

Algoritmo 2.2: Detección de líneas rectas (`SLID`).

Entrada: Imagen que contiene un tablero.
Salida: Conjunto de líneas rectas detectadas en la imagen.

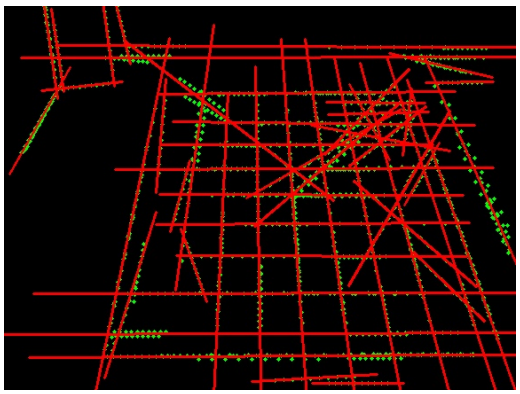
```
1 imagen ← simplificar(imagen);
2 bordes ← canny(imagen);
3 segmentos ← hough(bordes);
4 grupos ← agrupar_colineales(segmentos);
5 // Obtener la línea que mejor se ajuste a cada grupo
6 lineas ← fusionar(grupos);
7 devolver lineas
```

De cara a encontrar líneas en la imagen se utiliza el algoritmo de detección de bordes Canny [Can86] junto con el algoritmo probabilístico de la transformada de Hough para la detección de líneas [MGK00]. Para encontrar el mayor número posible de bordes, se analiza la imagen inicial ajustando los umbrales del operador Canny utilizando el método de umbral de gradiente [SP10]. Además, se aplican varias máscaras CLAHE [Rez04] y simplificaciones para reducir ruido y detalles de la imagen no necesarios en esta fase (Figura 2.2a).

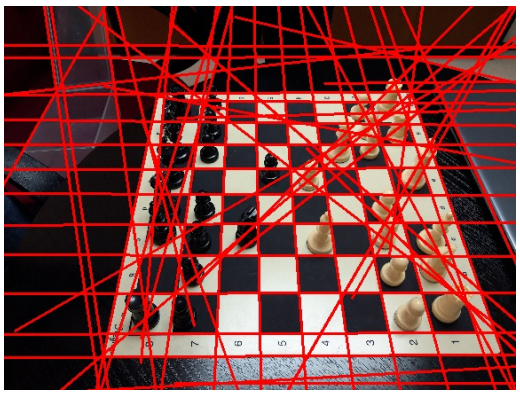
La siguiente acción a realizar es agrupar los segmentos que en realidad formen parte de la misma recta y que se podrán fundir en uno solo. Podemos ver en la Figura

2.2b como varias de las líneas de la cuadrícula del tablero se detectan de forma discontinua como segmentos diferentes. La función que decide si dos segmentos se deben unir o no está descrita en [CLW17]. A la hora de estudiar la colinealidad, tiene en cuenta también la longitud de los segmentos en comparación con el tamaño de la imagen. Esto se debe a que segmentos muy cortos cerca de un segmento muy largo en realidad pueden formar parte de la misma recta, aunque el ángulo que formen entre ellos sea relativamente grande. Conforme crece la longitud, esta restricción sobre el ángulo se vuelve más estricta.

Finalmente, los conjuntos de segmentos colineales obtenidos se fusionan en una única recta. Para ello, primero se convierten los segmentos en puntos, teniendo en cuenta que a mayor longitud más número de puntos, y posteriormente se busca la recta que mejor se ajuste a ellos (Figura 2.3a). Esto último se consigue a partir de un M-estimador para modelos aproximadamente lineales [Wie96] que encuentra dicha recta iterativamente mediante el algoritmo de mínimos cuadrados. Esta aproximación busca solventar algunos de los problemas que presentan los algoritmos que se basan en el estudio de los centroides definidos por los segmentos. De esta forma, el método se ajusta a la intuición de que la recta de mejor ajuste se debe aproximar lo máximo posible a los segmentos más largos del conjunto.



(a) Líneas obtenidas a partir de los puntos formados por los segmentos.



(b) Resultado final de la fase de detección de líneas.

Figura 2.3: Ejemplo de las etapas finales de fase de detección de líneas.

2.1.2. Búsqueda de puntos de la cuadrícula

A partir de las líneas detectadas en el paso anterior, se obtienen todos los puntos de intersección de las mismas. De estos, se selecciona el subconjunto de puntos que tenga una alta probabilidad de formar parte de la cuadrícula formada por las casillas del tablero de ajedrez. Para conseguirlo hay dos vías que se complementan: un primer detector geométrico que se encarga de filtrar positivamente los casos más sencillos y una red neuronal que termina de discriminar los puntos más complicados (Algoritmo 2.3).

Algoritmo 2.3: Búsqueda de puntos de la cuadrícula (`LAPS`).

Entrada: Imagen que contiene un tablero.
Líneas detectadas por `SLID`.

Salida: Conjunto de puntos que forman parte de la cuadrícula del tablero.

```
1  puntos_cuadricula ← [ ];
2  para cada punto ∈ intersecciones(lineas) hacer
3  |  matriz ← preprocesar(vecindad(imagen, punto));
4  |  es_punto_cuadricula ← detector_geometrico(matriz);
5  |  si es_punto_cuadricula entonces
6  |  |  puntos_cuadricula.insertar(punto)
7  |  en otro caso
8  |  |  es_punto_cuadricula ← red_neuronal(matriz);
9  |  |  si es_punto_cuadricula entonces
10 |  |  |  puntos_cuadricula.insertar(punto)
11 |  |  fin
12 |  |  // Si no pertenece a la cuadrícula, pasamos al siguiente
13 |  fin
14 fin
15 devolver puntos_cuadricula
```

Para identificar todas las intersecciones de las líneas obtenidas en el módulo anterior, se utiliza el algoritmo de barrido de Bentley-Ottmann [BO79]. Las vecindades de estos puntos en la imagen serán lo que se estudie para comprobar si efectivamente se tratan de puntos con una alta probabilidad de formar parte de las intersecciones del tablero. Partiendo de una pequeña matriz de píxeles centrada en cada uno de los puntos, se preprocesa para mantener solo la información relevante y, como hemos comentado anteriormente, primero se aplica un detector geométrico para marcar rápidamente como positivos las situaciones más comunes.

Este detector geométrico busca contornos y comprueba si el entorno del punto está formado por cuatro romboides. En caso afirmativo, estamos ante un punto que muy posiblemente forme parte de la cuadrícula del tablero que buscamos. Esto se debe a que los cuatro romboides se corresponderán con las cuatro casillas que forman parte de cada intersección (Figura 2.4a).

En caso de que no se detecte este patrón en los alrededores del punto, el siguiente paso es intentar distinguir si esto se debe, por ejemplo, a que una de las piezas está ocultando parcialmente la intersección (Figura 2.4b). Para ello se utiliza una red neuronal convolucional entrenada sobre un conjunto de casos muy diversos. Así se obtienen las probabilidades de que el punto pertenezca al conjunto con el que seguiremos trabajando en el siguiente módulo. Solo en caso de que el resultado sea positivo con una probabilidad muy alta, se da el punto como válido y se utilizará para identificar el tablero.

(a) Basta el detector geométrico.

(b) Es necesaria la red neuronal.

Figura 2.4: Ejemplo de puntos que sí forman parte de la cuadrícula.

2.1.3. Identificación de la posición del tablero

El paso final de cada iteración es identificar el tablero a partir de los puntos y las líneas encontradas anteriormente. Para ello, el algoritmo analiza los conjuntos de cuatro rectas que formen un cuadrilátero para decidir cuál de ellos es el que se corresponde con el tablero que estamos buscando (Algoritmo 2.4). Nótese que, como los puntos que hemos encontrado son los que forman las intersecciones de la cuadrícula, en realidad no estamos encuadrando el tablero completo, sino el espacio 6×6 formado por las casillas centrales. Esto no será un problema porque al final de la iteración se toma un margen del ancho de una casilla, teniendo en cuenta la perspectiva, alrededor de este marco. De esta forma, en última instancia, será este margen el que se corresponda con los bordes reales del tablero (ver Figura 2.6a).

El algoritmo toma la decisión de qué cuadrilátero es mejor en base a una función de coste que tiene su máximo cuando las cuatro líneas forman perfectamente el marco de un tablero de ajedrez, con la limitación al espacio 6×6 que hemos comentado anteriormente. Esta función *polyscore* [CLW17] está definida como, dado un marco F:

$$P(F) = \frac{L^4}{A_F^2} W_3(k) W_5(l).$$

Donde L es el número de puntos dentro del marco, A_F es el área del marco F, k es la distancia media de los puntos dentro del marco a su borde más cercano, l es la distancia del centroide del grupo de puntos dentro del marco al centroide del marco y W es la función de peso siguiente [1]:

$$W_j(x) = \frac{1}{1 + \sqrt[j]{\frac{x}{L}}}.$$

Intuitivamente, para maximizar $P(F)$ hay que maximizar L y minimizar A_F, k y l.

Para evitar la comprobación de las $\binom{n}{4}$ posibles formas de elegir cuatro rectas, se optimiza el algoritmo para solo revisar algunas de las combinaciones. Esta optimización empieza encontrando el mayor clúster de puntos de la cuadrícula mediante el algoritmo DBSCAN [Est+96] y descarta el resto, ya que este debería representar

[1] Existe una errata en [CLW17] confirmada por los autores. La fórmula está corregida en la implementación y el denominador es efectivamente L en lugar de A_F.

Algoritmo 2.4: Identificación de la posición del tablero (CPS).

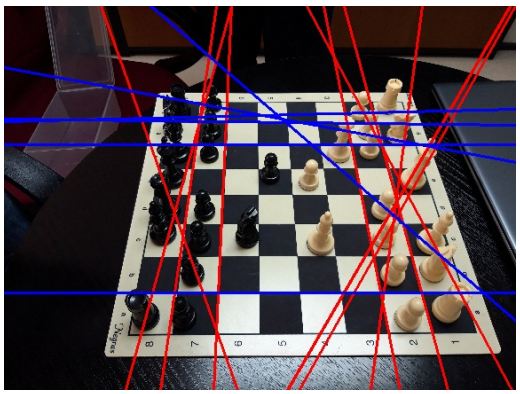
Entrada: Líneas detectadas por SLID.
Puntos de la cuadrícula devueltos por LAPS.

Salida: Coordenadas en la imagen de las cuatro esquinas de la cuadrícula.

```
// Obtener las líneas que bordean al mayor clúster de puntos de la cuadrícula.
clusters ← DBSCAN(puntos_cuadricula);
cluster_tablero ← max(clusters);
lineas_candidatas ← esta_cerca_borde(lineas, cluster_tablero);
// Escoger el par de líneas verticales y horizontales que se ajuste más al borde del tablero
lineas_verticales ← es_vertical(lineas_candidatas);
lineas_horizontales ← es_horizontal(lineas_candidatas);
valor_max ← −∞;
para cada v1, v2 ∈ lineas_verticales, h1, h2 ∈ lineas_horizontales hacer
    valor ← polyscore(v1, v2, h1, h2);
    si valor > max_valor entonces
        max_valor ← valor;
        mejor_marco ← [v1, v2, h1, h2];
    fin
fin
devolver intersecciones(mejor_marco)
```

al tablero que buscamos. A continuación, de todas las líneas que se han obtenido en el primer paso, solo se tienen en cuenta las que estén cerca de alguno de los puntos anteriores, que además no pasen cerca del centroide del clúster y, ayudándose de la función de *polyscore*, comprueba que también estén en las vecindades del borde del tablero.

Finalmente, se dividen las rectas anteriores en horizontales y verticales, teniendo en cuenta la perspectiva. De esta manera, basta con tomar pares de estas rectas verticales y horizontales para elegir el marco que maximiza la función de *polyscore* (Figura 2.5a). Las cuatro rectas obtenidas determinarán el espacio central del tablero y, a partir de esto, se podrá deducir su posición exacta (Figura 2.5b).



(a) Pares de líneas horizontales y verticales que se consideran.

(b) Espacio central del tablero identificado junto con los puntos de la cuadrícula detectados.

Figura 2.5: Ejemplo de las etapas de fase de identificación del tablero.

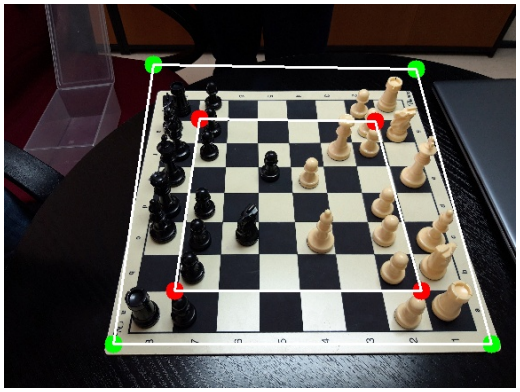
2.1.4. Fin de la iteración

Para pasar del cuadrado 6×6 al tablero completo, se forma un borde que, en última instancia, convergerá a sus límites reales. Esto se consigue añadiendo un margen a una distancia fija. Esta distancia será la correspondiente a una casilla cuando el tablero ocupe la totalidad de la imagen en la última iteración. Como en iteraciones intermedias el tablero ocupa solo un subconjunto del total, este borde siempre será mayor o igual que el necesario, conteniendo al tablero real (Figura 2.6a).

Veamos cómo obtener esta imagen final de la iteración. Esto servirá tanto como inicio de la siguiente iteración, si la hubiera, como imagen recortada del tablero después de transformarlo teniendo en cuenta la perspectiva.

La imagen final de cada iteración será un cuadrado de 1200 píxeles de lado. Así, el objetivo es que los puntos detectados que contienen al tablero (los puntos verdes en la Figura 2.6a) sean los nuevos extremos de la imagen para la siguiente

iteración. Para conseguir esto, se calcula la matriz de la proyección perspectiva que lleva los cuatro puntos que hemos obtenido a los puntos $(0, 0)$, $(1200, 0)$, $(0, 1200)$ y $(1200, 1200)$. A partir de esta matriz, se transformará la imagen inicial, recortada por los cuatro puntos, en el cuadrado que buscamos (Figura 2.6b).



(a) Cuadrado 6×6 central del tablero (rojo) y margen que tomamos (verde).

(b) Imagen después de aplicar la transformación en perspectiva.

Figura 2.6: Ejemplo del final de la primera iteración.

