Software de reconocimiento de piezas de ajedrez en tiempo real

Real-time chess piece recognition software

Ángel Molina Núñez
Carlos Plaza García-Abadillo
Katya Rengel Lazcano

Dirigido por Manuel Prieto Matías y Alberto del Barrio Garcia
Facultad de informática
Universidad Complutense de Madrid
Madrid, 2021-2022

Índice

Índice de figuras

Índice de cuadros

Lista de algoritmos

Dedicatoria

El siguiente trabajo se lo dedico a mis padres y a mi pareja que me han ayudado siempre y, en especial, durante mi etapa como estudiante apoyándome y ayudándome a llegar donde estoy.

Ángel Molina Núñez

A mi familia y amigos por haberme apoyado incondicionalmente durante esta dura etapa como estudiante.

Carlos Plaza García-Abadillo.

Este trabajo se lo dedico a mi madre que siempre fue, es y será mi ejemplo para seguir, a mis hermanos por todos sus consejos, a mi esposo por su apoyo y a mis hijos por haber tenido tanta paciencia y haber cedido su tiempo para que pueda conseguir una meta que en un principio fue algo personal y terminó siendo un logro de familiar.

Katya Alejandra Rengel Lazcano

Agradecimientos

Estamos muy agradecidos con todos los profesores que nos han ayudado durante estos años y en especial con los tutores de este proyecto Manuel Prieto Matías y Alberto Antonio del Barrio García por su extensa ayuda durante este año y su control continuo del proyecto a pesar de la situación de pandemia que hemos sufrido.

Asimismo agradecemos el trabajo realizado por David Mallasén Quitana por su colaboración en el inicio de este proyecto y de su proyecto pasado.

Resumen

El reconocimiento de las piezas de ajedrez en un tablero físico es un problema de visión que aún no se ha resuelto de manera eficiente. Se han logrado algunos avances a lo largo de los años, pero aún carecen de la precisión o el rendimiento suficientes para usarse a nivel práctico para digitalizar juegos en vivo. Este proceso de digitalización actualmente se realiza a nivel profesional con la ayuda de tableros de ajedrez y piezas especializadas que no están al alcance de la mayoría de los aficionados al ajedrez debido a su costo.

Hay un interés renovado en encontrar nuevos algoritmos para este problema, gracias al éxito de las redes neuronales convolucionales. Sin embargo, aún existen varias dificultades para su adopción práctica, especialmente en implementaciones que intentan satisfacer las restricciones de tiempo real de los juegos en vivo con recursos de hardware de presupuesto limitado.

En nuestro TFG, hemos tratado de superar algunas de estas limitaciones portando a una aplicación de Android algunos de los algoritmos más prometedores propuestos en este campo. Además, también hemos intentado mejorar el entrenamiento de las redes neuronales diseñando e implementando un dispositivo de bajo coste que simplifica la generación del dataset de entrenamiento.

Palabras clave: Arduino, Ajedrez, Visión artificial, Deep learning, Redes neuronales convolucionales.

Abstract

The recognition of chess pieces within a physical board is a vision problem that has not yet been solved efficiently. Some progress has been made over the years, but they still lack enough precision and or performance to be used at a practical level to digitize live games. This digitization process is currently performed at a professional level with the help of chessboards and specialized pieces that are not available to most chess enthusiasts due to their cost.

There is a renewed interest in finding new algorithms for this problem, thanks to the success of convolutional neural networks. However, there are still several difficulties for its practical adoption, especially in implementations that try to satisfy the real-time constraints of live games with budget-limited hardware resources.

In our TFG, we have tried to overcome some of these limitations by porting to an android app some of the most promising algorithms proposed in this field. In addition, we have also tried to improve the training of the neuronal networks by designing and implementing a low-cost device that simplifies the generation of the training dataset.

Keywords: Arduino, chess, Artificial Vision, Deep learning, Convolutional Neural Networks.

1. Introducción y Motivación del proyecto

La digitalización por imágenes de partidas de ajedrez es un problema aún sin resolver satisfactoriamente en inteligencia artificial. Esta técnica puede ser útil para la retransmisión en línea de partidas, tanto a nivel profesional donde se pueden retransmitir de forma digital partidas en vivo, o incluso competir contra motores de ajedrez pudiendo tener la opción de utilizar un tablero físico.

Actualmente una de las alternativas creadas son los tableros electrónicos como los elaborados por DGT [DGT] que funcionan como dispositivos de entrada capaces de detectar la posición de sus piezas en el tablero. Unos de los inconvenientes de éstos es su elevado precio, ya que pueden llegar a costar hasta 500€ sin ningún accesorio, por lo que no son una opción para muchas personas. Además de que estos tableros solo funcionan con sus propias piezas y con su software propio, lo cual limita al jugador.

Otra opción interesante son los robots automáticos que son capaces de jugar con un tablero físico, actualmente algunos de ellos se han hecho con software libre de manera que sean capaces de funcionar con computadoras como la conocida Raspberry Pi a unos precios económicos [Mey]. Por desgracia también tiene una serie de inconvenientes, como que no es capaz de empezar un juego desde una partida intermedia, o que para la utilización y creación de robot es necesario unos conocimientos de programación y robótica avanzados que muy pocas personas poseen, por lo que no es accesible para la mayoría.

La visión artificial ofrece una alternativa cada vez más atractiva para este tipo de problemas debido a su bajo coste y a poder ser implementada en gran cantidad de dispositivos. Para ser concretos mediante la elaboración de aplicaciones de inteligencia artificial se podría conseguir que cualquier computadora, desde PCs de sobremesa utilizando una webcam, hasta teléfonos móviles de uso doméstico fueran capaces de realizar estas tareas, actualmente solo realizables mediante costosos tableros, lo cual afecta notablemente a la realización de torneos y campeonatos de esta práctica que es el ajedrez.

Además, el uso de esta tecnología permitiría que pudiéramos recordar partidas realizadas de forma física e incluso de poder entrenar o competir contra un módulo (engine) de ajedrez sin necesidad de un monitor donde mostrar el tablero, dando así la opción al jugador de usar un tablero convencional de ajedrez.

La mayoría de proyectos que utilizan esta tecnología están especializados para funcionar únicamente bajo unos parámetros previos como la posición exacta de la

cámara o con el tablero específico con el que se entrenó para dotarle de visión artificial. No obstante actualmente se han creado versiones más versátiles que son capaces de lidiar con este inconveniente y obtener resultados precisos, aunque ninguno todavía ha sido capaz de funcionar en entornos Android (debido a que no existen conversiones de todas las librerías o frameworks de TensorFlow de Python [Teab] imprescindibles para la realización y conversión de imágenes 2D de tableros a formato de lectura para computadores, también conocido como terminología FEN [For]).

Es por ello por lo que vemos una necesidad en la elaboración de este proyecto con el fin de que sea de utilidad a futuros desarrolladores y personas de todo el mundo a la hora de encontrar alternativas a las opciones actuales.

El antecedente de nuestro trabajo es LiveChess2FEN, un framework desarrollado por David Mallasén Quintana como parte de su Trabajo de Fin de Grado en el curso 2019-20 [Dav; Mal] el cual incluye numerosas investigaciones en proyectos relacionados con la detección de tableros y piezas de ajedrez.

Una limitación práctica de LiveChess2FEN es que está implementado íntegramente en Python y utiliza una Nvidia Jetson Nano como plataforma [Nvi], por lo que todo su código, tanto los algoritmos utilizados por el como los autogenerados por los frameworks utilizados no se pueden utilizar para crear aplicaciones móviles. Además, el código disponible de LiveChess2FEN tampoco esta estructurado para funcionar directamente mediante vídeo de tal forma que se puedan capturar imágenes de la misma forma que se analizan según transcurra una partida. Por último, el entrenamiento de las redes neuronales utilizadas en LiveChess2FEN se realizó con una muestra de imágenes pequeña, ya que no hay públicamente disponibles ninguna base de datos de piezas de ajedrez bien etiquetadas. Mejorar el entrenamiento con una base de datos más amplia es por tanto otro aspecto interesante a explorar.

1.1. Objetivos

El entrenamiento de redes neuronales necesita para ser efectivo del entrenamiento con grandes cantidades de datos preprocesados que permiten la inferencia de parámetros. Este entrenamiento es muy costoso, pero afortunadamente, hay disponibles una gran cantidad de redes preentrenadas que podemos utilizar. No obstante, estas redes no ofrecen una solución satisfactoria para el problema concreto que planteamos, en nuestro caso la inferencia de piezas de ajedrez en una imagen.