2.2. Obtención de las casillas individuales

Una vez conocida la posición del tablero en la imagen original, tenemos que separarla en las casillas individuales de cara a clasificarlas para obtener el resultado final. La primera aproximación consiste simplemente en dividir la imagen de la última iteración de la detección del tablero en un 8×8. Sin embargo, esto presenta un problema importante que trataremos de resolver mediante un nuevo método.

Como se puede observar en la Figura 2.1, conforme las iteraciones se ajustan al tablero, las piezas que están en los bordes se ven recortadas parcialmente. Este problema es aún más grave en el caso de la fila superior de la imagen, en la que solo se puede ver la base de las piezas según el ángulo con el que se haga la foto. Salvo que la imagen se tome desde un plano cenital lo suficientemente alejado del tablero, lo normal es que cada pieza invada el espacio ocupado por sus casillas vecinas.

Además, por esto último, la parte inferior de cada casilla es la más propensa a contener información de casillas vecinas. Por este motivo, deberíamos fijarnos en más partes de la imagen además de en los límites formados por la propia casilla exclusivamente.

Figura 2.7: Esquinas detectadas en la imagen original.

2.2.1. Nuevo método

Como nuestro objetivo es tomar una foto desde un lateral del tablero, ya que es mucho más sencillo que tener que posicionar una cámara a cierta altura por encima, buscamos diseñar otro método que se adapte mejor a estas circunstancias. Para ello, nos basamos en la idea utilizada en [Din16] para tener en cuenta que las piezas vistas en perspectiva son más altas que el lateral de una casilla.

El problema de este método es que para poder tener en cuenta la totalidad de las piezas, no nos basta con la imagen final del tablero recortado. Es decir, las casillas individuales hay que pasar a obtenerlas de la imagen original. El resultado de las iteraciones al detectar el tablero no nos proporciona las coordenadas de este en la foto inicial, así que tenemos que invertir las transformaciones perspectivas que se realizan al pasar de una iteración a la siguiente. Como ejemplo, en la Figura 2.1 tendríamos que hacer la transformación (d) → (c) → (b) → (a).

Para conseguir esto, nos vamos guardando los cuatro puntos de las esquinas obtenidos al finalizar cada iteración. De esta forma, podremos calcular las matrices de las transformaciones en el orden contrario y bastará con invertir cada una de las matrices obtenidas y multiplicarlas para obtener la función que buscamos. A partir de esto ya tendremos la transformación que nos permitirá calcular las coordenadas en la foto inicial de un punto en la imagen final recortada. Dividiendo el resultado de la última iteración en la cuadrícula 8×8 del tablero, obtendremos las esquinas de cada una de las casillas (Figura 2.7).

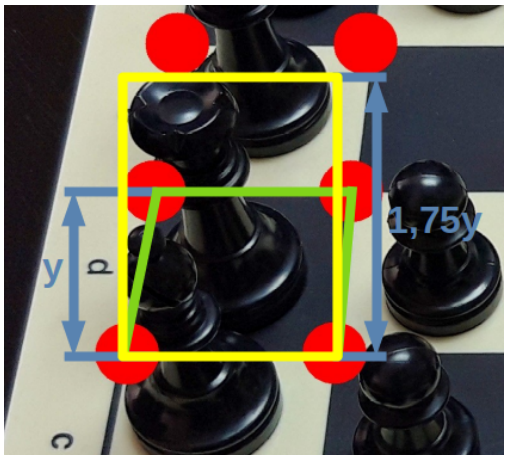
Una vez conocidas las coordenadas de las cuatro esquinas de una casilla, tenemos que calcular el rectángulo por el que recortaremos. Primero calculamos la altura de la imagen como la diferencia entre una esquina superior y una inferior multiplicado por un factor de 1,75. El hecho de tomar una mayor altura más hace que podamos ver mejor la parte de arriba de las piezas, que al fin y al cabo es la que mejor las diferencia, sin llegar a introducir en el rectángulo más que la base de la posible pieza de la casilla superior (Figuras 2.8b y 2.8a). Así, ampliamos el conocimiento de la

zona de la pieza que mejor las diferencia e introducimos el menor ruido posible. En caso de salirnos de la imagen por arriba, ajustamos la altura al máximo. Como estamos asumiendo que las fotos se toman desde un lateral del tablero y no desde una esquina, podemos tomar la base del rectángulo como el ancho de las dos esquinas inferiores (Figura 2.8).

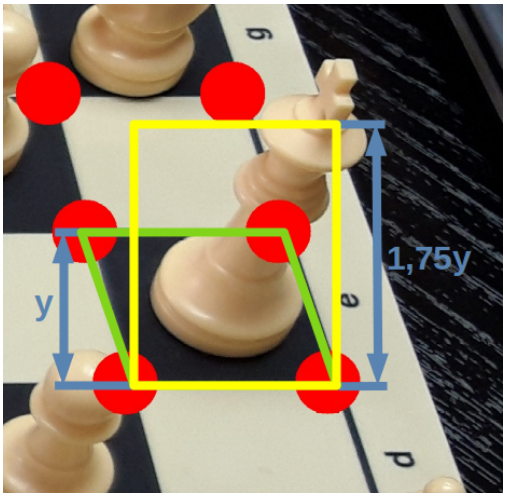
Como se puede observar en la Figura 2.8c, hay que ajustar el factor por el que multiplicamos la altura teniendo en cuenta también que los peones son más bajos y se podrían solapar los unos con los otros. Otra opción sería ajustar la altura según estemos en la parte inferior o superior de la imagen (Figuras 2.8d y 2.8c). Aún así, con la suficiente variedad de casos en el dataset, nuestra red neuronal convolucional que veremos en la sección 3.3 debería ser capaz de aprender en qué zona de la imagen fijarse para diferenciar las piezas más altas de las más bajas.

Veamos una posible generalización de este método en caso de querer también admitir fotos hechas desde las esquinas (en las que el tablero se vería como un rombo). Podríamos tomar una combinación lineal de las alturas de las esquinas de cada casilla de forma que se mantuviese lo comentado en el párrafo anterior cuando el borde inferior de la imagen fuese paralelo al borde inferior del tablero. Una opción sencilla que cumpliría esto es tomar la media de las alturas de las esquinas inferiores para situar la base del rectángulo y ajustar el factor por el que multiplicamos la altura en función de la diferencia entre el ancho y el alto de la casilla.

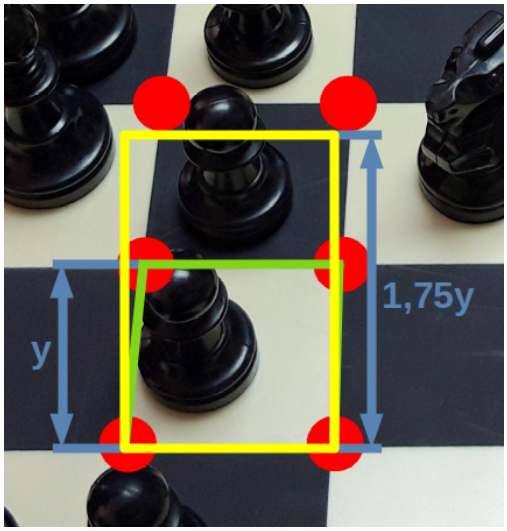
Nuestra idea inicial era obtener una gran cantidad de fotos tomadas a partidas reales en un club de ajedrez, pero desgraciadamente por las circunstancias actuales no hemos podido construir un dataset de piezas obtenidas de esta forma. Por lo tanto, como veremos en la sección 3.2, nos quedaremos con la primera aproximación que hemos visto al inicio de esta sección para recortar las casillas, ya que es la forma con la que se han obtenido los dataset que hemos podido utilizar. Planteamos en el capítulo 6, como trabajo futuro, seguir este estudio para incorporar el método que hemos propuesto.



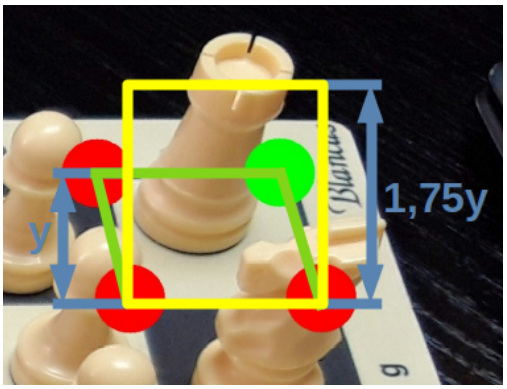
(a) Dama negra.



(b) Rey blanco.



(c) Peón negro.



(d) Torre blanca.

Figura 2.8: Recorte de las casillas mediante la primera aproximación (verde) y con el nuevo método (amarillo). En las dos primeras figuras, las más altas, vemos como con el rectángulo amarillo se llega a poder diferenciar la parte superior de las piezas. En las dos figuras inferiores, las piezas más bajas, vemos la diferencia que puede haber entre que estén situadas en la parte inferior o superior del tablero.

Capítulo 3
Clasificación de las piezas

Una vez detectada la posición del tablero en la imagen original, el siguiente paso es clasificar cada una de sus casillas. Cada casilla puede estar vacía o puede estar ocupada por una pieza de uno de los jugadores, por lo que tendremos que decidir a cuál de las 13 clases anteriores corresponde cada una de las 64 casillas del tablero.

Actualmente, la manera más utilizada y los mejores algoritmos para realizar la clasificación de una imagen en una serie de categorías son mediante el uso de redes neuronales convolucionales [RW17]. Además, estas redes se pueden acelerar enormemente como veremos en el capítulo 4.

3.1. Notación FEN

El resultado final de la clasificación será una cadena de caracteres que codificará las posiciones de las piezas en el tablero mediante la notación de Forsyth-Edwards (FEN) [Edw94]. Como solo tendremos una instantánea de la partida, únicamente podremos saber la posición de las piezas, no el jugador al que le toca mover o si hay posibilidad de enroque por ejemplo (factores que sí que se tienen en cuenta en la notación FEN completa). Así, la cadena será una serie de ocho bloques de caracteres alfanuméricos representando cada fila del tablero separadas por el carácter /.

Las filas se escriben de izquierda a derecha y de arriba a abajo desde la perspectiva de las blancas. Los caracteres se corresponden con las iniciales de los nombres de las piezas en inglés (a excepción del caballo, que se representa con una n), en mayúsculas si la pieza es blanca y en minúsculas si es negra. Las casillas en blanco se abrevian con un número del 1 al 8 indicando el número de casillas vacías contiguas.

La posición inicial de una partida se corresponde por tanto con la siguiente cadena: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR. En la Figura 3.1 podemos ver un ejemplo de una posición más avanzada de la partida.

3.2. Dataset etiquetado de piezas

Obtener un dataset etiquetado de imágenes de tableros o de piezas de ajedrez no es tarea sencilla, ya que no existe ninguno disponible de forma pública que sea completo o variado [Din16] [CLW17]. Usualmente, en estos casos, cada autor se crea su pequeña muestra de piezas con las que poder poner a prueba su idea. Sin embargo, para entrenar un modelo de deep learning son necesarias una gran cantidad

r1bqnrk1/pp1ppBbp/6p1/n3P3/3N4/2N1B3/PPP2PPP/R2QK2R

Figura 3.1: Notación FEN (Fischer - Reshevsky [B35] USA-ch NY (6), 1958).

de imágenes y, si además queremos que funcione para diferentes tipos de piezas, hace falta introducir esta variedad en los datos de entrenamiento.

Como hemos comentado en el apartado 2.2.1, por la situación actual no hemos podido construir un dataset de tableros lo suficientemente grande al que aplicar el método que hemos desarrollado. Para entrenar nuestros modelos con la suficiente variedad, hemos juntado dos datasets etiquetados de piezas [Yan16] [Sch19] que nos han permitido construir un framework funcional con el que poder trabajar. De esta forma nuestro conjunto de imágenes consta de casi 55000 fotos cuadradas de piezas de ajedrez de 150 y 227 píxeles de ancho en formato JPEG. Estas diferencias de tamaño no serán un problema al entrenar nuestra red ya que todas las imágenes serán preprocesadas y estandarizadas.

3.3. Entrenamiento de los modelos

Nuestras redes neuronales convolucionales las definimos y las entrenamos utilizando la API de alto nivel Keras [Cho+15]. De esta forma implementaremos sobre la librería TensorFlow [Aba+15] distintos modelos con los que probar la clasificación. De cara al entrenamiento, podremos hacer uso de las tarjetas gráficas de manera transparente para acortar el tiempo total necesario.

Una técnica muy extendida a la hora de entrenar modelos de aprendizaje profundo es el *transfer learning*. Este método permite adaptar redes ya entrenadas sobre un conjunto de imágenes al dominio con el que estamos trabajando, reduciendo el tiempo necesario para el entrenamiento y requiriendo un dataset más pequeño. Así, podemos entrenar nuestro modelo basándonos en el conocimiento que han aprendido las capas convolucionales de la red para identificar contornos, curvas u otras características de las imágenes. Congelaremos los pesos de estas capas y bastará con

enseñar a las nuevas capas que añadamos sobre el modelo cuáles de estos patrones identifican a cada una de las clases finales, las piezas de ajedrez.

El dataset sobre el que se basan los modelos preentrenados disponibles en Keras es ImageNet [Den+09]. ImageNet se compone de unas 14 millones de imágenes divididas en casi 22000 categorías, de las cuales para estos casos se utilizan las aproximadamente 1,2 millones de imágenes divididas en 1000 categorías que componen el ILSVRC [Rus+15].

Otro recurso muy relacionado con el anterior es el *fine tuning*. Una vez entrenado el modelo sobre nuestro dataset habiendo congelado los pesos de las capas convolucionales, se procede a ir descongelando las últimas capas para que la red se pueda terminar de adaptar a este ámbito. De esta forma podremos terminar de exprimir todas características que se pueden extraer de nuestras imágenes. Este entrenamiento también será más rápido, ya que partiremos de unos pesos en las capas de nuestro modelo que tardarán menos en converger que una inicialización aleatoria.

Para elegir los modelos de Keras que probar con nuestro dataset, nos ayudamos del estudio exhaustivo hecho en [Bia+18]. Este nos resulta especialmente útil ya que los autores prueban también las distintas opciones existentes sobre la Jetson Nano que, como veremos en la sección 4.1, será la plataforma sobre la que ejecutaremos la inferencia del modelo final.

Basándonos en estos resultados y los modelos que están disponibles de forma oficial en Keras, decidimos en primera instancia probar MobileNetV2 [San+18], NASNetMobile [Zop+18], DenseNet201 [Hua+17] y Xception [Cho16]. También observamos que AlexNet [KSH12] y SqueezeNet-v1.1 [Ian+16] obtienen mejores resultados en cuanto a tiempo, aunque son menos precisos sobre ImageNet y no están incluidos de forma nativa en Keras. En caso de que pudiesen aprender bien sobre nuestro dataset y obtener una precisión similar, estos modelos proporcionarían una solución más rápida en teoría. Veremos estos resultados en la sección 4.4.

De cara al entrenamiento hemos utilizado una serie de mecanismos para facilitar la convergencia de los modelos. Para evitar el sobreajuste de la red a nuestro dataset, monitorizamos el avance en la precisión sobre los datos de validación para parar el entrenamiento cuando esta ya no crezca más. Así, el número de épocas no es un valor fijo si no que depende de la evolución de este dato. Además, nos vamos guardando los pesos que mejores resultados proporcionan y reducimos la tasa de aprendizaje de forma dinámica.

En esta fase del desarrollo utilizamos un servidor con 2 CPU Intel Xeon Gold 6138 a 2GHz y 40 cores, 96 GB de memoria RAM y una GPU Nvidia Tesla V100 con 32GB de memoria HBM2. Esta GPU está formada por 5120 núcleos CUDA y 640 núcleos tensoriales (*Tensor Cores*). Los núcleos tensoriales son elementos de cómputo diseñados para acelerar el entrenamiento de redes neuronales, ya que permiten la multiplicación y acumulación de matrices 4×4 de elementos en coma flotante de 16 bits. Esto nos permite reducir enormemente el tiempo necesario para el entrenamiento y las pruebas iniciales de los distintos modelos.

Figura 3.2: Ejemplo de vector de probabilidades para una casilla del tablero.

3.4. Inferencia

Una vez tenemos nuestro modelo entrenado sobre el dataset de piezas, tenemos que utilizarlo para predecir cada casilla de un tablero nuevo. Para este proceso, primero tenemos que cargar las imágenes que hemos recortado de cada casilla y preprocesarlas para que tengan el formato requerido por nuestra red. Una vez terminado esto, se las pasamos como argumento a nuestro modelo y éste nos devuelve un vector de 13 componentes. Como tenemos 13 clases distintas (las piezas de ambos colores y la casilla vacía), cada una de las posiciones del vector tendrá el valor correspondiente a la probabilidad de que nuestra imagen pertenezca a esa clase (Figura 3.2).

A partir de este vector de probabilidades, utilizamos el conocimiento del dominio que tenemos para afinar mejor la configuración de piezas del conjunto del tablero. Introduciremos así reglas que tendrán en cuenta todas las casillas a la vez, al contrario que la clasificación de cada casilla mediante el modelo de aprendizaje profundo, que es agnóstica con lo que sucede a su alrededor.

Primero calculamos qué dos casillas tienen una mayor probabilidad de contener a los reyes de cada bando. Esto lo hacemos para asegurarnos de que solo hay exactamente un rey de cada color en cada configuración. Posteriormente fijamos todas las casillas en blanco ya que los modelos las detectan con una precisión muy alta. Finalmente, clasificaremos el resto de casillas teniendo en cuenta que cada pieza tiene un máximo de apariciones. Además, los dos alfiles de cada jugador, en el caso de que aún conserve ambos, deben estar en casillas de colores diferentes [1].

Veamos a continuación cómo hacemos la clasificación de las piezas que no son reyes. Primero ordenamos todas las casillas según su probabilidad de contener cada una de las 10 clases restantes (en total son 13 pero ya hemos fijado los reyes y las casillas vacías). Así, obtenemos 10 listas de pares de pieza y casilla ordenadas de

[1] En nuestro estudio hemos supuesto que, de haber promociones, estas han sido a una dama. En estos casos se puede elegir cualquier pieza menos un rey u otro peón. Sin embargo, las ocasiones en las que un jugador no elige una dama son poco frecuentes. Esta simplificación se puede eliminar completamente como comentaremos en el capítulo 6 llevando un histórico de toda la partida. En estos casos se podrá saber exactamente el número y tipo de piezas que tendrá cada jugador.

mayor a menor probabilidad de que la casilla contenga la pieza. Una vez tenemos estas ordenaciones, iteramos escogiendo el elemento de la cabeza de la lista que tenga una probabilidad mayor. Si aún no se ha alcanzado el máximo de ese tipo de pieza y la casilla que representa en el tablero aún no se ha rellenado, fijamos la pieza en el tablero. En cualquier caso, la eliminamos de su lista de piezas ordenadas ya que no la tendremos que volver a tratar (Figura 3.3).

En la siguiente iteración escogeremos la pieza que tenga la segunda mayor probabilidad y así sucesivamente. Finalizamos el algoritmo cuando hayamos rellenado todo el tablero. Resumimos estos pasos en el Algoritmo 3.1. De esta forma, como veremos en el apartado 4.4.2, conseguimos aumentar la precisión de la clasificación de las piezas y introducimos coherencia en la configuración final.

3.4.1. Inferencia en tableros consecutivos

En los casos en los que estemos digitalizando jugadas sucesivas de una partida, se pueden implementar mejoras sustanciales de cara a aumentar la precisión de la clasificación de las piezas. En esta sección hemos introducido un algoritmo que incluye información sobre el dominio en la inferencia. De esta manera, ampliábamos la visión a todo el tablero en conjunto en lugar de solo a cada casilla individualmente. Si además tenemos información sobre varias configuraciones consecutivas, se pueden añadir estos datos adicionales para completar y verificar toda la digitalización.

Los modelos de aprendizaje profundo que hemos entrenado son capaces de detectar las casillas vacías sin fallar prácticamente nunca. Así, teniendo dos instantáneas consecutivas, podemos inferir qué pieza se ha movido en función de su casilla inicial y final. Este es el caso del caballo y su movimiento peculiar en forma de L, pero no es el único. Por ejemplo, si una pieza se ha movido dos casillas en diagonal, solo hay dos opciones: o es un alfil o es una dama, ya que los peones y el rey solo pueden moverse en diagonal a casillas vecinas.

Esta misma idea sirve para las situaciones en las que una pieza captura a otra, ya que la precisión al diferenciar los colores de las piezas también es muy alta. Otro escenario que se puede mejorar con esta información son las promociones de piezas. Si sabemos que en el tablero anterior había un peón en la séptima fila y ahora hay otra pieza en la última fila, podemos deducir que ha habido una promoción.

Hemos implementado otro algoritmo que tiene en cuenta también la configuración de la instantánea anterior de cara a inferir el movimiento que se ha realizado. Partiendo de la cadena en notación FEN obtenida de la imagen previa y del vector de probabilidades que resulta de la red neuronal, primero calcula las casillas que han tenido un cambio importante. Estas son las que han pasado de estar vacías a estar ocupadas o viceversa y las que han cambiado de color. Incluyendo esta información se puede inferir la acción que ha ocurrido. Por ejemplo, que una pieza negra haya capturado a una blanca o que una pieza blanca se haya movido.

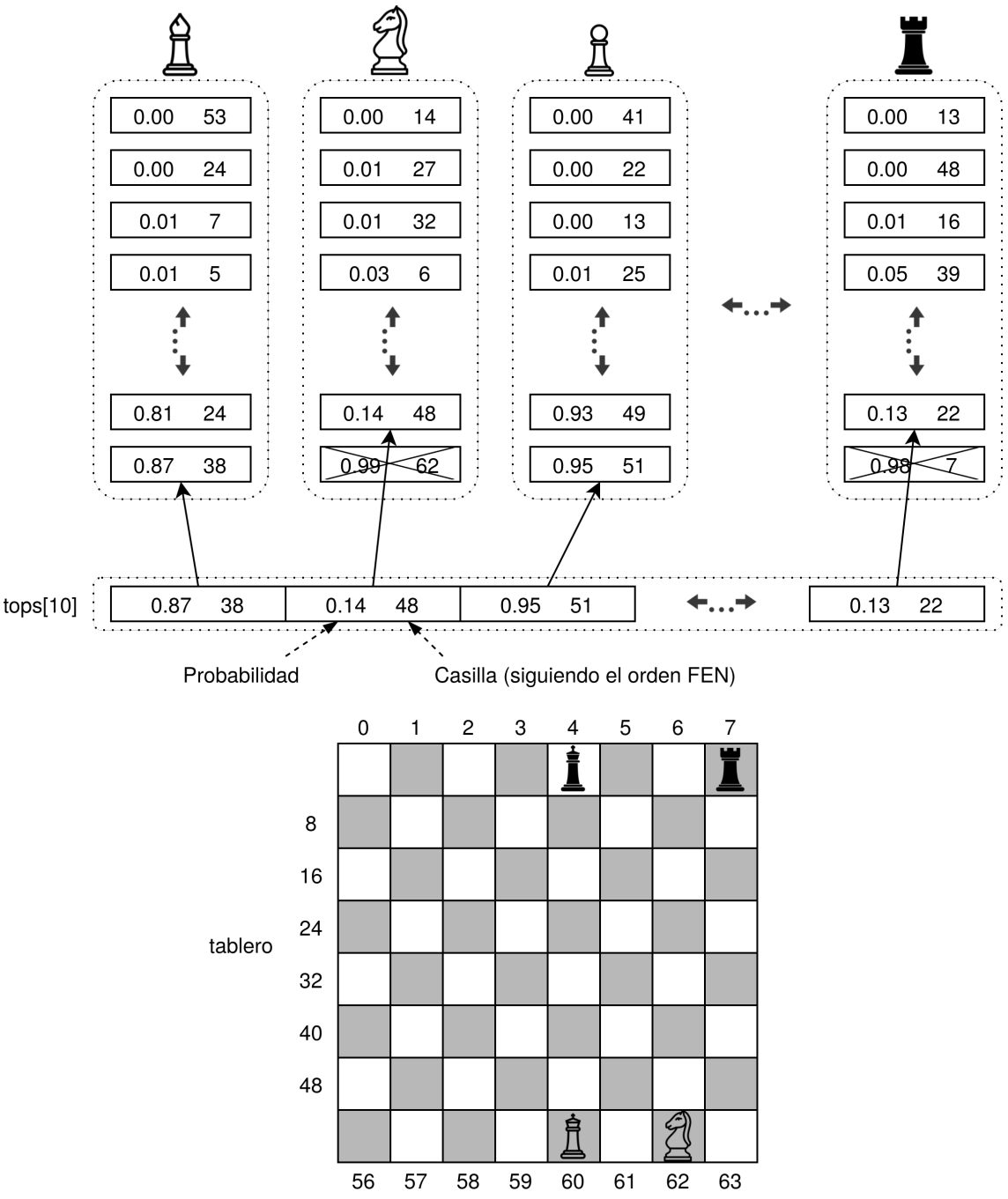


Figura 3.3: Esquema de los datos empleados en la inferencia de la configuración del tablero. Los reyes se fijan al principio del algoritmo, en esta instantánea se han fijado también un caballo blanco (que tenía probabilidad 0,99 de estar en la casilla 62) y una torre negra (que tenía probabilidad 0,98 de estar en la casilla 7.)

Algoritmo 3.1: Cálculo de la configuración del tablero a partir de los vectores de probabilidades de cada casilla.

Entrada: Vectores de probabilidades de cada casilla.
Salida: Configuración del tablero.

```
1  tablero ← [ ] * 64 // Lista vacía de las 64 casillas
2  // Fijamos los reyes, análogo para el rey negro
3  rey_blanco ← max_prob(vectores_probs, 'K');
4  tablero[rey_blanco] ← 'K';
5  ...
6  por_rellenar ← 62;
7  // Fijamos las casillas vacías
8  para cada casilla ∈ vectores_probs hacer
9      si max_prob(casilla) = '_' entonces
10         tablero[casilla] ← '_';
11         por_rellenar ← por_rellenar − 1;
12     fin
13 fin
14 // Ordenamos los vectores de probabilidades obteniendo las listas
       mostradas en la Figura 3.3 y el vector tops
15 ...
16 // Terminamos de rellenar el tablero en el orden dado por las
       probabilidades de las piezas
17 mientras por_rellenar > 0 hacer
18     pieza ← max_prob(tops);
19     si ¬alcanzado_max(pieza, piezas_usadas) ∧ tablero[pieza] = [ ] entonces
20         tablero[pieza] ← pieza;
21         por_rellenar ← por_rellenar − 1;
22         piezas_usadas[pieza] ← piezas_usadas[pieza] + 1;
23     fin
24     // Actualizamos las listas y los punteros de tops
25     tops[pieza] ← ∅;
26     ...
27 fin
28 devolver tablero
```

Sabiendo esto, podemos deducir en función de la casilla inicial y la final qué piezas son compatibles con el movimiento realizado. De esta manera podremos afinar más la pieza que se encuentre en la casilla final del movimiento (Algoritmo 3.2). En las pruebas y resultados que mostramos en los capítulos 4 y 5 deshabilitamos este algoritmo, ya que requiere información que no tendremos al tomar una única instantánea de un tablero. Esto será útil y se podrá ampliar como comentaremos en el capítulo 6 de cara a digitalizar partidas completas.

Algoritmo 3.2: Inferencia del tipo de pieza a partir de su movimiento.

Entrada: Cadena FEN de la imagen anterior.
Vectores de probabilidades de la imagen actual.
Salida: Conjunto de piezas compatibles con el movimiento realizado.

```
1 casillas ← casillas_cambiadas(FEN_anterior, vectores_probs);
2 (casilla_ini, casilla_fin, accion) ← mov_inferido(FEN_anterior, vectores_probs,
3                                                   casillas);
4 piezas_posibles ← piezas_compatibles(casilla_ini, casilla_fin, accion);
5 devolver piezas_posibles
```

Capítulo 4
Aceleración

El paso final y la parte más importante de nuestro trabajo es la aceleración de todo el framework que hemos construido. Desde la detección inicial del tablero hasta la clasificación de las piezas para obtener la digitalización completa. Existen multitud de técnicas para hacer más eficiente la ejecución de un programa, específicamente para la ejecución eficiente de redes neuronales profundas [Sze+17] [Kim+19] [MDB20]. En este capítulo veremos las que hemos utilizado nosotros y cómo las hemos adaptado a este caso concreto.

4.1. Plataformas de inferencia

En base al uso final de nuestro programa, decidimos que lo más apropiado sería su ejecución en local en un sistema empotrado con hardware específico para la aceleración. Existen diversos sistemas que cumplen estas características y se adaptan al cálculo de la inferencia de modelos de deep learning. Tres de ellos serían el Intel Neural Compute Stick 2 [Int], los sistemas Coral de Google [Goo] o la familia Jetson de Nvidia [Nvia].