En estos casos, suele utilizarse transferencia de aprendizaje (transfer learning),

una metodología habitual en Deep Learning en la que en lugar de entrenar redes desde cero, se utiliza como punto de partida una red previamente entrenada para una tarea, en nuestro caso de reconocimiento de objetos. Es en este *segundo* aprendizaje donde se necesita una base de datos de piezas de ajedrez etiquetadas. Sin embargo, las muestras que hay de piezas de ajedrez en bases de datos públicas son muy limitadas. Es por esto que, nuestro primer objetivo plantea precisamente la creación de una base de datos propia la cual tome imágenes desde varios ángulos de cada una de las piezas que forman el tablero. Para simplificar la creación de una base de datos de gran magnitud, nos planteamos el diseño e implementación un sistema capaz de tomar rápidamente esas fotografías de forma automática.

El siguiente objetivo ha sido continuar con el trabajo desarrollado en LiveChess2FEN y adaptar y mejorar tanto los algoritmos como el resto de código de este entorno a Android. Se persigue con ello una adopción más fácil en móviles y tablets, no siendo necesario la utilización de ningún hardware específico.

El último objetivo es valorar los resultados obtenidos en plataformas móviles de gama media. LiveChess2FEN sólo se ha probado con una Nvidia Jetson Nano. Aunque los resultados en esta plataforma eran prometedores es un dispositivo moderadamente costoso y que no esta al alcance de la mayoría de potenciales usuarios.

Para facilitar la futura extensión del proyecto, todo el código fuente desarrollado se encuentra disponible en las referencias de esta memoria [Ángb] [Ánga].

1.2. Plan de trabajo y Organización de la memoria

Este es el plan de trabajo que hemos seguido para tratar de cumplir con el mayor éxito posible los objetivos anteriormente detallados.

- Desarrollar un sistema automático capaz de fotografiar piezas de ajedrez que, además, está basado en hardware y software libre utilizando una webcam, varios sensores y motores y un microcontrolador Arduino Uno [Mas]. El dispositivo se conecta mediante 2 puertos USB a un ordenador con sistema Linux [Tor] y, tras colocar una pieza, comienza a hacer fotografías desde diferentes ángulos con el fin de cubrir el mayor número de posibilidades. Con este dispositivo se automatiza la generación de las bases de datos de entrenamiento.
- Utilizar, en medida de lo posible, los desarrollos del proyecto LiveChess2FEN, para portar dicho framework a una aplicación Android.

La memoria del proyecto está dividida en 10 apartados siendo esta introducción el primero de ellos. Durante los próximos apartados desarrollaremos el proyecto en el mismo orden que el plan de trabajo anteriormente mencionado. En cuanto a los apartados 8 y 9 son las traducciones al inglés de la 'Introducción' y las 'Conclusiones' respectivamente. Finalmente en el apartado 10 desarrollaremos las contribuciones de cada uno de los miembros al proyecto.

2. ChessPieceScanner

El entrenamiento de redes neuronales se basa en grandes cantidades de datos preprocesados que permiten la inferencia de parámetros. En internet podemos encontrar una gran cantidad de redes preentrenadas que podemos utilizar, no obstante, pueden no ofrecer una solución para el problema que planteamos, en nuestro caso la inferencia de piezas de ajedrez en una imagen.

Es por esto que, hemos decidido desarrollar un sistema semiautomático para fotografiar piezas de ajedrez que, además, está basado en hardware y software libre utilizando una webcam y un microcontrolador Arduino Uno [Mas].

El dispositivo se conecta mediante 2 puertos USB a un computador con sistema Linux y, tras colocar una pieza, comienza a hacer fotografías desde diferentes ángulos con el fin de cubrir el mayor número de posibilidades.

Para facilitar la réplica de este sistema por parte de otros desarrolladores interesados en el proyecto, hemos hecho el esfuerzo en esta memoria por describir con detalle como es el montaje de este dispositivo.

2.1. Planos

El dispositivo consta de dos bases circulares, una inferior de tamaño más grande que se sostiene mediante 4 patas y que sirve de base para todos los componentes y una superior sobre la que se coloca la pieza que deseemos escanear y que llamaremos eje Z (Figura 1 y 2).