Los Neural Compute Stick 2 de Intel permiten añadir mediante una conexión USB una unidad de procesamiento de visión (*Vision Processing Unit* o VPU), esto es, un acelerador hardware de bajo consumo dedicado al cálculo de algoritmos de visión artificial, como pueden ser las redes neuronales convolucionales. Los sistemas Coral de Google están compuestos o bien por una unidad USB o bien por una placa dedicada, permitiendo disponer en ambos casos de una unidad de procesamiento tensorial (*Tensor Processing Unit* o TPU) para la aceleración de redes neuronales. La diferencia fundamental de una TPU con respecto a un GPU es el volumen de cálculo que pueden alcanzar, estando las TPU optimizadas para tamaños de *batch* más grandes. Además, están pensadas íntegramente para el cómputo utilizando precisión reducida, de forma similar a los núcleos tensoriales que proporcionan las GPU de Nvidia más recientes.

En nuestro caso vamos a utilizar la Nvidia Jetson Nano [Nvib], el dispositivo más pequeño de la familia Jetson. Está compuesto por una placa dedicada con dos modos de consumo, 5 o 10 W. Unos valores muy reducidos en comparación con su potencia de cómputo, ya que puede llegar a los 472 GFLOPs en FP16. Sus especificaciones técnicas principales son las siguientes:

- Una GPU con arquitectura NVIDIA Maxwell de 128 núcleos CUDA.
- Una CPU ARM de cuatro núcleos Cortex-A57 a 1,43 GHz.

- 4 GB de memoria RAM LPDDR4 a 25,6 GB/s.
- Conectividad Gigabit Ethernet y conexión HDMI y DisplayPort.

Estas características nos ofrecen, además de una GPU dedicada de Nvidia con el potencial de TensorRT para la inferencia (como veremos en la sección 4.2), una CPU capaz de realizar el cómputo secuencial necesario para detectar los tableros. Del mismo modo, hemos optado por usar esta plataforma ya que, al incluir una GPU, permite acelerar también las fases de detección del tablero. Este tipo de arquitectura se viene usando con éxito desde hace más de una década en las tareas relacionadas con el procesamiento de imágenes [Set+07] [Ten+08].

Este sistema forma parte de la familia Tegra X1 "Erista", con una disminución de la frecuencia de la CPU y disponiendo solamente de la mitad de núcleos CUDA que los modelos utilizados en la Nvidia Shield TV [Nvid] o la Nintendo Switch [Nin]. Esta familia fue la primera de Nvidia que se diseñó pensando en dispositivos portátiles, además de ser la primera arquitectura que incluyó cierto soporte para operaciones en punto flotante de 16 bits. En algunas situaciones, como en el caso de que se trate de la misma operación, dos operaciones en FP16 se pueden empaquetar juntas a ejecución sobre un único núcleo CUDA de 32 bits (Figura 4.1) [HS]. Esto último lo utilizaremos de cara a la aceleración de la inferencia de nuestras redes neuronales.

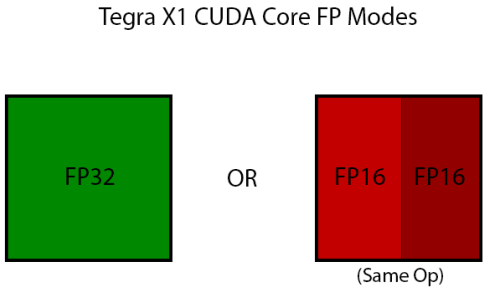


Figura 4.1: Ejecución sobre un núcleo CUDA de 32 bits de dos operaciones del mismo tipo en FP16.

4.2. Optimizadores: ONNXRuntime y TensorRT

Como hemos comentado en la sección 3.3, tanto la definición de nuestra red neuronal como su entrenamiento los hemos realizado utilizando la librería Keras sobre TensorFlow. Veremos ahora que la inferencia empleando estas librerías se puede acelerar enormemente sobre otras plataformas. En esta sección estudiaremos las dos que hemos utilizado: el formato de representación de modelos ONNX (Open Neural Network Exchange) [ONN] y su optimizador ONNXRuntime [Mic] y TensorRT [Nvie], la librería de Nvidia para la optimización de la inferencia sobre sus tarjetas gráficas (Figura 4.2).

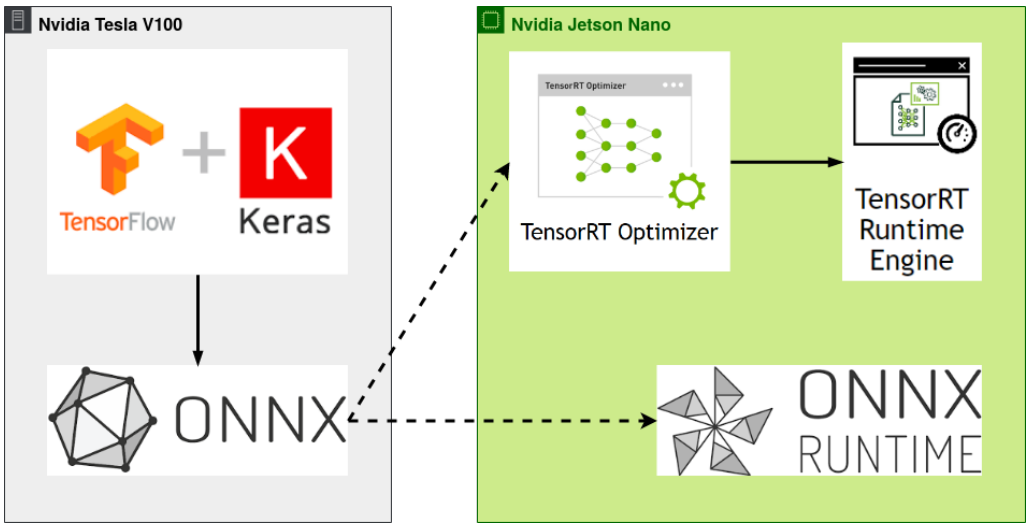


Figura 4.2: Flujo de trabajo para la aceleración de la inferencia.

ONNX es un formato libre de representación de modelos de aprendizaje automático que define un conjunto de operadores y un formato de archivo normalizado. De esta forma se obtiene una representación estándar que permite la interconexión entre una gran variedad de librerías de IA, herramientas, motores de ejecución y compiladores. Posibilita el desarrollo de los modelos mediante librerías como Keras, Caffe, MATLAB, PyTorch o TensorFlow y su posterior despliegue en entornos de ejecución diseñados para acelerar la inferencia sobre un hardware específico.

ONNXRuntime es un motor de inferencia de alto rendimiento que permite optimizar la ejecución de modelos de machine learning aprovechando de forma más eficiente las capacidades de cada hardware. El modelo ONNX lo convierte a una representación interna en memoria principal y le aplica una serie transformaciones, como por ejemplo la división del modelo en una serie de partes para que cada una se ejecute sobre un entorno de ejecución o acelerador diferente (CPU, nGraph, CUDA, TensorRT...).

ONNX también nos sirve como representación intermedia a partir de Keras para utilizar optimizadores sobre hardware más específicos, como puede ser TensorRT para las tarjetas gráficas de Nvidia. Utilizar TensorRT de forma directa en vez de indirectamente a partir de ONNXRuntime nos permite aplicar más y mejores optimizaciones adaptadas específicamente al hardware final. Esto se debe a que el paso del modelo en formato ONNX a TensorRT se realiza sobre este hardware, disponiendo de todos los parámetros del sistema definitivo. Sin embargo, estas transformaciones serán a costa de una disminución muy leve en la precisión obtenida.

Las optimizaciones que realiza TensorRT de cara a reducir el tiempo necesario para la inferencia son las siguientes:

- Disminuir el tamaño en memoria de los pesos para maximizar el ancho de banda, a expensas de reducir mínimamente la precisión del modelo. De esta

forma, se puede pasar de números en coma flotante de 32 bits a 16 bits o incluso a enteros de 8 bits mediante transformaciones y ajustes adicionales.

- Fusionar capas del modelo para optimizar el uso de la memoria de la GPU y el ancho de banda. Donde sea posible, transforma en un único nodo una porción más compleja del modelo inicial.
- Adaptar los algoritmos y datos usados en la ejecución de las capas del modelo a la GPU objetivo.
- Redistribuir los datos para minimizar la cantidad de huecos en memoria.
- Paralelizar la ejecución de forma que se utilice de forma eficiente toda la GPU.

4.3. Detección del tablero

Como ya hemos visto en el capítulo 2, la fase de detección del tablero se compone de distintos módulos que se ejecutan de forma iterativa hasta obtener la posición final. Para entender mejor el coste computacional de todo ese proceso, analizamos varias ejecuciones del código con la herramienta de análisis de rendimiento (*profiler*) que proporciona Python por defecto.

Los primeros pasos tras decidirnos por basar la detección del tablero en este método han sido:

- Adaptar el código a las versiones actuales del lenguaje y de las librerías utilizadas, sobretodo el paso de Python 2 a Python 3, OpenCV 2 a 4 y TensorFlow 1 a 2.
- Refactorizar el proyecto para que tenga mayor claridad y poder incluirlo más facilmente en nuestro framework.
- Adecuar todo el código a la guía de estilo de Python PEP8 [RWC01].

Una vez obtenido un marco funcional, pasamos a hacer un primer análisis del rendimiento del código utilizando el *profiler* de Python. Esta herramienta nos proporciona una serie de estadísticas y nos describe cuántas veces y durante cuánto tiempo se ha ejecutado cada función del código. En particular, por cada función nos indica el número veces que se ha llamado y el tiempo total y por llamada que ha tardado, distinguiendo entre incluir o no el tiempo que hayan empleado sus subfunciones en ejecutarse. Esto nos permitirá dedicar esfuerzo a optimizar las zonas del código que nos vayan a proporcionar una mayor disminución del tiempo.

Las ejecuciones para analizar el rendimiento de la detección de tablero las realizamos sobre nuestra plataforma final, la Nvidia Jetson Nano. Las imágenes de prueba son el conjunto de 10 fotos que utilizan los autores de [CLW17] para obtener sus

resultados. Estas son imágenes de situaciones muy variadas, que contienen otros objetos que forman líneas rectas adicionales además del tablero y que tienen sombras, distorsiones y ruido.

Los tiempos mostrados aquí son desde antes de leer las imágenes de entrada hasta después de escribir la imagen detectada de cada tablero (ver Figura 2.1). En particular, este no incluye la obtención de las casillas individuales. Para un análisis completo de los tiempos finales ver el capítulo 5.

4.3.1. Primer análisis: Optimización mediante ONNX

En la ejecución del código en esta fase obtenemos que se tarda una media de 16,01 segundos en procesar cada uno de los 10 tableros de prueba. Este tiempo lo obtenemos utilizando la librería `timeit` y ejecutando 5 veces el conjunto completo de tableros. Los siguientes datos desgranados vienen dados por el *profiler*, que conlleva un ligero sobrecoste, por lo que los expresamos como porcentajes del tiempo total para abstraernos de esta diferencia.

Inicialmente, el 38,1 % del tiempo se emplea en la fase de identificación de la posición del tablero (CPS) 2.1.3, el 37,4 % en la fase de búsqueda de puntos de la cuadrícula (LAPS) 2.1.2 y el 18 % en la fase de detección de líneas rectas (SLID) 2.1.1. De aquí, observamos que más de tres cuartas partes del tiempo de búsqueda de puntos de la cuadrícula se emplea en la inferencia de la red neuronal, desarrollada por [CLW17], utilizada para terminar de decidir los puntos que no filtra el detector geométrico (véase el Algoritmo 2.3). Por lo tanto, un buen punto de partida para optimizar el código sería acelerar esta red neuronal.

Para ello, primero recuperamos del proyecto original la estructura del modelo y los pesos entrenados para detectar los puntos que forman parte de la cuadrícula del tablero. Posteriormente, transformamos el modelo a formato ONNX para poder reducir el tiempo necesario para su ejecución mediante ONNXRuntime, como hemos visto en la sección 4.2. De esta manera, haciendo unos cambios mínimos en el código, podremos aprovechar un modelo previamente entrenado optimizándolo de forma casi transparente.

Tras optimizar esta parte del programa, hacemos una nueva prueba de rendimiento y comprobamos que ahora se tarda una media de 10,33 segundos en procesar cada uno de los 10 tableros de prueba. Así, centrándonos en mejorar las funciones en las que se invierte una mayor cantidad de tiempo, conseguimos un *speedup* de 1,55.

4.3.2. Segundo análisis: Reducción del sobrecoste de NumPy

Una vez optimizada la ejecución de la red neuronal en la fase LAPS tras el primer análisis, volvemos a evaluar el rendimiento del código para ver por dónde podemos seguir mejorando. En este segundo análisis, aproximadamente el 41,4 % del tiempo

se emplea en la identificación de la posición del tablero, el 28,6 % en la detección de líneas rectas y el 21,8 % en la búsqueda de puntos de la cuadrícula (recordemos que esto antes ocupaba el 37,4 % del tiempo).

Llegado a este punto, nos sorprende que casi la tercera parte del tiempo total se emplea en el cálculo de un producto vectorial utilizando la librería NumPy [Oli06]. Esta operación forma parte de una pequeña función utilizada tanto por la identificación de la posición del tablero como por la detección de líneas rectas. El cálculo es simplemente

$$\|(y - x) \times (x - z)\|,$$

donde observamos que x, y y z son vectores de dos componentes. Sabiendo esto, podemos transformar el mismo cálculo a otra forma equivalente y mucho más sencilla sin necesidad de hacer llamadas a la librería NumPy. Las llamadas a librerías, al introducir pequeños sobrecostes por su amplio espectro de posibles usos, acaban siendo menos eficientes.

El cálculo anterior puede simplificarse como

$$|(y_1 - x_1)(x_2 - z_2) - (y_2 - x_2)(x_1 - z_1)|,$$

donde hemos denotado $x = (x_1, x_2)$, $y = (y_1, y_2)$ y $z = (z_1, z_2)$. Esta operación puede hacerse directamente sin necesidad de utilizar librerías externas. También hemos evitado el cálculo de la norma del vector resultante del producto vectorial introduciendo simplemente un valor absoluto.

Hay que notar que esta operación acaba realizándose más de 25000 veces por tablero detectado. Por lo tanto, intentamos ver si se puede reducir su uso de alguna manera. Reordenando la forma de calcular si una línea está cerca del borde de la cuadrícula (véase el Algoritmo 2.4), evitamos llamadas a esta operación cuyos resultados a veces no eran necesarios. Así, conseguimos reducir casi un 20 % el número de invocaciones a esta función.

Tras estos cambios hacemos una nueva prueba de rendimiento y comprobamos que ahora se tarda una media de 5,55 segundos en procesar cada uno de los 10 tableros de prueba, consiguiendo un *speedup* de 1,86 sobre la optimización anterior.

4.3.3. Tercer análisis: Reducción del sobrecoste de Bentley-Ottmann

Después de observar cómo las llamadas a librerías pueden suponer un coste computacional mayor de lo necesario si no están justificadas, volvemos a analizar el código por si hubiese alguna otra situación similar. En el tercer análisis, aproximadamente el 40,2 % del tiempo se emplea en la búsqueda de puntos de la cuadrícula, el 32,3 % en la identificación de la posición del tablero y el 20,6 % en la detección de líneas rectas.

Aquí, destaca el hecho de que una función dedicada al cálculo de las intersecciones de un conjunto de líneas mediante el algoritmo de Bentley-Ottmann ocupa

casi el 39 % del tiempo total. Vimos al principio del Algoritmo 2.3 que este método de barrido se emplea para obtener todos los puntos candidatos a formar parte de la cuadrícula del tablero. Sin embargo, aquí vemos que además de para ese caso, también se está utilizando para calcular la intersección de cada par de líneas verticales y horizontales candidatas a ser el marco de la cuadrícula 6×6 del tablero (función `polyscore` del Algoritmo 2.4). Este segundo uso supone el 56,5 % del tiempo empleado en este cálculo.

El algoritmo de Bentley-Ottmann permite calcular las intersecciones de un conjunto de líneas de forma más eficiente, $\mathcal{O}((n+k)\log n)$, que la trivial, $\mathcal{O}(n^2)$, donde n es el número de líneas y k el número de intersecciones. Para conseguir esto se añade un sobrecoste que no compensa para conjuntos pequeños de líneas, en nuestro segundo caso son solo 4 líneas.

Solventamos esta situación calculando directamente las 6 posibles intersecciones de 2 líneas de entre el conjunto de 4. Esto consiste básicamente en el cálculo de determinantes de orden 2 y operaciones elementales, que en nuestro caso, al ser un conjunto pequeño, no supone ninguna ineficiencia práctica. De esta forma reducimos el tiempo en procesar cada uno de los 10 tableros de prueba a una media de 4,22 segundos por tablero, consiguiendo un *speedup* de 1,32 sobre la optimización anterior.

4.4. Clasificación de las piezas

En las secciones 3.3 y 3.4 hemos detallado el proceso de clasificación de las piezas del tablero utilizando una red neuronal profunda construida mediante Keras. Una vez establecido el proceso a través del cual podemos completar la digitalización del tablero, estamos en situación de escoger qué modelo utilizaremos para la inferencia y cómo aceleraremos su ejecución.

Efectuaremos las pruebas sobre 5 imágenes con tableros de ajedrez que contienen piezas de distinto tipo al utilizado en el conjunto de datos de entrenamiento. De esta forma, podremos comprobar lo robusto que es cada modelo ante cambios en el tipo de las piezas. Cada tablero tiene entre 21 y 32 piezas en posiciones diversas extraídas de partidas reales. A diferencia del esquema que mostramos inicialmente en las Figuras 1.1 y 1.2, las fotos para estas pruebas las hemos tomado desde un plano cenital. Adaptándonos así a las situaciones que hemos comentado en la sección 2.2 al obtener las casillas individuales del tablero (Figura 4.3). Como propondremos en el capítulo 6, en el caso de disponer de un dataset con el que recortar las casillas utilizando el método visto en el apartado 2.2.1, sí que pasaríamos a tomar las fotos desde un lateral.

Los valores para la precisión los tomaremos como el porcentaje de casillas predichas correctamente por nuestro programa. Esto no coincide exactamente con lo que sería el valor *Top-1* del modelo, ya que además tenemos en cuenta el conocimiento sobre el dominio que hemos detallado en la sección 3.4.

Figura 4.3: Una de las imágenes de prueba para la clasificación de las piezas. Tomada en el XXXVII Torneo de Ajedrez Pueblo Nuevo 60'+30".

4.4.1. Elección del modelo

Adelantábamos en la sección 3.3 los modelos que hemos escogido para la clasificación. Una vez entrenados sobre el dataset de piezas de ajedrez, tenemos que hacer una comparación entre las precisiones obtenidas y el tiempo necesario para la inferencia. Para ello, en una primera aproximación ejecutamos la inferencia sobre la Jetson Nano utilizando los modelos obtenidos directamente de Keras y obtenemos los resultados de la Tabla 4.1.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95 %	18,73s	94 %	28,95s	94 %	11,81s	98 %	5,85s
Test 2	94 %	18,73s	95 %	28,98s	92 %	12,01s	89 %	5,81s
Test 3	95 %	18,73s	94 %	28,88s	91 %	11,73s	92 %	5,88s
Test 4	91 %	18,72s	91 %	28,96s	91 %	11,68s	91 %	5,85s
Test 5	97 %	18,72s	88 %	29,03s	98 %	11,89s	92 %	5,90s
Media	94 %	18,73s	92 %	28,96s	93 %	11,82s	92 %	5,86s

Tabla 4.1: Precisión y tiempo obtenido sobre la Jetson Nano para cada uno de los modelos iniciales ejecutando la inferencia mediante Keras.

De estos primeros resultados podemos concluir que los cuatro modelos son lo suficientemente complejos como para aprender las características que proporciona la imagen de una casilla, ya que proporcionan valores similares de precisión. De ellos, el más rápido es el que tiene un menor número de parámetros, es decir, MobileNetV2.

Comparando, MobileNetV2 tiene unos 3,6 millones de parámetros frente a los 5,4 millones de NasNetMobile, los 20,3 millones de DenseNet201 o los 23 millones de Xception.

Viendo estos resultados nos preguntamos si con modelos más sencillos, cuya inferencia debería ser más rápida, podríamos obtener una precisión similar. Así, decidimos entrenar también una red AlexNet y otra SqueezeNet-v1.1. Además de estas dos redes, la propia MobileNetV2 contiene un parámetro α que controla el ancho de la red. Con $\alpha = 1$, el número de filtros en cada capa es el valor por defecto que proporcionan los autores del modelo. Con $\alpha > 1$ se aumenta y con $0 < \alpha < 1$ se disminuye proporcionalmente el número de filtros en cada capa. Así, incluimos en las pruebas la propia red MobileNetV2 con $\alpha = 0{,}5$ y $\alpha = 0{,}35$, valores para los cuales Keras proporciona pesos sobre ImageNet para inicializar el entrenamiento.

	MobileNetV2 $\alpha = 0{,}5$		MobileNetV2 $\alpha = 0{,}35$		AlexNet		SqueezeNet-v1.1	
Test 1	95 %	3,10s	89 %	3,02s	81 %	–	97 %	1,27s
Test 2	91 %	3,17s	75 %	3,05s	61 %	–	86 %	1,28s
Test 3	94 %	3,09s	83 %	3,02s	81 %	–	92 %	1,28s
Test 4	88 %	3,08s	83 %	3,02s	75 %	–	84 %	1,29s
Test 5	91 %	3,12s	89 %	3,04s	73 %	–	94 %	1,28s
Media	92 %	3,11s	84 %	3,03s	74 %	–	91 %	1,28s

Tabla 4.2: Precisión y tiempo obtenido sobre la Jetson Nano para cada uno de los modelos más sencillos ejecutando la inferencia mediante Keras.

En la Tabla 4.2 podemos observar los resultados de esta segunda batería de pruebas. Comprobamos que MobileNetV2 mantiene la precisión con $\alpha = 0{,}5$, reduciendo el tiempo empleado en la inferencia a casi la mitad que con $\alpha = 1$. El otro modelo que consigue una precisión superando el 90 % es SqueezeNet-v1.1, que además es el más rápido de todos los candidatos al ejecutarse directamente mediante Keras sobre la Jetson Nano. La precisión de MobileNetV2 con $\alpha = 0{,}35$ es notablemente inferior a la variante con $\alpha = 0{,}5$ sin conseguir apenas mejoras en el tiempo.

El modelo que peores resultados obtiene es AlexNet. En este caso, al igual que con SqueezeNet-v1.1, hemos realizado la implementación a partir de cada capa basándonos en otros modelos que no están incluidos oficialmente en la librería de Keras. Sin embargo, al contrario que con el resto de modelos, no hay disponibles pesos sobre ImageNet para la implementación que hemos realizado de AlexNet. Estos han sido los mejores resultados que hemos obtenido tras varios entrenamientos. Este hecho ha podido ser un factor a la hora de explicar por qué la precisión es inferior al resto cuando en teoría este modelo se comporta como SqueezeNet-v1.1 sobre ImageNet. Además, ha sido demasiado pesado en memoria para ejecutarse de esta manera sobre la Jetson Nano (los resultados de la Tabla 4.2 son los obtenidos en nuestro PC, de ahí que no incluyamos valores para los tiempos). Podremos hacer un mejor análisis

del coste computacional al ejecutar este modelo sobre la Jetson Nano en el apartado 4.4.2.

En este apartado hemos podido obtener unas cotas de los valores esperables para la precisión en la clasificación de las piezas. Para ello, hemos tenido en cuenta además los tiempos necesarios para la inferencia utilizando la aproximación más directa, la ejecución mediante Keras. La elección de un modelo adecuado que no sea ni demasiado complejo ni demasiado sencillo nos permitirá mejorar los resultados totales considerablemente. Como hemos estudiado, hay una diferencia de un orden de magnitud en el tiempo necesario para la inferencia entre modelos que obtienen una precisión similar. Diferencia que podría haber sido incluso mayor si hubiésemos escogido modelos aún más complejos de aquellos analizados en [Bia+18].

4.4.2. Optimización de la inferencia

Una vez hecho el análisis de los modelos que podemos utilizar para la inferencia, pasamos a optimizarlos utilizando los métodos que hemos estudiado al principio de esta sección, ONNXRuntime y TensorRT. Para ello, primero tenemos que convertir los modelos en formato HDF5 [The97] que tenemos en Keras, al formato estándar de ONNX. Una vez los tenemos en esta forma, pasamos a ejecutarlos sobre la Jetson Nano utilizando ONNXRuntime.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95 %	9,21s	94 %	9,03s	94 %	3,42s	98 %	1,38s
Test 2	94 %	9,19s	95 %	9,02s	92 %	3,40s	89 %	1,39s
Test 3	95 %	9,20s	94 %	9,05s	91 %	3,43s	92 %	1,38s
Test 4	91 %	9,20s	91 %	9,03s	91 %	3,43s	91 %	1,37s
Test 5	97 %	9,20s	88 %	9,01s	98 %	3,41s	92 %	1,38s
Media	94 %	9,20s	92 %	9,03s	93 %	3,42s	92 %	1,38s

	MobileNetV2 $\alpha = 0{,}5$		MobileNetV2 $\alpha = 0{,}35$		AlexNet		SqueezeNet-v1.1	
Test 1	95 %	0,86s	89 %	0,75s	83 %	2,34s	97 %	0,81s
Test 2	91 %	0,85s	75 %	0,72s	61 %	2,38s	86 %	0,76s
Test 3	94 %	0,87s	83 %	0,77s	80 %	2,37s	92 %	0,83s
Test 4	88 %	0,83s	83 %	0,76s	78 %	2,35s	84 %	0,78s
Test 5	91 %	0,86s	89 %	0,77s	72 %	2,31s	94 %	0,80s
Media	92 %	0,85s	84 %	0,75s	75 %	2,35s	91 %	0,80s

Tabla 4.3: Precisión y tiempo obtenido sobre la Jetson Nano para cada uno de los modelos ejecutando la inferencia mediante ONNXRuntime.

Podemos observar en la Tabla 4.3 que los tiempos se reducen considerablemente al utilizar ONNXRuntime como motor de inferencia, llegando algunos modelos

a situarse por debajo del segundo por tablero. Además, esto se logra sin ningún perjuicio en los valores de la precisión. La precisión en AlexNet sí se ve modificada ligeramente al alza al haberse realizado las pruebas del apartado anterior en otra plataforma diferente. Estas pequeñas variaciones son esperables, ya que cada motor de inferencia se adapta al hardware y los tamaños de datos disponibles.

Veamos pues qué resultados obtenemos si transformamos los modelos en formato ONNX a los nuevos grafos de las redes modificadas por TensorRT. Para aprovechar al máximo el hardware disponible en la Jetson Nano, esta transformación la ejecutamos sobre la propia placa. Así, obtenemos los nuevos motores de inferencia optimizados que se ejecutarán sobre la GPU.

Mostramos los resultados de la ejecución de todos los modelos utilizando TensorRT en la Tabla 4.4. En este caso, la precisión de los modelos sobre nuestros tableros de prueba se mantiene constante, aunque podrían ser esperables variaciones mínimas al estar pasando de números en coma flotante de 32 a 16 bits. Los tiempos sobre MobileNetV2 solo disminuyen ligeramente con respecto a los obtenidos utilizando ONNXRuntime. Sin embargo, el resto de modelos sí experimentan mejoras notables.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95 %	6,31s	94 %	7,25s	–	–	98 %	1,20s
Test 2	94 %	6,32s	95 %	7,20s	–	–	89 %	1,19s
Test 3	95 %	6,31s	94 %	7,06s	–	–	92 %	1,22s
Test 4	91 %	6,30s	91 %	7,03s	–	–	91 %	1,19s
Test 5	97 %	6,30s	88 %	7,05s	–	–	92 %	1,19s
Media	94 %	6,31s	92 %	7,12s	–	–	92 %	1,20s

	MobileNetV2 $\alpha = 0{,}5$		MobileNetV2 $\alpha = 0{,}35$		AlexNet		SqueezeNet-v1.1	
Test 1	95 %	0,77s	89 %	0,75s	83 %	1,75s	97 %	0,46s
Test 2	91 %	0,80s	75 %	0,79s	61 %	1,69s	86 %	0,47s
Test 3	94 %	0,77s	83 %	0,75s	80 %	1,69s	92 %	0,46s
Test 4	88 %	0,78s	83 %	0,72s	78 %	1,68s	84 %	0,46s
Test 5	91 %	0,78s	89 %	0,71s	72 %	1,68s	94 %	0,47s
Media	92 %	0,78s	84 %	0,74s	75 %	1,70s	91 %	0,46s

Tabla 4.4: Precisión y tiempo obtenido sobre la Jetson Nano para cada uno de los modelos ejecutando la inferencia mediante TensorRT.