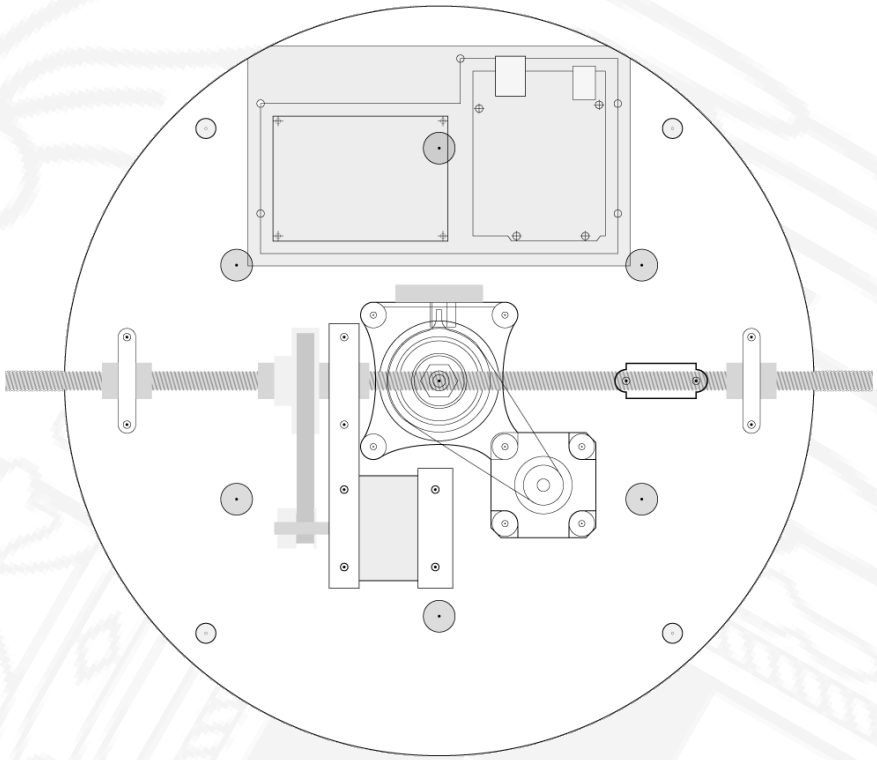


Figura 1: Base inferior del dispositivo a escala 1:3

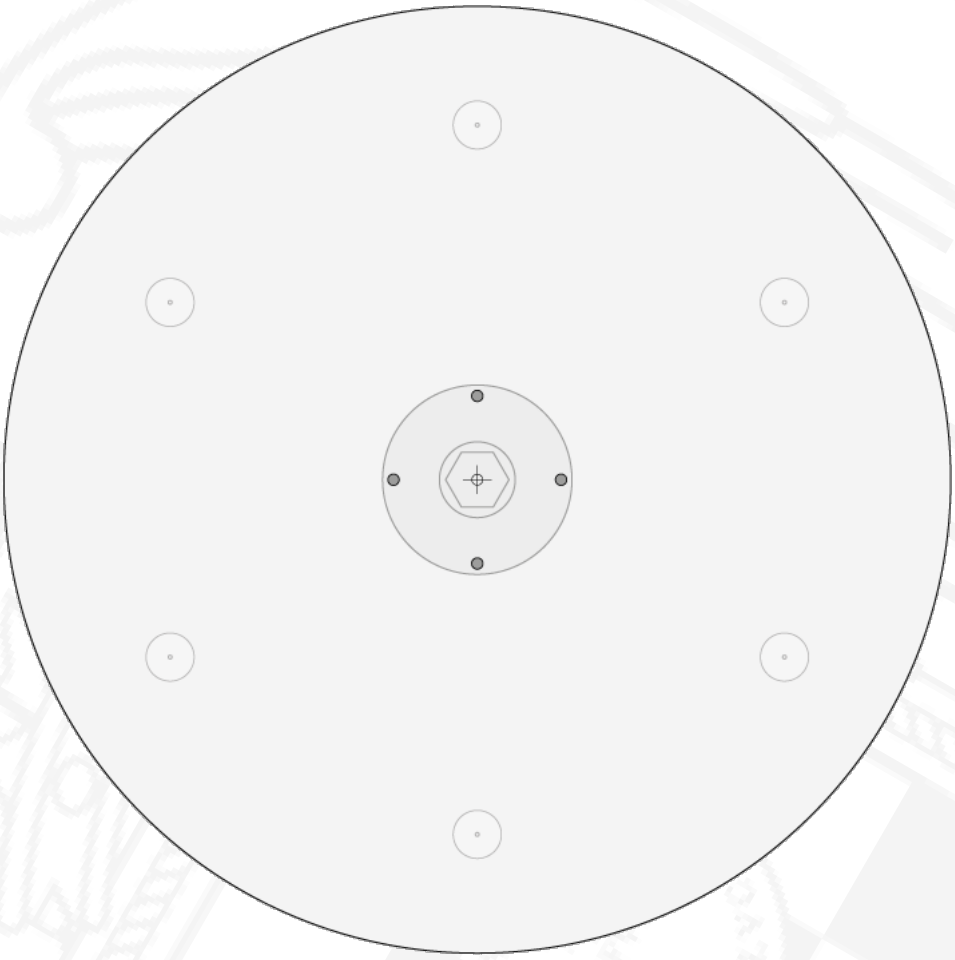


Figura 2: Base superior del dispositivo a escala 1:2

La webcam se monta en el interior de una pieza que debajo permite la instalación del servomotor que permitirá el giro de la misma. Como veremos más adelante a este eje de rotación le llamaremos eje J. Esta pieza se monta sobre otra por medio de tornillos de métrica 3 y esta otra pieza encaja con 4 varillas de metal que llegan hasta el eje que llamaremos Y que consiste en una varilla roscada de métrica 8 que está montada sobre la base inferior (Figura 3).

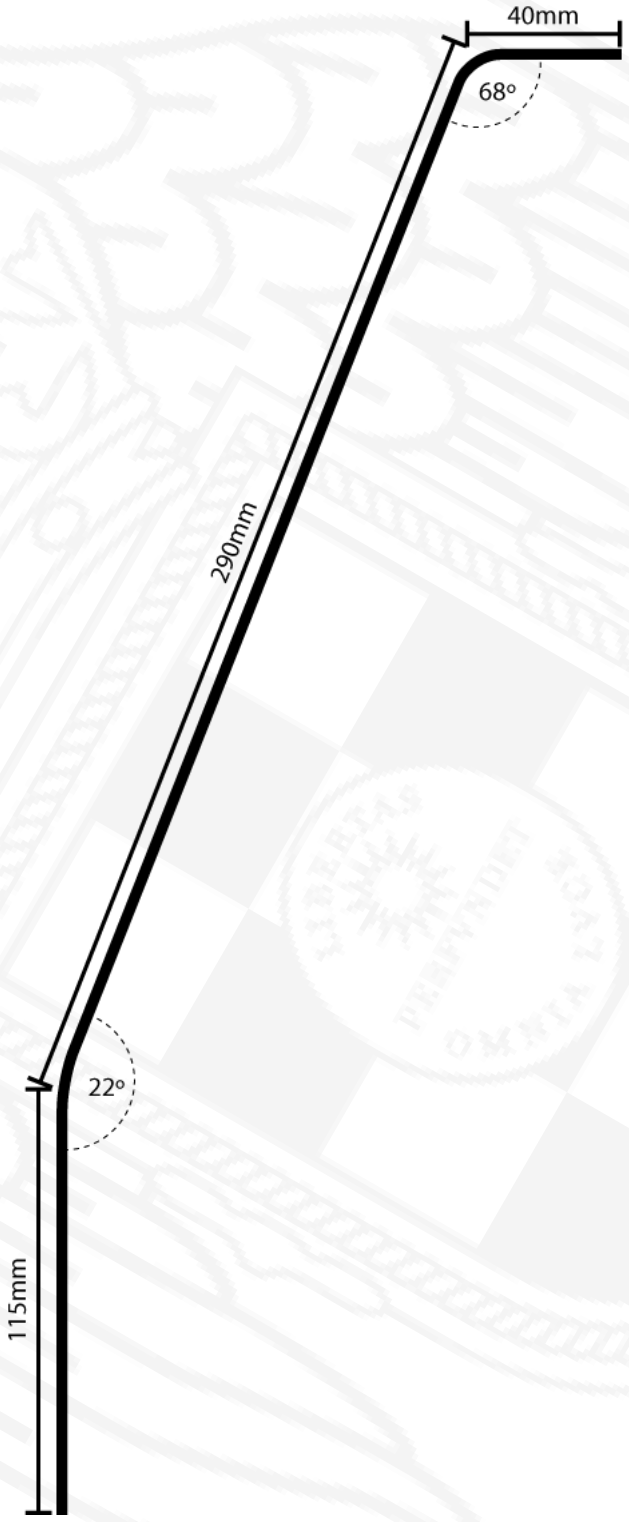


Figura 3: Varilla de metal de unión entre la webcam y la base a escala 1:2

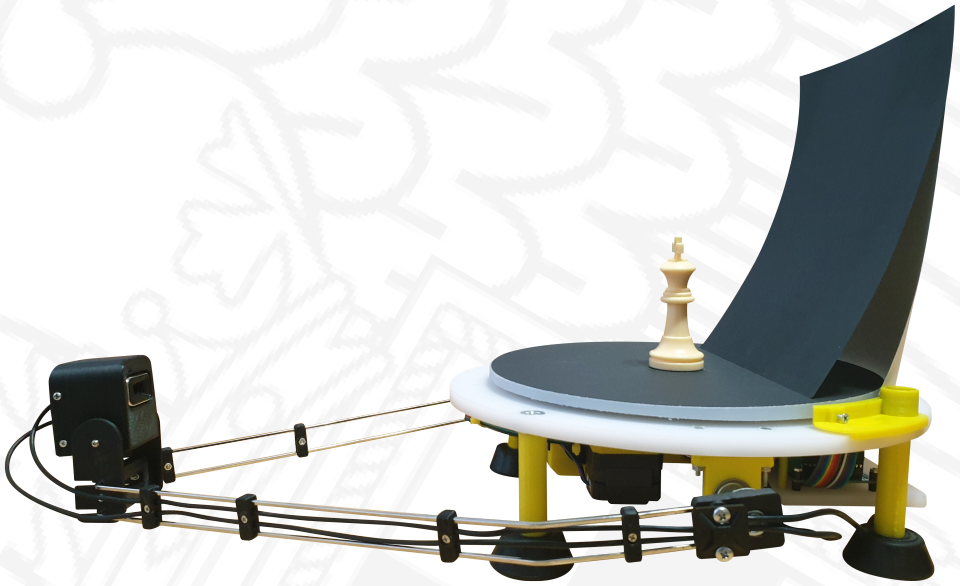


Figura 4: Lateral del dispositivo

Como mencionamos antes, existen 3 ejes de movimiento en el dispositivo, que hemos denominado eje J, Y y Z. El eje J es un eje horizontal situado en la webcam y que permite que se mantenga tangente a la circunferencia que describe el eje Y que está situado debajo de la base inferior y cuyo centro es el centro de la varilla roscada. En cuanto al eje Z, es un eje vertical que permite la rotación de las piezas.

El microcontrolador Arduino Uno y la placa de control se montan sobre un soporte específico que también se ancla a la base inferior del dispositivo (Figuras 5 y 6).

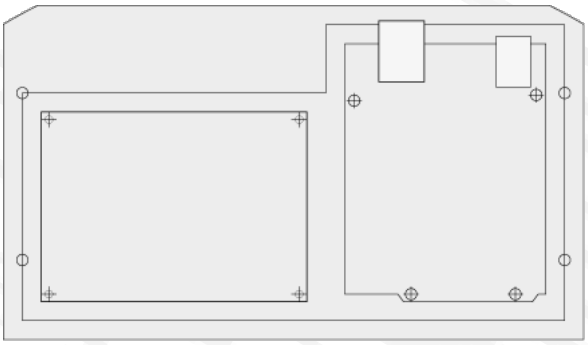


Figura 5: Soporte de Arduino y placa de control a escala 1:2

Figura 6: Soporte de Arduino y placa de control en el dispositivo

2.2. Piezas 3D y detalle del montaje

Para ensamblar todas las bases, varillas y componentes hemos desarrollado piezas que pueden ser impresas en cualquier impresora 3D. Aunque es recomendable que la impresión se realice en plástico ABS ya que es más resistente a la temperatura y después de un uso prolongado los motores pueden alcanzar temperaturas de entre 40 y 50 grados centígrados.