No nos ha sido posible la transformación de NASNetMobile a TensorRT por incompatibilidades de versiones de las librerías con algunas de las capas del modelo. Esto nos supone un pequeño contratiempo a la hora de reducir aún más el tiempo de cómputo de esta red. Sin embargo, su ejecución sobre ONNXRuntime proporciona un tiempo comparable con los que obtenemos para el resto de redes utilizando

TensorRT, con lo que se puede seguir teniendo en cuenta de cara a un futuro.

Una vez optimizada la ejecución de nuestros modelos mediante TensorRT, nos preguntamos si podríamos conseguir reducir aún más los tiempos necesarios para la inferencia. Una técnica utilizada ampliamente cuando no están en uso simultáneamente todos los recursos disponibles, es realizar la inferencia agrupando varias imágenes a la vez. Esta ejecución por *batches* permite a TensorRT implementar optimizaciones adicionales para que se utilice de forma efectiva toda la GPU. Probamos pues a ejecutar la inferencia de las 64 imágenes correspondientes a las casillas de todo el tablero conjuntamente.

En la Tabla 4.5 mostramos los resultados de esta nueva ejecución en *batches*. La precisión se sigue manteniendo sin cambios y los tiempos se reducen un poco más en general, aunque hay alguna red que se mantiene con los tiempos anteriores. Probamos también otros tamaños de *batches* más pequeños, pero esto no mejora más los tiempos.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95 %	5,61s	94 %	6,03s	–	–	98 %	0,90s
Test 2	94 %	5,67s	95 %	6,04s	–	–	89 %	0,90s
Test 3	95 %	5,62s	94 %	6,05s	–	–	92 %	0,90s
Test 4	91 %	5,60s	91 %	6,05s	–	–	91 %	0,90s
Test 5	97 %	5,60s	88 %	6,04s	–	–	92 %	0,90s
Media	94 %	5,62s	92 %	6,04s	–	–	92 %	0,90s

	MobileNetV2 $\alpha = 0{,}5$		MobileNetV2 $\alpha = 0{,}35$		AlexNet		SqueezeNet-v1.1	
Test 1	95 %	0,59s	89 %	0,52s	83 %	0,61s	97 %	0,46s
Test 2	91 %	0,60s	75 %	0,53s	61 %	0,62s	86 %	0,47s
Test 3	94 %	0,60s	83 %	0,52s	80 %	0,62s	92 %	0,46s
Test 4	88 %	0,60s	83 %	0,53s	78 %	0,61s	84 %	0,46s
Test 5	91 %	0,59s	89 %	0,52s	72 %	0,62s	94 %	0,47s
Media	92 %	0,60s	84 %	0,52s	75 %	0,62s	91 %	0,46s

Tabla 4.5: Precisión y tiempo obtenido sobre la Jetson Nano para cada uno de los modelos ejecutando la inferencia mediante TensorRT en un *batch* de 64 imágenes.

Las transformaciones que hemos estudiado en este apartado nos han permitido reducir considerablemente el tiempo necesario para clasificar las piezas del tablero. Este ha sido el paso final de cara a conseguir una ejecución eficiente de nuestro framework sobre la plataforma que hemos escogido.

Capítulo 5
Resultados

En este capítulo analizaremos los resultados finales que hemos obtenido y los compararemos tanto con los resultados iniciales como con otras propuestas. Además, estudiaremos el tiempo total que emplea nuestro programa en ejecutar la digitalización completa, desde la lectura de la imagen de un tablero hasta el cálculo de la notación FEN de su configuración.

5.1. Detección del tablero

Veamos primero los resultados obtenidos al detectar el tablero. Como hemos detallado en la sección 4.3, las modificaciones que hemos hecho sobre el código propuesto en [CLW17] no perjudican la precisión final. Por tanto, mantenemos sus mismos valores: 99,5 % de precisión al detectar las intersecciones de la cuadrícula central del tablero (Algoritmo 2.3) y 95 % de aciertos al encontrar la posición completa de forma precisa.

En cuanto a los tiempos finales, hemos conseguido una mejora muy notable. Analizamos los tiempos ejecutando nuestro programa sobre la Nvidia Jetson Nano con el conjunto de 10 fotos utilizado en [CLW17]. Como buscamos reducir la latencia al máximo, medimos el tiempo transcurrido desde antes de la lectura de la imagen de entrada, hasta la escritura de la imagen recortada del tablero.

En la Tabla 5.1 mostramos los tiempos medios por tablero obtenidos ejecutando el código original, después de adaptarlo, pasando por las distintas mejoras vistas en la sección 4.3 y terminando con la implementación final que incluye algunas optimizaciones menores adicionales. El conjunto de todas las optimizaciones que hemos implementado han conseguido reducir la latencia de la detección del tablero a menos de una cuarta parte. Concretamente, obtenemos un *speedup* de 4,27, pasando de 16,38 a 3,84 segundos por tablero de media.

	Original	Adaptado	ONNX	NumPy	Bentley-Ottmann	Final
Tiempo	16,38s	16,01s	10,33s	5,55s	4,22s	**3,84s**
Speedup	-	1,02	1,55	1,86	1,32	1,10
Acumulado	-	1,02	1,59	2,95	3,88	**4,27**

Tabla 5.1: Tiempo medio por imagen obtenido sobre la Jetson Nano para cada una de las versiones de la detección del tablero. Los *speedups* son los obtenidos con respecto al paso anterior y el acumulado con respecto al tiempo inicial.

En [CLW17], los autores comentan en los resultados que el único inconveniente de su método es que es hasta dos veces más lento que otras alternativas. Con las optimizaciones que hemos introducido nosotros se cambian las tornas, y previsiblemente es este método el que resulta dos veces más rápido que los propuestos en [EA10] y [DK15], además de ser más preciso.

5.2. Clasificación de las piezas

Hemos visto en el capítulo 3 y en la sección 4.4 el desarrollo que hemos seguido para la elección, entrenamiento y optimización de las redes neuronales convolucionales que hemos utilizado para la clasificación de las piezas. Además, en la sección 3.4 incluimos algunas reglas del ajedrez para cohesionar toda la información que obtenemos del tablero de cara a la inferencia. Así, para cada red tendremos por un lado el valor *Top-1*, que nos indica el porcentaje de aciertos del modelo al elegir siempre el máximo de cada vector de probabilidades, y por otro lado la precisión conseguida tras estas mejoras globales.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95 %	95 %	91 %	94 %	94 %	94 %	95 %	98 %
Test 2	92 %	94 %	86 %	95 %	91 %	92 %	89 %	89 %
Test 3	97 %	95 %	94 %	94 %	91 %	91 %	92 %	92 %
Test 4	89 %	91 %	91 %	91 %	89 %	91 %	89 %	91 %
Test 5	91 %	97 %	83 %	88 %	97 %	98 %	92 %	92 %
Media	93 %	**94 %**	89 %	**92 %**	92 %	**93 %**	91 %	**92 %**
	Top-1	Dominio	*Top-1*	Dominio	*Top-1*	Dominio	*Top-1*	Dominio

	MobileNetV2 $\alpha = 0{,}5$		MobileNetV2 $\alpha = 0{,}35$		AlexNet		SqueezeNet-v1.1	
Test 1	95 %	95 %	84 %	89 %	83 %	83 %	91 %	97 %
Test 2	86 %	91 %	72 %	75 %	61 %	61 %	81 %	86 %
Test 3	94 %	94 %	80 %	83 %	77 %	80 %	89 %	92 %
Test 4	84 %	88 %	81 %	83 %	73 %	78 %	83 %	84 %
Test 5	89 %	91 %	86 %	89 %	72 %	72 %	89 %	94 %
Media	90 %	**92 %**	81 %	**84 %**	73 %	**75 %**	87 %	**91 %**
	Top-1	Dominio	*Top-1*	Dominio	*Top-1*	Dominio	*Top-1*	Dominio

Tabla 5.2: Valor *Top-1* y precisión tras introducir conocimiento del dominio para cada uno de los modelos.

En la Tabla 5.2 mostramos las precisiones obtenidas por cada modelo que hemos entrenado para los tableros de prueba descritos al inicio de la sección 4.4. Aquí podemos observar que la precisión en general aumenta al introducir información sobre el dominio, experimentando subidas de entre el 1 y el 4 % de media. Nótese

que en algún caso comete algún fallo más. Esto es debido, por ejemplo, a errores a la hora de escoger en qué casilla está uno de los reyes, ya que si hay dos casillas con una probabilidad muy alta de ser un rey, puede darse el caso de que esta pieza se encuentre en la casilla que tenga una probabilidad ligeramente inferior a la otra (Algoritmo 3.1).

Errores similares a este son esperables por los valores de probabilidades devueltos por los modelos. Lo más importante es que esto aumenta la precisión en general e inserta coherencia en los resultados, ya que en todo momento son configuraciones de piezas que podrían darse en una partida de ajedrez. Estas precisiones se mantienen constantes al ejecutarse sobre los diferentes optimizadores.

Una vez establecida la precisión de la inferencia, veamos los tiempos finales que hemos obtenido tras la optimización de los modelos vista en el apartado 4.4.2. En la Tabla 5.3 resumimos los mejores resultados que hemos conseguido con cada modelo. Como se puede observar, siempre que hemos podido transformar los modelos, el mejor motor de inferencia ha sido TensorRT con un tamaño de *batch* de 64. Además, los tiempos de los modelos que consideramos al principio estaban comprendidos entre 5,86 y 28,96 segundos por tablero (ver Tabla 4.1), frente a los 0,46 a 6,04 segundos conseguidos finalmente.

	Tiempo medio	Precisión media	Motor de inferencia
SqueezeNet-v1.1	0,46s	91 %	TensorRT b64
MobileNetV2 $\alpha = 0{,}35$	0,52s	84 %	TensorRT b64
MobileNetV2 $\alpha = 0{,}5$	0,60s	92 %	TensorRT b64
AlexNet	0,62s	75 %	TensorRT b64
MobileNetV2	0,90s	92 %	TensorRT b64
NASNetMobile	3,42s	93 %	ONNXRuntime
Xception	5,62s	94 %	TensorRT b64
DenseNet201	6,04s	92 %	TensorRT b64

Tabla 5.3: Mejores tiempos por tablero y precisiones obtenidas para cada modelo sobre la Jetson Nano, junto con el optimizador utilizado.

Para concluir esta sección, veamos en una gráfica la comparación entre tiempo necesario y precisión conseguida para todos los modelos. En la Figura 5.1 mostramos esta relación junto con el tamaño en número de parámetros de cada red. Como se puede observar, la curva naranja indica qué redes forman el frente de Pareto, es decir, aquellas que optimizan la relación entre los dos valores estudiados. Por

tanto, reducimos el número de modelos a considerar de 8 a 4 y, dependiendo de nuestras necesidades, podríamos optar por una de las siguientes redes: SqueezeNet-v1.1, MobileNetV2 ($\alpha = 0{,}5$), NASNetMobile o Xception. Nótese que la ejecución de NASNetMobile es sobre ONNXRuntime.

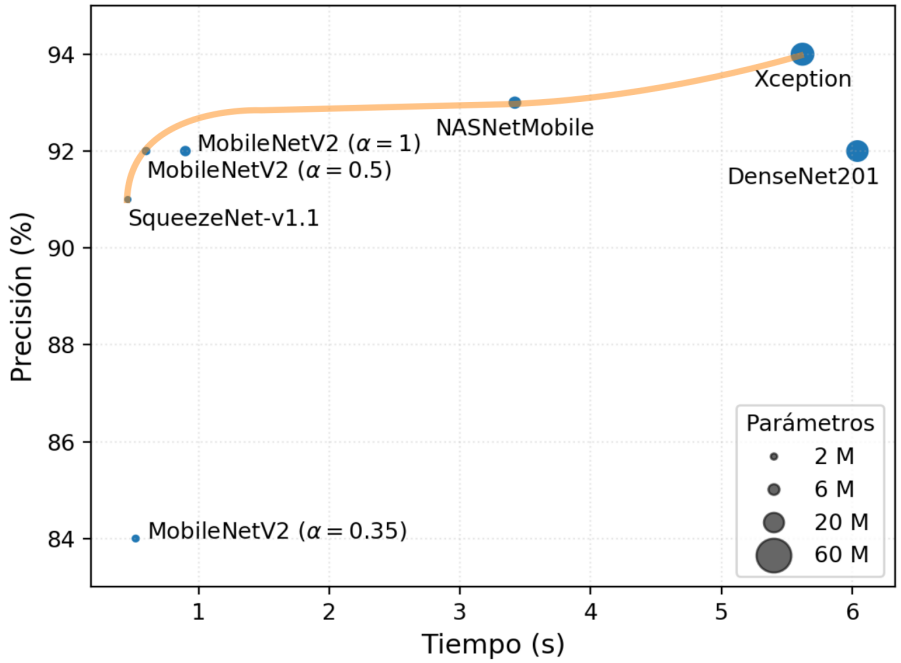


Figura 5.1: Precisión obtenida por cada modelo en función del tiempo necesario para la inferencia de un tablero sobre la Jetson Nano mediante TensorRT en un *batch* de 64 imágenes, salvo NASNetMobile, cuyo mejor tiempo es sobre ONNXRuntime. Omitimos AlexNet por obtener una precisión mucho menor.

5.3. Digitalización total

La completa digitalización de un tablero de ajedrez comprende desde que el sistema detecta que hay una nueva imagen a procesar hasta que termina de procesarla y produce una cadena en notación FEN. En las secciones anteriores hemos estudiado en detalle los resultados que hemos obtenido para la detección del tablero en la imagen inicial y la clasificación de las piezas una vez separadas las casillas individuales. Además de estas dos acciones, que suponen la practica totalidad del tiempo de ejecución, existen otras tres operaciones que hay que completar para terminar todo el proceso.

En primer lugar, una vez detectado el tablero de ajedrez debemos separar las casillas individuales (sección 2.2). Esto supone unos 80 milisegundos sobre la Jetson Nano. Además, después de obtener los vectores de probabilidades de la red que es-

temos utilizando, hay que inferir cuál es la configuración de piezas (sección 3.4) y finalmente transformar esta información a la notación FEN (sección 3.1). La suma del tiempo necesario para ejecutar estas dos operaciones es inferior a los 10 milisegundos. Por tanto, adicionalmente a los tiempos de los procesos que hemos estudiado en profundidad, debemos añadir algo menos de 100 milisegundos para completar la digitalización.