El montaje debe comenzar por el eje vertical o eje Z de la base inferior, cogeremos un tornillo de métrica 8 al que le colocaremos un rodamiento de diámetro interior 8mm, una arandela que actúe como separador, la polea, el cuentavueltas del eje Z (Figura 7, un rodamiento de diámetro interior 20mm y una tuerca que fije todo. Una vez tengamos este eje encajaremos el exterior del primer rodamiento en la base inferior, dejando que la cabeza del tornillo sobresalga por la parte superior, colocamos la correa en la polea y encajaremos el segundo rodamiento sobre el soporte impreso del eje (Figura 8) después colocamos el otro extremo de la correa sobre la polea del motor y lo fijamos todo con tornillos a la base.Finalmente debemos colocar el sensor óptico de final de carrera y cubrirlo con su tapa (Figura 9).

Figura 7: Cuentavueltas del eje Z

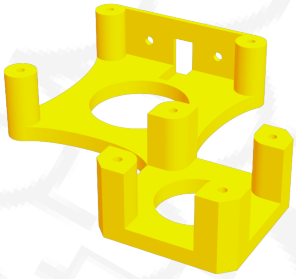


Figura 8: Soporte del eje Z

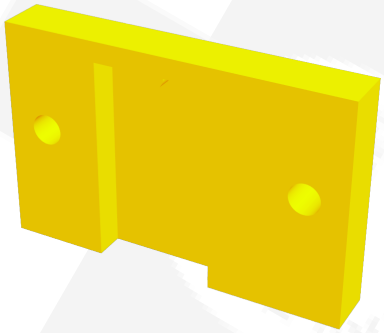


Figura 9: Tapa del sensor óptico de final de carrera del eje Z

Continuaremos después colocando el segundo sensor óptico de final de carrera, el correspondiente al eje Y, para ello necesitaremos el soporte impreso (Figura 10).

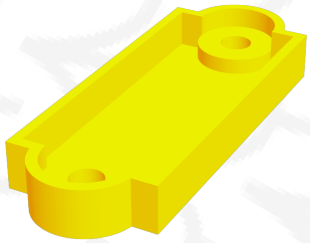


Figura 10: Soporte del sensor óptico de final de carrera del eje Y

Seguidamente montamos el eje horizontal o eje Y de la base inferior, necesitaremos una varilla roscada de métrica 8 sobre la que pondremos una polea con una tuerca a cada lado para impedir el movimiento y después encajaremos sobre esta un rodamiento de diámetro interior 20mm. Luego montaremos el rodamiento sobre el soporte impreso del eje (Figura 11), al que también colocaremos el motor, y uniremos las poleas mediante la correa. En este punto la correa está destensada, pero no debemos preocuparnos. Ahora colocamos el cuentavueltas del eje Y (Figura 12) fijándolo a la varilla con una tuerca a cada lado y haciéndolo coincidir con el sensor que hemos colocado previamentete.

En este momento podemos colocar los soportes laterales (Figura 13) del eje -también con una tuerca en cada uno de sus lados- a los que previamente debemos colocar un rodamiento de diámetro interior 8 y los fijamos a la base colocando los separadores de estos soportes (Figura 14).

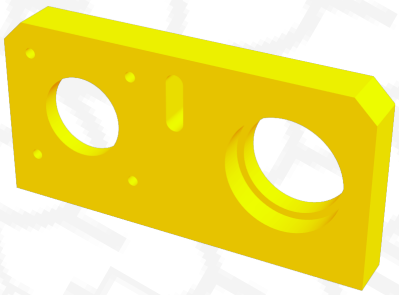


Figura 11: Soporte del eje Y

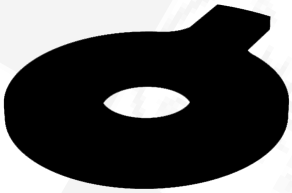


Figura 12: Cuentavueltas del eje Y

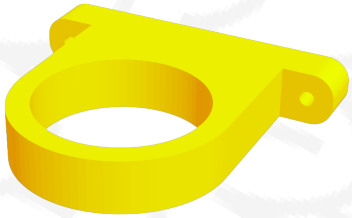


Figura 13: Soporte lateral del eje Y

Figura 14: Separador del soporte lateral del eje Y

Una vez tengamos todo fijado es el momento de tensar la correa, para ello con la ayuda de un tornillo de métrica 6 colocamos un rodamiento en la ranura vertical que ha quedado libre del soporte del eje Y y lo arrastramos hacia abajo hasta que la correa quede tensa, una vez eso ocurra, lo apretamos.

En este momento podemos retirar la base inferior y ensamblar la parte de la webcam, para ello introducimos las varillas en el soporte de la webcam (Figura 15) y con ayuda de unos tornillos las apretamos para que queden fijas. Una vez hecho

eso, fijamos al soporte los laterales (Figuras 16 y 17) con ayuda de otros 2 tornillos e introduciendo los alambres por las guías. Después introducimos la webcam en la caja (Figura 18) y pasamos el cable por el interior del muelle, también introducimos el servomotor y lo fijamos con otros 2 tornillos a la caja. Más tarde colocamos la tapa trasera (Figura 19) teniendo cuidado de colocar el muelle en su sitio y habiendo pasado los cables de la webcam y el servomotor por el hueco central y colocado la tapa de los cables (Figura 20). Finalmente atornillamos la caja a los soportes laterales.

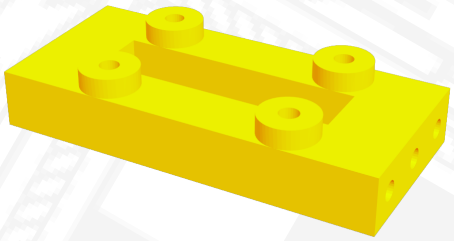


Figura 15: Soporte de la webcam

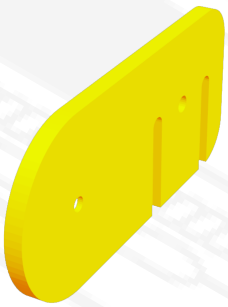


Figura 16: Soporte lateral derecho de la webcam

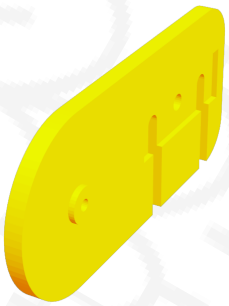


Figura 17: Soporte lateral izquierdo de la webcam

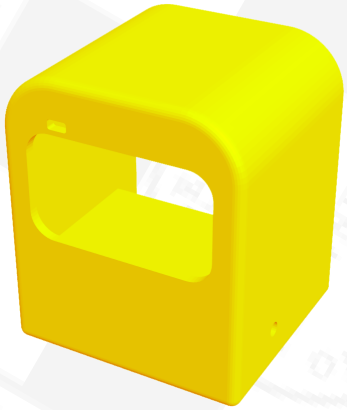


Figura 18: Caja de la webcam

Figura 19: Tapa trasera de la webcam

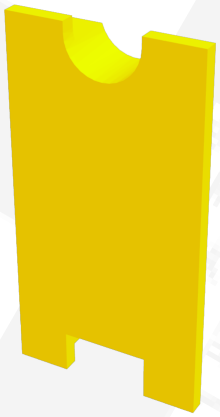


Figura 20: Tapa de los cables de la webcam

Llegados a este punto podemos colocar los soportes de los otros extremos de las varillas que nos permitirán ensamblarlas con la varilla roscada de la base inferior o eje Y (Figura 21).

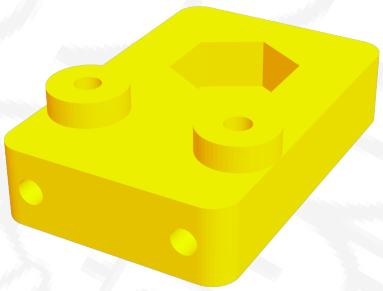


Figura 21: Soporte de ensamble de las varillas de la webcam y el eje Y

Para ensamblar ahora estas piezas al eje Y necesitaremos introducir una tuerca y una arandela en cada extremo de la varilla roscada hasta que choquen con las tuercas de los cojinetes. Después introducimos la varilla roscada por los agujeros de las piezas hasta que estas choquen con las tuercas. Una vez aquí ponemos otras dos tuercas hasta enrasarlas con los bordes de la varilla roscada y tiramos de los soportes hacia fuera hasta que las tuercas exteriores se embutan en el hueco. Finalmente aflojamos las primeras tuercas que introdujimos hasta que los soportes queden fijos.

Ahora debemos posicionar las varillas perfectamente paralelas a la base inferior, aflojar las tuercas del cuentavueltas del eje Y, colocar el cuentavueltas de modo que el sensor óptico quede tapado y volver a apretar las tuercas.

A continuación, colocamos el soporte trasero del motor del eje Y (Figura 22) para evitar que el motor cabecee por la tensión de la correa.

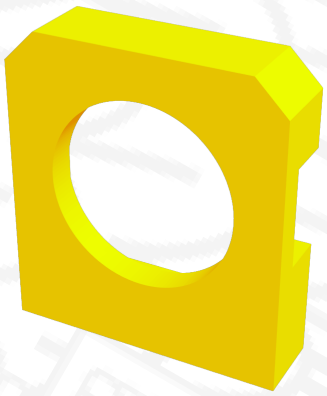


Figura 22: Soporte trasero del motor del eje Y

En este momento podemos montar las patas de la base inferior cubriéndolas con los embellecedores (Figura 23) y la base del microcontrolador y la placa de control haciendo uso de los separadores (Figura 24). Asimismo, podemos guiar los cables de la webcam y del servomotor por las varillas utilizando las guías laterales para cables (Figura 25), de este modo evitamos que con el movimiento puedan enredarse los cables provocando daños en el dispositivo y/o el computador.

Figura 23: Embellecedores de las patas

Figura 24: Separadores del soporte del microcontrolador y placa controladora

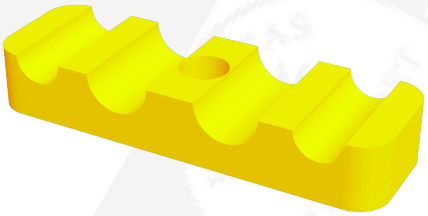


Figura 25: Guía lateral para cables

Ensamblamos el soporte del fondo, para ello necesitaremos pegar con ayuda de un pegamento acrílico las piezas superior e inferior de los soportes izquierdo y derecho (Figuras 26, 27, 28 y 29). Después introducimos 2 pequeñas pletinas que apretamos con ayuda de un tornillo en cada extremo y colocamos el papel entre ellas para que quede recto. Finalmente debemos introducir unos imanes en los huecos laterales que harán que se peguen a los grandes tornillos que sujetan las patas y evitarán que nuestro fondo se caiga.

Figura 26: Soporte superior derecho del fondo

Figura 27: Soporte inferior derecho del fondo

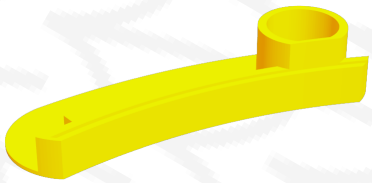


Figura 28: Soporte superior izquierdo del fondo

Figura 29: Soporte inferior izquierdo del fondo

Finalmente colocamos las esferas de deslizamiento en la parte superior de la base inferior y atornillamos a la base superior el soporte de giro (Figura 30).

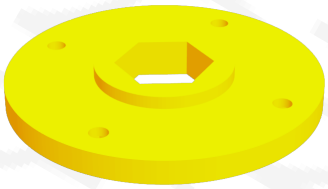


Figura 30: Soporte de giro de la base superior

Ahora tenemos nuestro dispositivo listo para funcionar. Véase la figura inferior para ver el resultado del dispositivo elaborado.

Figura 31: Vista general del dispositivo

2.3. Materiales utilizados

- Polietileno de color blanco para las bases superior e inferior
- Patas de compresor doméstico de aire acondicionado
- Filamento ABS para la impresión de las piezas 3d
- Varillas metálicas de métrica 4
- Varilla roscada de métrica 8
- Tornillería de métricas 3mm, 6mm y 8mm
- 2 rodamientos 6804ZZ
- 2 rodamientos F680ZZ
- 3 rodamientos ABEC-1
- 6 bolas transportadoras de acero de 8mm
- 2 juegos de poleas GT2 reducción 3:1
- 2 bridas metálicas que actúan como pletinas

En un primer momento consideramos utilizar tableros de madera MDF como material para las bases superior e inferior y como base para el microcontrolador y la placa controladora, pero este material fue descartado por resultar demasiado flexible y al tensar la correa del eje Y se producía una ligera curvatura sobre la base. El polietileno sin embargo es un material más rígido que permite tensar la correa sin que se deforme.

2.4. Hardware

- Arduino Uno rev.3
- Motor NEMA-17 de 40mm
- Motor NEMA-14 de 28mm
- 2 drivers A4988
- 2 sensores ópticos de final de carrera
- 1 servomotor HD-1160A
- Placa de prototipado
- 5 resistencias de 1kΩ
- 1 resistencia de 330Ω
- 1 resistencia de 560Ω
- 1 condensador electrolítico de 200uF
- 2 diodos led
- 11 jumper
- Transformador de 12V y 1.5A

El microcontrolador se comunica con el computador por medio de un cable USB a través de un puerto serie con una velocidad de 115.200 baudios. La comunicación se realiza por medio de comandos, una vez recibido el comando y realizada la acción se envía un ACK a la computadora para indicar que se ha completado la acción. Podemos ver los comandos disponibles en el cuadro 1.

Comando	**Descripción**
cps0	Imprime una breve descripción del objetivo del dispositivo.
cps100$1	Establece la posición actual del eje $1 del dispositivo como la posición inicial. • $1 puede tomar los valores ‘j’, ‘y’ ó ‘z’.
cps101$1	Calibra automáticamente el eje $1 del dispositivo. • $1 puede tomar los valores ‘j’, ‘y’ ó ‘z’.
cps200$1	Libera el motor del eje $1 cesando la entrega de corriente. • $1 puede tomar los valores ‘j’, ‘y’ ó ‘z’.
cps201$1	Mueve el eje $1 hasta la posición $2 en grados. • $1 puede tomar los valores ‘j’, ‘y’ ó ‘z’. • $2 puede tomar los valores [0, 120] si $1 es ‘j’, [0, 90] si $1 es ‘y’ ó [0, 360) si $1 es ‘z’.
cps300$1	Devuelve el ángulo actual del eje $1 en grados. • $1 puede tomar los valores ‘j’, ‘y’ ó ‘z’.
cps301$1	Devuelve el estado del dispositivo óptico de final de carrera del eje $1. • $1 puede tomar los valores ‘y’ ó ‘z’.
cps400$1	Establece el tiempo de espera entre pulsos del motor del eje $1. • $1 puede tomar los valores ‘y’ ó ‘z’.
cps401$1	Establece el número de pasos que debe avanzar o retroceder el motor del eje $1 para moverse 1 grado. • $1 puede tomar los valores ‘y’ ó ‘z’.

Cuadro 1: Tabla de comandos disponibles del microcontrolador

El microcontrolador se conecta a todos los demás componentes hardware a través de la placa controladora, en la que también están los drivers de los motores paso a paso. Esta placa además tiene una entrada de 12V para la alimentación de los motores y obtiene los 5V de alimentación del microcontrolador. La placa controladora dispone de 5 puertos híbridos que pueden ser usados para sensores y/o servomotores, cada puerto dispone de un cable de alimentación VDD, otro de tierra GND y otro de datos que puede ser usado como entrada o como salida. El diagrama de conexiones de la placa controladora está disponible en la (Figura 32).

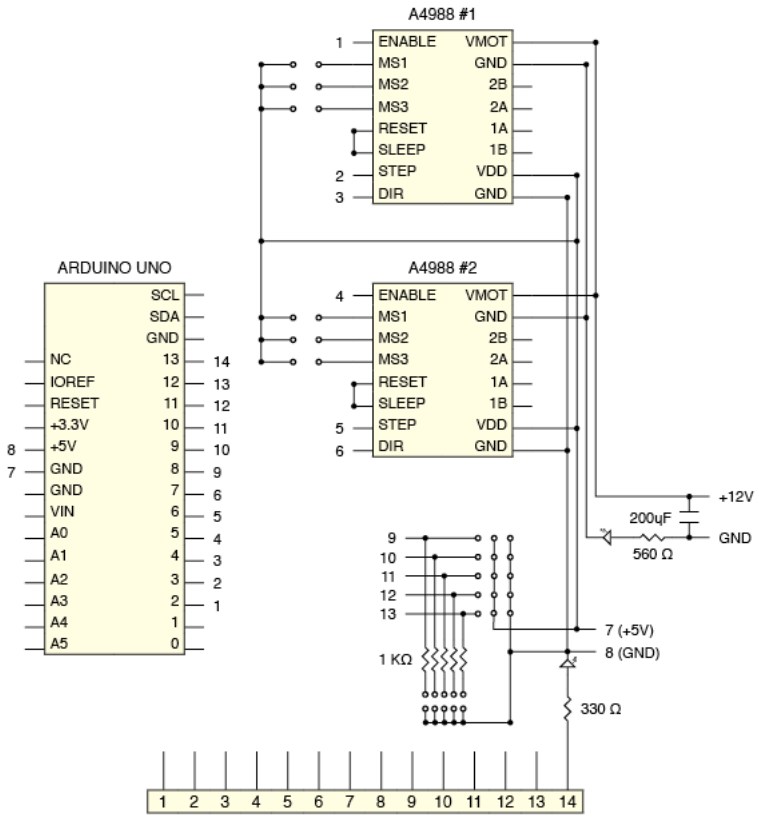


Figura 32: Diagrama de conexiones de la placa controladora

Antes de conectar los motores debemos calibrar los drivers para que entreguen la corriente necesaria al motor y que no resulte dañado. Para ello necesitamos conocer la corriente de funcionamiento para cada uno de nuestros motores, en nuestro caso tenemos un motor paso a paso del tipo NEMA-17 de 1,5A y otro del tipo NEMA-14 de 0,8A. Para calcular el V_{ref} (voltaje de referencia) del driver aplicamos la siguiente fórmula:

$$V_{ref} = I_{max} * (8 * r_s)$$

En nuestro caso, para el primer motor tenemos una I_{max} (intensidad máxima) de 1,5A y para el segundo una I_{max} de 0,8A. Ambos controladoras tienen una resistencia de sensibilidad de 0,1Ω por lo que este valor será constante en nuestros cálculos.

Sustituyendo los valores de I_{max} y r_s en la fórmula obtenemos los siguientes valores de V_{ref}. Para el primer motor, con I_{max} 1,5A:

$$V_{ref} = 1,5A * (8 * 0,1\Omega) = 1,2V$$

Para el segundo motor, con I_{max} 0,8A:

$$V_{ref} = 0,8A * (8 * 0,1\Omega) = 0,64V$$

Una vez obtenidos los voltajes de referencia para el funcionamiento de los motores debemos comprobar en qué modo va a trabajar nuestro motor, pasos completos, medios pasos, etc. y adecuar el voltaje según indique el fabricante en la hoja de datos de nuestro motor paso a paso. Para nuestro primer motor, el fabricante incluye el cuadro 2.

Full Step	**Half Step**	**1/4 Step**	**1/8 Step**	**1/16 Step**	**Phase 1 Current**	**Phase 2 Current**
	1	1	1	1	100,00	0,00
				2	99,52	9,80
			2	3	98,08	19,51
				4	95,69	29,03
		2	3	5	92,39	38,27
				6	88,19	47,14
			4	7	83,15	55,56
				8	77,30	63,44
1	2	3	5	9	70,71	70,71

Cuadro 2: Consumo máximo según el modo de trabajo del motor

Como nuestro primer motor está trabajando con pasos de 1/16 -los tres jumpers de la placa controladora están cerrados- el V_{ref} del motor es el 100 % del V_{ref} calculado, por lo que debemos configurar el driver con ese valor. Para el segundo motor, que trabaja con pasos de 1/8 -sólo los jumpers MS1 y MS2 están cerrados- el V_{ref} del motor es aproximadamente el 83 % del V_{ref} calculado lo que nos da un nuevo valor de $V_{ref} = 0{,}53V$ con el que debemos configurar el driver.

Por defecto el programa del microcontrolador está configurado para conectar el servomotor al puerto 10 del microcontrolador, el sensor óptico de final de carrera del eje Z al puerto 11 y el del eje Y al puerto 12. No obstante podemos configurar los números de los puertos de entrada/salida desde el fichero 'configuration.h' y después cargar el programa en el microcontrolador para aplicar los cambios.

2.5. Software

El microcontrolador recibe los comandos desde el computador y lleva a cabo las operaciones necesarias, una vez acaba de ejecutar el comando responde con un ACK para informar de que ha terminado y que está listo para ejecutar el siguiente comando. Además, el microcontrolador almacena las posiciones de los ejes J, Y y Z para evitar inconsistencias y desde el computador se consulta el estado cuando se necesita.

La computadora se encarga de enviar las órdenes necesarias para la correcta colocación de los ejes y el manejo de la webcam. La comunicación con el microcontrolador se realiza por medio de un puerto serie, que en Linux se accede a través del fichero de dispositivo ubicado en '/dev/ttyUSB$1' donde $ toma un valor numérico comprendido en (0, ∞) y nos comunicamos leyendo y escribiendo como si se tratara de un fichero regular, aunque para la configuración del puerto (paridad, bits, control de flujo, velocidad, etc.) utilizamos la estructura termios -incluida en 'termios.h'- que permite establecer todos estos parámetros.

Por otro lado, la comunicación con la webcam se realiza a través de otro fichero de dispositivo, ubicado en '/dev/video$ 1' donde $ 1 toma un valor numérico comprendido en (0, ∞). En este caso la comunicación no se ha realizado mediante la librería V4L2 de Linux que permite el manejo de dispositivos de vídeo de manera sencilla. Con esta librería podemos configurar la resolución de la cámara, el formato de las imágenes y parámetros como el contraste, la saturación, la luminosidad, el brillo, etc. En nuestro caso, establecemos el formato de captura a MJPEG por lo que la cámara conectada debe dar soporte a ese formato.

2.6. Ejemplo de ejecución

Tras conectar el microcontrolador y la webcam al computador y comprobar que se crean los archivos correspondientes en el directorio '/dev' en nuestro caso '/dev/ttyUSB0' y '/dev/video0' respectivamente podemos iniciar la aplicación con el comando '$ bin/chesspiecescanner /dev/ttyUSB0 /dev/video0'. Una vez iniciada nos aparecerá el menú principal de la aplicación (Figura 33).

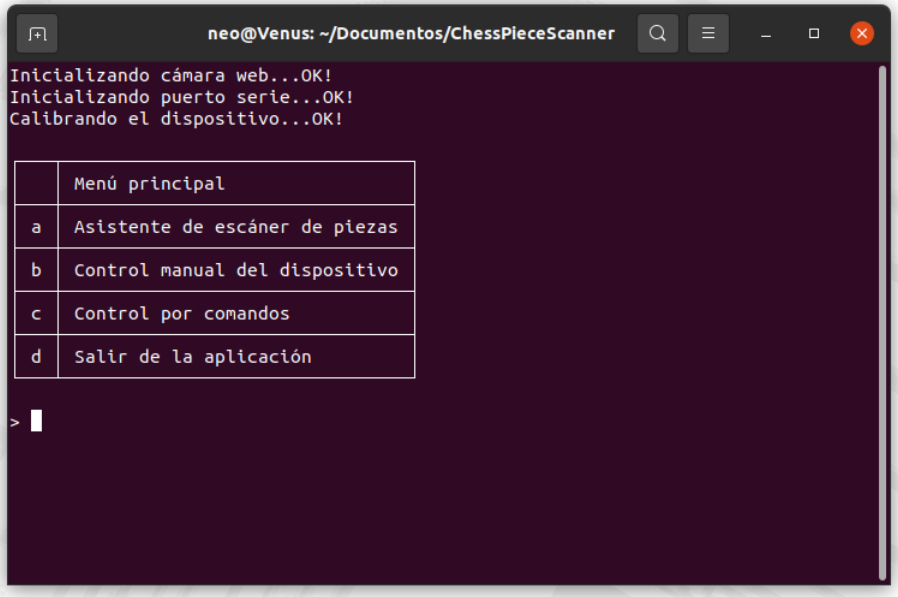


Figura 33: Menú principal de la aplicación chesspiecescanner

Desde este menú podemos seleccionar entre iniciar el 'Asistente de escáner de piezas', acceder al 'Control manual del dispositivo', acceder al 'Control por comandos' o 'Salir de la aplicación'.

Si seleccionamos la primera opción, pulsando la tecla "a" y pulsando la tecla "enter' a continuación, nos aparecerá el menú del asistente (Figura 34) desde donde podremos realizar fotografías de forma automática a una pieza. En este menú tenemos 3 opciones posibles, en la primera se realizarán fotografías moviendo únicamente los ejes J e Y del dispositivo para comprobar que las imágenes se capturan correctamente. En la segunda opción nos permite escanear 180 grados de la pieza, útil cuando las piezas son simétricas. En cuanto a la tercera opción escaneará la pieza completa. Finalmente disponemos de una cuarta opción que nos permite introducir los parámetros de escaneo manualmente.