Resumimos todos los tiempos que hemos obtenido en la Tabla 5.4. Aquí podemos ver de forma más clara que existe una variable importante a la hora de desplegar el sistema, qué red utilizar para la clasificación de las piezas. Esto dependerá del uso final del programa y, como veremos en el capítulo de conclusiones y trabajo futuro, de las necesidades y mejoras que se puedan implementar.

		Test 1	Test 2	Test 3	Test 4	Test 5
Detección tablero		4,21s	3,30s	4,28s	3,69s	3,35s
Separar casillas individuales				0,08s		
Obtener vectores de probabilidades	SqueezeNet-v1.1			0,46s		
	MobileNetV2 ($\alpha = 0{,}5$)			0,60s		
	NASNetMobile			3,42s		
	Xception			5,61s		
Inferir piezas + Notación FEN				0,01s		
Total	SqueezeNet-v1.1	4,76s	3,85s	4,83s	4,24s	3,90s
	MobileNetV2 ($\alpha = 0{,}5$)	4,90s	3,99s	4,97s	4,28s	4,04s
	NASNetMobile	7,72s	6,81s	7,79s	7,20s	6,86s
	Xception	9,91s	9,00s	9,98s	9,39s	9,05s

Tabla 5.4: Resumen de tiempos totales sobre la Jetson Nano por tablero de prueba para cada uno de los modelos que forman el frente de Pareto.

Al estudiar los tiempos que emplea nuestro programa en ejecutar cada una de estas fases observamos que, si optamos por las redes más pequeñas para la clasificación de las piezas, la inmensa mayoría del tiempo transcurre en la fase de detección del tablero. En situaciones en las que sepamos que el tablero y la cámara van a estar inmóviles, podemos aprovechar para no recalcular su posición cada vez. Así, simplemente tenemos que mantener las coordenadas de las esquinas que hemos calculado anteriormente, comprobar que el tablero sigue en el mismo sitio y pasar a separar las casillas directamente.

Siguiendo la idea introducida en el apartado 2.1.2, hemos implementado un algoritmo que nos permite comprobar de una forma rápida si el tablero sigue en la misma posición. Para ello, utilizamos el detector geométrico y la red neuronal vistos al buscar los puntos de la cuadrícula. Si las 49 esquinas del cuadrado central 6×6

del tablero siguen en su sitio, podremos afirmar que la información que teníamos sigue siendo correcta. Al contar las esquinas que sí forman parte de la cuadrícula tendremos que tomar un margen de tolerancia, ya que hay puntos que pueden estar ocluidos por alguna pieza. De forma experimental hemos comprobado que acertando 20 de los 49 puntos que comprobamos es suficiente para afirmar que el tablero sigue en el mismo sitio (Algoritmo 5.1).

Algoritmo 5.1: Comprobación de la posición del tablero.

Entrada: Imagen que contiene un tablero.
Esquinas candidatas a seguir formando parte del cuadrado central 6×6 del tablero.

Salida: Si el tablero sigue en la misma posición.

```
1  esquinas_correctas ← 0;
2  para cada punto ∈ esquinas hacer
3  |  matriz ← preprocesar(vecindad(imagen, punto));
4  |  es_punto_cuadricula ← detector_geometrico(matriz);
5  |  si es_punto_cuadricula entonces
6  |  |  esquinas_correctas ← esquinas_correctas + 1;
7  |  en otro caso
8  |  |  es_punto_cuadricula ← red_neuronal(matriz);
9  |  |  si es_punto_cuadricula entonces
10 |  |  |  esquinas_correctas ← esquinas_correctas + 1;
11 |  |  fin
12 |  |  // Si no pertenece a la cuadrícula, pasamos a la siguiente
13 |  fin
14 fin
15 devolver esquinas_correctas ≥ tolerancia (= 20)
```

La ejecución de este algoritmo sobre la Jetson Nano tarda unos 150 milisegundos por tablero. Luego en caso de devolver un resultado positivo conseguimos una reducción enorme en el tiempo total. En la Tabla 5.5 resumimos los tiempos cuando esta comprobación devuelve cierto.

Como podemos observar, esta predicción reduciría el tiempo total necesario para digitalizar nuevas posiciones a menos de un segundo. Además, con este tiempo se podría añadir un muestreo periódico de imágenes para capturar posibles jugadas intermedias. Se podrían evitar de esta forma los casos en los que haya, por ejemplo, una mano tapando parte del tablero.

Este pequeño sobrecoste que habría que añadir en caso de que la comprobación devuelva falso se ve altamente compensado. En la Tabla 5.4 hemos visto que el tablero que se detecta más rápido tarda 3,30 segundos, luego con que la comprobación devolviera cierto en 1 de cada 14 ocasiones, se verían amortizadas estas llamadas.

		Tablero inmóvil
Comprobación tablero		0,15s
Separar casillas individuales		0,08s
Obtener vectores de probabilidades	SqueezeNet-v1.1	0,46s
	MobileNetV2 ($\alpha = 0{,}5$)	0,60s
	NASNetMobile	3,42s
	Xception	5,61s
Inferir piezas + Notación FEN		0,01s
Total	SqueezeNet-v1.1	0,70s
	MobileNetV2 ($\alpha = 0{,}5$)	0,84s
	NASNetMobile	3,66s
	Xception	5,85s

Tabla 5.5: Resumen de tiempos totales para cada uno de los modelos que forman el frente de Pareto sobre la Jetson Nano cuando la comprobación de la posición del tablero devuelve cierto.

Capítulo 6
Conclusiones y trabajo futuro

La visión artificial es un campo que está tomando una gran relevancia en muchas situaciones. El acceso de la sociedad a dispositivos capaces de interactuar con el mundo que les rodea ha supuesto la automatización de tareas muy diversas, bien sea mediante cálculos en sistemas empotrados o utilizando el poder de cómputo de potentes servidores alojados en la nube. En el ámbito del ajedrez, uno de los primeros juegos para los que se desarrolló un algoritmo de inteligencia artificial, también se puede aplicar este campo a la automatización de la digitalización de las partidas. En este trabajo hemos explorado una de las opciones que mejores resultados proporciona en la actualidad y hemos creado un programa que es capaz de seguir el ritmo de una partida de ajedrez sobre un sistema empotrado.

Como hemos podido observar en los resultados finales (Tabla 5.4), el tiempo total utilizando tanto SqueezeNet-v1.1 como MobileNetV2($\alpha = 0{,}5$) se mantiene por debajo de los 5 segundos para todos los tableros de prueba sobre la Jetson Nano. Esto es alcanzando una precisión de 91 y 92 % respectivamente a la hora de clasificar las piezas. Más aún, si el tablero permanece inmóvil, estos tiempos se reducen hasta los 0,70 y 0,84 segundos respectivamente.

El tiempo que nos marcamos como objetivo inicialmente fueron 5 segundos por jugada. Por lo tanto, después de partir de un tiempo total que rondaba entre los 25 y 45 segundos por tablero, duración que no hacía factible su uso práctico, hemos podido alcanzar una meta que permitiría su despliegue en situaciones reales. Todo ello realizando los cálculos íntegramente en un sistema empotrado sin necesidad de conectividad con ninguna red. El código completo del programa que hemos desarrollado se encuentra de forma pública en nuestro repositorio de GitHub: https://github.com/davidmallasen/LiveChess2FEN.

Entrenando las distintas redes neuronales convolucionales para la clasificación de las piezas nos hemos encontrado con una barrera a la hora de mejorar la precisión. Los modelos más sencillos han sido capaces de aprender prácticamente toda la información disponible en el dataset que hemos podido recopilar, y el hecho de tomar modelos más complejos no ha proporcionado casi beneficio (Tabla 4.1). Teniendo en cuenta esto, consideramos que parte de este problema se solucionaría introduciendo más información en el dataset, por ejemplo mediante el nuevo método de separación de casillas que hemos propuesto (apartado 2.2.1). Así evitaríamos los casos en los que solo se ve la base de la pieza, situación en la que ni siquiera un humano es capaz de distinguirla.

En el caso de que mejorando el dataset se obtuvieran resultados más precisos con un modelo que requiera un mayor esfuerzo de cómputo, la disponibilidad de

núcleos tensoriales para el cálculo de operaciones en FP16 sería una mejora crítica. Una evolución de la Jetson Nano, que tiene el mismo factor de forma, es la Jetson Xavier NX [Nvic]. Actualmente, este módulo ronda los 400$, unas 4 veces más que la Jetson Nano, aunque previsiblemente con la reciente salida de la arquitectura Ampere habrá una reducción de precios o nuevos modelos con precio inferior.

La Jetson Xavier NX tiene una GPU con una arquitectura Volta que aporta 384 núcleos CUDA, 48 núcleos tensoriales y 2 núcleos NVDLA (*NVidia Deep Learning Accelerator*), frente a los 128 núcleos CUDA de la Jetson Nano que hemos empleado nosotros. Además, dispone de una CPU hexa-core y 8 GB de RAM, ante los cuatro núcleos y 4 GB de nuestra plataforma. Todo ello hace que sea capaz de alcanzar 6 TFLOPS en FP16, unas 12 veces más de lo que disponemos actualmente. Además, a diferencia de la Jetson Nano, permite la ejecución de operaciones en enteros de 8 bits, a una razón de 21 TOPS. Esto último se podría aprovechar transformando los modelos y calibrando los rangos de los pesos mediante las capacidades que proporciona TensorRT.

El hecho de disponer de más núcleos de CPU haría factible la paralelización de algunas secciones del código de la detección del tablero. Por ejemplo, se podrían calcular paralelamente en diferentes procesos las distintas máscaras CLAHE en los pasos hasta la obtención de los segmentos, como hemos visto en el Algoritmo 2.2. Esto ha supuesto una mejora de un 10 % en el tiempo total sobre una CPU Intel Core i7-4770K con 4 núcleos con *hyperthreading*, utilizando la clase ProcessPoolExecutor de Python. Sin embargo, al hacer la prueba sobre la Jetson Nano los tiempos se han mantenido constantes.

En el apartado 3.4.1 hemos introducido un algoritmo que permite inferir las piezas que se mueven en un tablero a partir de dos imágenes consecutivas. En el caso de conocer con certeza la configuración inicial de las piezas, por ejemplo digitalizando una partida desde el comienzo, podremos conocer de forma precisa el número y tipo de piezas que hay en cada momento sobre el tablero. Esta información se podría incluir en el Algoritmo 3.1 para afinar más la función `alcanzado_max`.

En ocasiones, sobre todo en partidas infantiles, alguno de los jugadores realiza un movimiento ilegal. Teniendo la certeza del tipo de pieza que realiza el movimiento, por ejemplo por imágenes de las acciones anteriores, se podría advertir de estas situaciones. Esto serviría tanto como información adicional para el usuario, como para corregir malas predicciones que podrían darse a raíz de estas jugadas ilícitas.

Además, en todos los casos en los que sabemos que una pieza ha permanecido quieta, podemos utilizar los vectores de probabilidades que hemos obtenido en imágenes anteriores para aumentar nuestro conocimiento. Por ejemplo, podríamos tomar la media de estas probabilidades en las fotos en las que no se ha movido. De esta manera mejoraríamos la inferencia en situaciones en las que antes había una pieza ocluyendo parte de la casilla que estamos considerando.

Capítulo 7
Introduction (English)

The recognition of chess pieces and chessboards is a computer vision problem that has not yet been solved efficiently. However, its solution is crucial for many experienced players who want to compete against chess engines and specialized programs, but who also prefer to make decisions using a physical chessboard. It is also important for chess tournament organizers who want to broadcast the games online or for amateur players who want to share their games with friends. These digitizing tasks are usually performed by humans or with the help of specialized chessboards and pieces.

In order to digitize a game automatically, it is necessary to be able to recognize both the chessboard and afterwards the layout of the pieces. In live games, in addition to precision, it is critical to minimize the time required to perform such recognition, since the moves can happen very quickly. This is the case of the game modes known as "Blitz" and "Bullet", in which each player has between 1 and 5 minutes to play the entire game. This is also true in classic games when there is little time left.

7.1. Related work

Specialized boards that physically detect the pieces are a hardware solution for automatic chess digitization. Recent examples currently used in major tournaments can be found in [Squ] or [DGTa]. However, these boards are expensive and difficult to deploy in many areas. As an example, a set of DGT chessboard and pieces (brand used in official tournaments) costs between €500 and €1000 [DGTb].

Other alternatives are those provided by robots that move the pieces on a board, such as [Mat+11] or more recently [CW19]. These robots are based on positioning a zenith camera over the board and detecting the differentials between one movement and the next. A drawback of this is that it is necessary to start from a known initial layout. A board with a generic layout could not be digitized this way. In addition, errors should be taken into account, because they could add up in each new play.

The solutions offered by computer vision are an alternative to consider, since they provide cheaper and increasingly accurate systems, as well as being a major technological challenge. For comparison, the platform we will be using for inference, an Nvidia Jetson Nano, costs around €110 [Nvib] and does not just stick to one function, it could be reused for another task.

These computer vision procedures are based on combining and adapting trans-

formations and detectors already known and used in other fields, such as the Harris corner detector and the Hough transform [EA10]. Many of the methods often assume big simplifications, such as determining the exact position of the camera, using boards specifically designed with markers to aid in corner detection, or directly through user interaction [Din16]. However, some generic solutions that overcome these restrictions already exist.

For example, there are methods that allow us to classify occurrences of various objects in arbitrary places using convolutional neural networks (CNNs) [Gao+17], but they do not have the precision required to obtain their exact position. The authors of [Ben+16] describe a method for object detection that also uses CNNs trained through weak supervision. That is, it is enough to have labeled images of the objects in order to train the network, without the need to also provide the exact position of the object in each training image. The problem with this approach is that it takes many iterations to precisely locate an object, which does not make it very practical in situations that require real-time responses.

The authors of [CLW17] propose a method for chessboard detection that is robust against light conditions and the angle from which the images are taken. In addition, it works with most styles of boards and it overcomes many of the weaknesses that we have been discussing. It is an iterative process in which the position of the board is refined in several phases. The authors get 99,5 % accuracy in detecting the intersections of the center board grid and find the full position of the chessboard accurately 95 % of the time.

Regarding piece classification once the board has been located, in [Din16] a method is proposed that uses a support vector machine (SVM). This is trained on the features extracted by SIFT [Low04], thus achieving 85 % accuracy when classifying the pieces. In [CLW17] authors claim to achieve 95 % accuracy classifying chess pieces using a convolutional neural network (CNN), which they enhance by clustering similar pieces, taking into account their height and area, and using a game engine to obtain the probability of particular layouts. The code for these enhancements has not been released, so we were unable to analyze this particular approach.