Después de indicarle el modo de escaneo el programa nos preguntará el nombre de la pieza que va a escanear (Figura 35) y comenzará a capturar fotografías. Una vez acabe se volverá al menú principal (Figura 33) para que el usuario seleccione que desea hacer a continuación. Las imágenes se guardarán en la carpeta 'photos' dentro de un directorio con el nombre introducido anteriormente.

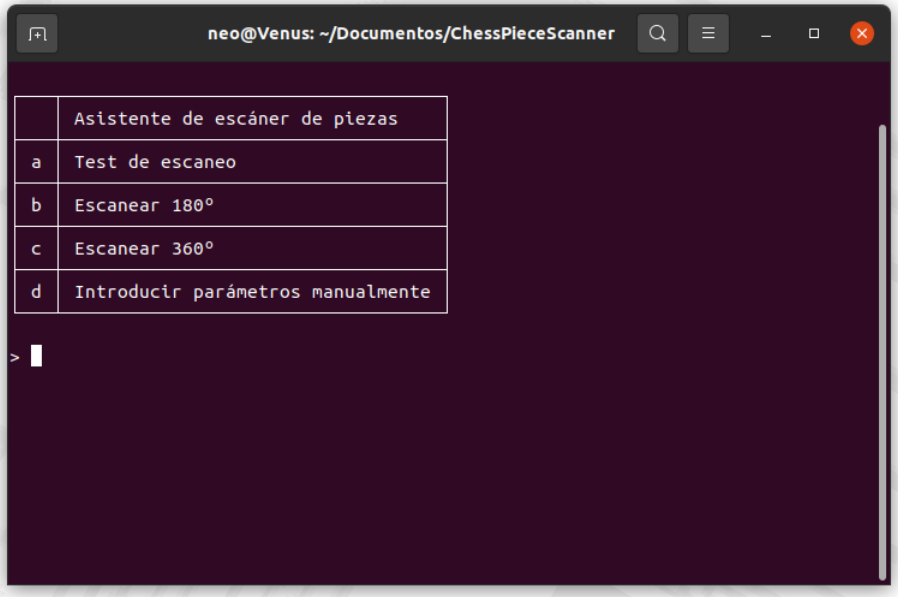


Figura 34: Menú del asistente de escáner de la aplicación chesspiecescanner

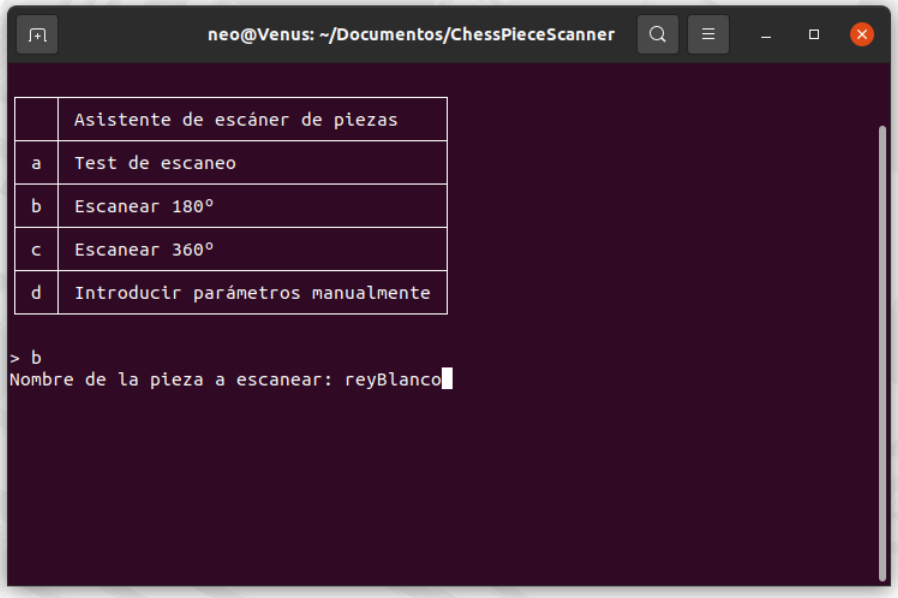


Figura 35: Solicitud del nombre de la pieza de la aplicación chesspiecescanner

Si seleccionamos la segunda opción el programa consultará al microcontrolador las posiciones de los ejes J, Y y Z y después nos mostrará una tabla con las opciones de control disponibles (Figura 36). Si realizamos una fotografía con la letra 'P' se guardará en el directorio 'photos' con el nombre de 'preview.jpg'.

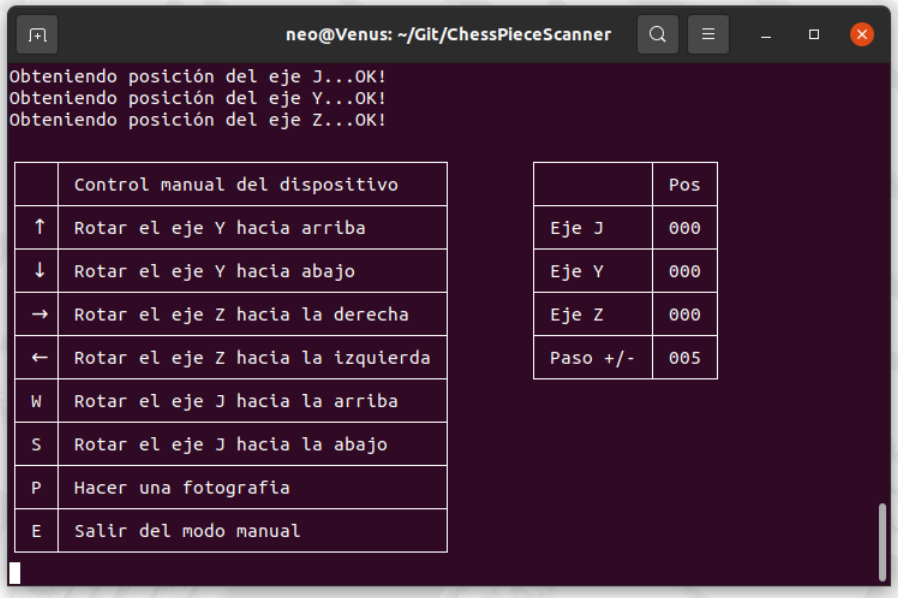


Figura 36: Menú de control manual de la aplicación chesspiecescanner

Al seleccionar la tercera opción el programa cargará una terminal que permite la comunicación por medio de comandos con el microcontrolador y mostrará el resultado de la ejecución (Figura 37).

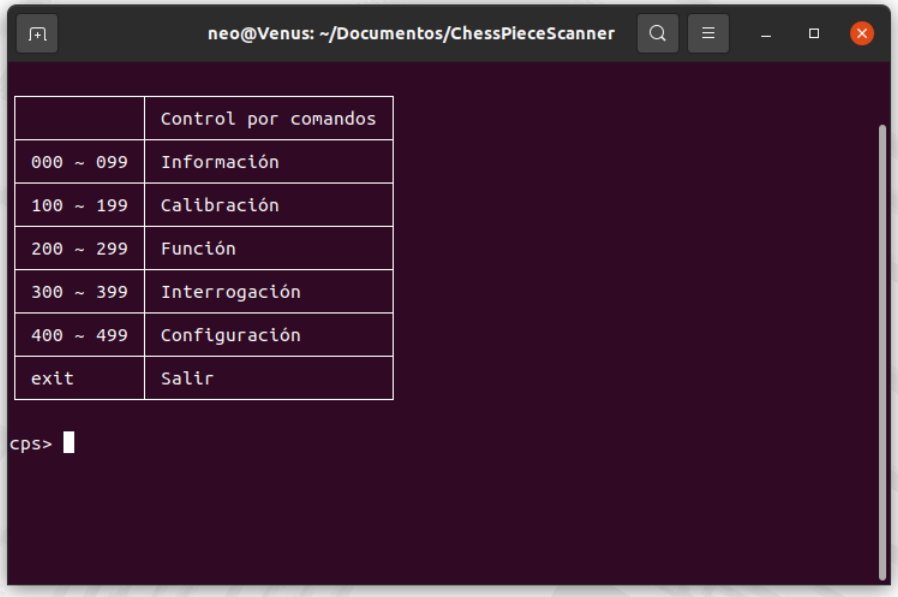


Figura 37: Terminal de la aplicación chesspiecescanner

La última opción finaliza la ejecución del programa desconectando el dispositivo y cerrando los puertos del microcontrolador y la webcam (Figura 38).