7.2. Objectives and work plan

The processing of each frame is still too slow in order to host live broadcasts. Therefore, accelerating this computation is a very important challenge. Our final objective is to build a functional framework that is capable of executing the entire process of digitizing a chess game photo in real-time, making all the necessary calculations on specialized hardware.

To accomplish this, we have made a series of changes to a method already implemented that detects the chessboard. We have refactored the code and modified some modules to include new possibilities (see for example section 2.2.1 or board

position checking in section 5.3). In addition, we have optimized the code based on performance analysis tools and we have accelerated its execution on our hardware (section 4.3). To complete the framework, we have built a convolutional neural network to classify the obtained pieces and we have drastically reduced the necessary inference time in the final system (section 4.4).

We have carried this out having in mind its use in amateur games or in tournaments, taking the photos from the side of the board each time a player presses the clock (Figure 1.1). However, sometimes players forget to press the clock. In these cases, periodic sampling based on the processing speed of an image would be necessary to try to capture all the moves. We show a schematic of the camera position in Figure 1.2.

Capítulo 8
Conclusions and future work

Computer vision is a field that is acquiring great relevance in many situations. Universal access to devices capable of interacting with the world around them has meant the automation of very diverse tasks, either through embedded systems or using the computing power of servers hosted in the cloud. In chess, one of the first games for which an artificial intelligence algorithm was developed, computer vision can also be applied to automate the digitization of games. In this work we have explored one of the options that currently provide the best results and we have created a program that can keep up with a chess game on an embedded system.

As we have seen in the final results (Table 5.4), the total time using both SqueezeNet-v1.1 and MobileNetV2($\alpha = 0.5$) remains below 5 seconds for all test boards on the Jetson Nano. Achieving an accuracy of 91 and 92 % respectively when classifying the chess pieces. Furthermore, if the board remains static, these times are reduced to 0.70 and 0.84 seconds respectively.

Initially, the time we set as our goal was 5 seconds per board. Therefore, starting from a total time between 25 and 45 seconds, a cost that did not make its practical use feasible, we have been able to reach a target that would allow its deployment in practical scenarios. Moreover, the calculations are performed entirely on an embedded system without the need for connectivity to any network. The complete source code of the framework that we have developed is publicly available in our GitHub repository: https://github.com/davidmallasen/LiveChess2FEN.

Whilst training the different convolutional neural networks, we have found a barrier when it comes to improving precision for piece classification. The simplest models have been able to learn practically all the information available in the dataset that we have been able to collect, and training more complex models has provided almost no benefit (Table 4.1). With this in mind, we believe that part of this problem would be solved by introducing more information into the dataset, for example by means of the new method for square separation that we have proposed (section 2.2.1). This way we would avoid the cases in which only the base of the piece is seen, a situation in which not even a human is able to distinguish it.

If more accurate results were obtained by improving the dataset with a model that requires a greater computational effort, the availability of tensor cores for calculating operations in FP16 would be a critical improvement. An evolution of the Jetson Nano, which has the same form factor, is the Jetson Xavier NX [Nvic]. Currently, this module costs around $400, about 4 times more than the Jetson Nano, although with the recent release of the Ampere architecture, a reduction in prices or new models for a lower price are expected.

The Jetson Xavier NX has a Volta architecture GPU that provides 384 CUDA cores, 48 tensor cores and 2 NVDLA cores (*NVidia Deep Learning Accelerator*), compared to the 128 CUDA cores of the Jetson Nano that we have used. In addition, it has a hexa-core CPU and 8 GB of RAM, compared to the four cores and 4 GB of our platform. All this makes it capable of reaching 6 TFLOPS in FP16, about 12 times more than what we currently have. In addition, unlike the Jetson Nano, it allows 8-bit integer operations, achieving 21 TOPS. This could be exploited by transforming the models and calibrating the weight ranges using the capabilities provided by TensorRT.

An increase in the number of CPU cores would make the parallelization of some sections of the board detection code feasible. For example, the different CLAHE masks in the steps until obtaining the segments, as we have seen in Algorithm 2.2, could be calculated in parallel in different processes. This has resulted in an improvement in the total time by around 10 % on a quad-core Intel Core i7-4770K CPU with *hyperthreading*, using the Python ProcessPoolExecutor class. However, when testing this on the Jetson Nano, times have remained unchanged.

In section 3.4.1 we have introduced an algorithm that infers the pieces that have moved on a board from two consecutive images. In cases in which we can identify with certainty the initial configuration of the pieces, for example when digitizing a game from the beginning, we would be able to know precisely the number and type of pieces that are on the board at any time. This information could be included in Algorithm 3.1 to further refine the `reached_max` (`alcanzado_max`) function.

Sometimes, especially in children's games, one of the players makes an illegal move. Having the certainty of the type of piece that performs the movement, for example due to images of the previous actions, the algorithm could warn of these situations. This would serve both as extra information for the user, and to correct bad predictions that could occur as a result of these illegal movements.

Furthermore, in all the cases in which we know that a piece has remained still, we can use the probability vectors that we have obtained in previous images to increase our overall knowledge. For example, we could use the average of the probabilities in the photos in which a piece has not moved. In this way, we would improve the inference in situations where, in some images, there is another piece occluding part of the square that we are considering.

Referencias

[Aba+15] M. Abadi y col. *TensorFlow: Large-scale machine learning on heterogeneous systems*. 2015. URL: https://www.tensorflow.org/.

[Ben+16] A. J. Bency, H. Kwon, H. Lee, S. Karthikeyan y B. S. Manjunath. "Weakly supervised localization using deep feature maps". En: *European Conference on Computer Vision*. 2016, págs. 714-731.

[BO79] J. L. Bentley y T.A. Ottmann. "Algorithms for reporting and counting geometric intersections". En: *IEEE Transactions on Computers* C-28.9 (1979), págs. 643-647.

[Bia+18] S. Bianco, R. Cadène, L. Celona y P. Napoletano. "Benchmark analysis of representative deep neural network architectures". En: *IEEE Access* 6 (2018), págs. 64270-64277. DOI: 10.1109/ACCESS.2018.2877890.

[Can86] J. Canny. "A computational approach to edge detection". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6 (1986), págs. 679-698.

[CW19] A. Chen y K. Wang. "Robust Computer Vision Chess Analysis and Interaction with a Humanoid Robot". En: *Computers* 8 (feb. de 2019), pág. 14. DOI: 10.3390/computers8010014.

[Cho16] F. Chollet. *Xception: Deep learning with depthwise separable convolutions*. 2016. arXiv: 1610.02357.

[Cho+15] F. Chollet y col. *Keras*. https://keras.io. 2015. (Visitado 05-2020).

[CLW17] M. A. Czyzewski, A. Laskowski y S. Wasik. *Chessboard and chess piece recognition with the support of neural networks*. 2017. arXiv: 1708.03898. URL: https://github.com/maciejczyzewski/neural-chessboard.

[DK15] C. Danner y M. Kafafy. *Visual chess recognition*. 2015. URL: http://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Danner_Kafafy.pdf.

[Den+09] J. Deng y col. "ImageNet: A large-scale hierarchical image database". En: *IEEE Computer Vision and Pattern Recognition (CVPR)* (2009).

[DGTa] DGT. *Tableros electrónicos*. URL: http://www.digitalgametechnology.com/index.php/products/electronic-boards (visitado 02-2020).

[DGTb] DGTShop. *Tienda online de DGT*. URL: https://www.dgtshop.nl/ (visitado 05-2020).

[Din16] J. Ding. *ChessVision: Chess Board and Piece Recognition*. 2016. URL: https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf (visitado 06-2019).

[Edw94] S. J. Edwards. *Portable game notation specification and implementation guide: Forsyth-Edwards notation*. 1994.

[EA10] A. de la Escalera y J.M. Armingol. "Automatic Chessboard Detection for Intrinsic and Extrinsic Camera Parameter Calibration". En: *Sensors (Basel, Switzerland)* 10 (mar. de 2010), págs. 2027-44. DOI: 10.3390/s100302027.

[Est+96] M. Ester, H.-P. Kriegel, J. Sander, X. Xu y col. "A density-based algorithm for discovering clusters in large spatial databases with noise". En: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. Ed. por E. Simoudis, J. Han y U. M. Fayyad. Vol. 96. 1996, págs. 226-231.

[Gao+17] F. Gao y col. "Dual-branch deep convolution neural network for polarimetric SAR image classification". En: *Applied Sciences* 7 (abr. de 2017), pág. 447.

[Goo] Google. *Coral*. URL: https://coral.ai/products/ (visitado 05-2020).

[HS] J. Ho y R. Smith. *NVIDIA Tegra X1 preview and architecture analysis*. Ed. por AnandTech. URL: https://www.anandtech.com/show/8811/nvidia-tegra-x1-preview/2 (visitado 05-2020).

[Hua+17] G. Huang, Z. Liu, L. Van Der Maaten y K. Q. Weinberger. "Densely connected convolutional networks". En: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). DOI: 10.1109/cvpr.2017.243.

[Ian+16] F. N. Iandola y col. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. 2016. arXiv: 1602.07360.

[Int] Intel. *Neural Compute Stick 2*. URL: https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html (visitado 05-2020).

[Kim+19] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida y N. Bagherzadeh. "Efficient Mitchell's approximate log multipliers for convolutional neural networks". En: *IEEE Transactions on Computers* 68.5 (2019), págs. 660-675. DOI: 10.1109/TC.2018.2880742.

[KSH12] A. Krizhevsky, I. Sutskever y G. E. Hinton. "Imagenet classification with deep convolutional neural networks". En: *Advances in neural information processing systems*. 2012, págs. 1097-1105.

[Low04] D. Lowe. "Distinctive image features from scale-invariant keypoints". En: *International Journal of Computer Vision* 60 (2004), págs. 91-110.

[MGK00] J. Matas, C. Galambos y J. Kittler. "Robust detection of lines using the progressive probabilistic Hough transform". En: *Computer Vision and Image Understanding* 78.1 (2000), págs. 119-137.

[Mat+11] C. Matuszek y col. “Gambit: An autonomous chess-playing robotic system”. En: mayo de 2011, págs. 4291-4297. DOI: 10.1109/ICRA.2011.5980528.

[Mic] Microsoft. *ONNXRuntime*. URL: https://github.com/microsoft/onnxruntime/.

[MDB20] R. Murillo, A. A. Del Barrio y G. Botella. “Deep PeNSieve: A deep learning framework based on the posit number system”. En: *Digital Signal Processing* 102 (2020), pág. 102762. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2020.102762.

[Nin] Nintendo. *Switch Technical Specs*. URL: https://www.nintendo.com/switch/tech-specs/ (visitado 05-2020).

[Nvia] Nvidia. *Jetson Embedded Systems*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (visitado 05-2020).

[Nvib] Nvidia. *Jetson Nano*. URL: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-nano/ (visitado 05-2020).

[Nvic] Nvidia. *Jetson Xavier NX*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/ (visitado 05-2020).

[Nvid] Nvidia. *Shield TV*. URL: https://www.nvidia.com/en-us/shield/ (visitado 05-2020).

[Nvie] Nvidia. *TensorRT*. URL: https://developer.nvidia.com/tensorrt (visitado 05-2020).

[Oli06] Travis Oliphant. *NumPy: A guide to NumPy*. Trelgol Publishing USA. 2006. URL: http://www.numpy.org/.

[ONN] ONNX. *Open Neural Network Exchange*. URL: https://github.com/onnx/onnx.

[RW17] W. Rawat y Z. Wang. “Deep convolutional neural networks for image classification: A comprehensive review”. En: *Neural Computation* 29.9 (2017), págs. 2352-2449. DOI: 10.1162/neco_a_00990.

[Rez04] A. M. Reza. “Realization of the contrast limited adaptive histogram equalization (CLAHE) for real-time image enhancement”. En: *Journal of VLSI signal processing systems for signal, image and video technology* 38 (2004), págs. 35-44.

[RWC01] G. van Rossum, B. Warsaw y N. Coghlan. *PEP 8 - Style guide for python code*. 2001. URL: https://www.python.org/dev/peps/pep-0008/.

[Rus+15] O. Russakovsky y col. “ImageNet Large Scale Visual Recognition Challenge”. En: *International Journal of Computer Vision (IJCV)* 115.3 (2015), págs. 211-252. DOI: 10.1007/s11263-015-0816-y.

[San+18] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov y L.-C. Chen. "MobileNetV2: Inverted residuals and linear bottlenecks". En: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018). DOI: 10.1109/cvpr.2018.00474.

[Sch19] C. Schubiner. *Chessboard image to FEN*. 2019. URL: https://github.com/cschubiner/chessboard-image-to-fen.

[SP10] D. Sen y S. K. Pal. "Gradient histogram: Thresholding in a region of interest for edge detection". En: *Image and Vision Computing* 28.4 (2010), págs. 677-695.

[Set+07] J. Setoain, M. Prieto, C. Tenllado, A. Plaza y F. Tirado. "Parallel Morphological Endmember Extraction Using Commodity Graphics Hardware". En: *IEEE Geoscience and Remote Sensing Letters* 4.3 (2007), págs. 441-445.

[Squ] SquareOff. *Tablero de ajedrez inteligente*. URL: https://www.squareoffnow.com/ (visitado 02-2020).

[Sze+17] V. Sze, Y.-H. Chen, T.-J. Yang y J. S. Emer. "Efficient processing of deep neural networks: A tutorial and survey". En: *Proceedings of the IEEE* 105.12 (2017), págs. 2295-2329. DOI: 10.1109/jproc.2017.2761740.

[Ten+08] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel y F. Tirado. "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting". En: *IEEE Transactions on Parallel and Distributed Systems* 19.3 (2008), págs. 299-310.

[The97] The HDF Group. *Hierarchical Data Format, version 5*. 1997. URL: http://www.hdfgroup.org/HDF5/ (visitado 05-2020).

[Wie96] D. P. Wiens. "Asymptotics of generalized M-estimation of regression and scale with fixed carriers, in an approximately linear model". En: *Statistics and Probability Letters* 30.3 (1996), págs. 271-285.

[Yan16] D. Yang. *Chess ID*. 2016. URL: https://github.com/daylen/chess-id.

[Zop+18] B. Zoph, V. Vasudevan, J. Shlens y Q. V. Le. "Learning transferable architectures for scalable image recognition". En: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018). DOI: 10.1109/cvpr.2018.00907.