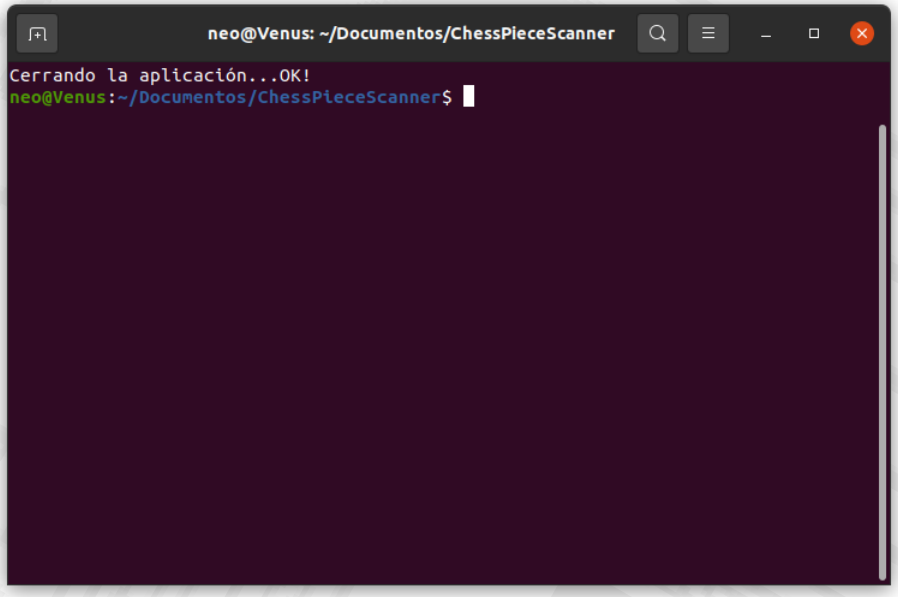


Figura 38: Cierre de la aplicación chesspiecescanner

2.7. Configuración

Los parámetros utilizados para la toma de fotografías y posterior recorte son configurables por medio del archivo 'conf/images.conf' donde se encuentran todos los ángulos de los ejes J e Y desde los que el dispositivo debe realizar fotografías y el rectángulo de recorte para cada una. También podemos configurar si se deben recortar las imágenes cambiando el valor de 'crop_enabled' entre 0,1.

Asimismo podemos configurar la webcam por medio del archivo 'conf/webcam.conf' donde podemos cambiar el brillo, el contraste, la saturación, la exposición y demás parámetros con los que se tomarán las fotografías.

La configuración de la webcam se aplica al iniciar la aplicación (Figura 33) por lo que si modificamos los valores deberemos reiniciar la aplicación para que estos surjan efecto. Por otro lado, la configuración de la toma de fotografías se lee al iniciar el asistente de escaneo (Figura 34) por lo que si modificamos algún valor en la configuración se aplicará cuando entremos a este menú.

2.8. Resultados

Tras realizar las fotografías de todas las piezas con fondo blanco y negro se obtuvo un dataset con un total de 13.824 imágenes de distintos ángulos que permiten inferir los detalles diferenciadores de cada una de las piezas. Véase la figura 39 dónde se muestran las 4 configuraciones utilizadas para realizar las fotografías.

Actualmente el Dataset esta elaborado únicamente con las piezas que nos entregaron al principio de curso, por lo que para mejorar su diversidad y hacerlo capaz de detectar con eficacia otros tipos de piezas sería necesario aumentar la muestra.

Figura 39: Imágenes tomadas por ChessPieceScanner

3. Reconocimiento del tablero

La detección del tablero en el que se disponen las piezas de ajedrez es la primera tarea que debe realizar la aplicación. En nuestro proyecto hemos adaptado los algoritmos de detección que se utilizaron en el proyecto LiveChess2FEN [Mal], que estaban inspirados en los trabajos de Maciej A. Czyzewski et al. [al]. Resumiremos aquí las partes esenciales de este algoritmo y nos centraremos en los cambios y adaptaciones que le hemos realizado para poder utilizarla en nuestra aplicación.

3.1. Algoritmo básico

Para reconocer la posición de la piezas en el tablero primero necesitaremos reconocer cada usa de las casillas de este, además de sus esquinas. El método utilizado realiza varias iteraciones en las cuales con la imagen inicial esta se va adaptando en cada ciclo a una imagen con únicamente el tablero. A su vez, cada iteración contiene 4 fases diferenciadas por la que detecta los vértices finales con los cuales es capaz de reconstruir la imagen de forma que solo obtenga una vista cenital del tablero y sus 64 casillas diferenciadas.

- SLID (Straight line detector): Detección de líneas rectas que puedan ser asociadas con un tablero de ajedrez
- LAPS (Lattice points search): Búsqueda de los puntos que puedan formar la cuadricula que engloba al tablero
- CPS (Chessboard position search): Estimar cual de todos los puntos es aquel que tiene mayor posibilidad de ser los que forman las esquinas de nuestro tablero.
- Identificación y recorte de la imagen con la posición del tablero.

3.2. Fase SLID

La fase SLID es la encargada de la detección de las líneas rectas del tablero e ignorar aquellas que no pertenezcan a él, también tiene el objetivo de unir líneas que han quedado separadas por situarse atrás de una de las piezas del tablero.

Algorithm 1 Pseudocódigo de una iteración:

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

```
1: procedure (imagen)
2:     lineas ← SLID(imagen)                    ▷ Detección rectas
3:     puntos ← LAPS(imagen, lineas)            ▷ Mapa de puntos
4:     vertices ← CPS(lineas, puntos)           ▷ esquinas tablero
5:     return(vertices, puntos)
6: end procedure
```

Algorithm 2 Algoritmo Fase SLID:

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

```
1: procedure (imagen)
2:     imagen ← simplificar(imagen)
3:     bordes ← canny(imagen)                   ▷ detecta los bordes
4:     segmentos ← hough(bordes)                ▷ busca líneas
5:     grupos ← agruparcolineales(segmentos)
6:     lineas ← fusionar(grupos)
7:     return(lineas)
8: end procedure
```

Esta fase usa el algoritmo de Canny [Can] para la detección de bordes del tablero y el algoritmo de Hough [J M00] para encontrar todas las posibles líneas de la imagen que pueden pertenecer al tablero.

Una vez obtenidas los segmentos se deben unir aquellos que pertenecen a la misma línea, la función agruparcolineales decide si dos segmentos son partes de una misma linea y se podrían fusionar, para decidir si los segmentos pertenecen o no se tiene en cuenta la longitud del segmento y el ángulo de inclinación con respecto a la imagen, tal y como es mencionado en [al]. Una vez se ha decidido que dos o más segmentos forman una recta se unen. Se crea así una nueva recta la cual se ajusta lo máximo posible a como sería la unión de ambos segmentos.

3.3. Fase LAPS

La fase LAPS generara los puntos del tablero que se forman con las intersecciones entre dos líneas generadas en la fase anterior. Una vez encontrados todos los puntos posibles, se seleccionan aquellos más prometedores de ser parte del tablero.

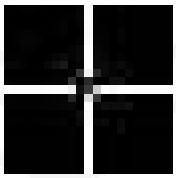
Algorithm 3 Identificación de la posición del tablero:

Entrada: Imagen que contiene un tablero.
Líneas detectadas por SLID.
Salida: Matriz de puntos que forman la cuadricula del tablero.

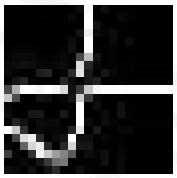
```
1:  procedure (imagen)
2:      puntos_cuadricula ← matriz[]
3:      for punto ∈ intersecciones(lineas) do
4:          matriz ← preprocesar(vecindad(imagen, punto)
5:          es_punto_cuadricula ← detector_geométrico(matriz)
6:          if es_punto_cuadricula then                ▷ modo detector geométrico
7:              puntos_cuadricula.insertar(punto)
8:              mejormarco ← marco
9:          else                                       ▷ modo red neuronal
10:             es_punto_cuadricula ← red_neuronal(matriz)
11:             if es_punto_cuadricula then
12:                 puntos_cuadricula.insertar(punto)
13:             end if
14:         end if
15:     end for
16:     return intersecciones(mejor_marco)
17: end procedure
```

El algoritmo utiliza en primer lugar un detector geométrico simple el cual comprueba el contorno alrededor del cruce de las dos rectas, si estas forman una cruz significa que es probablemente un punto perteneciente al tablero.

Si no se detecta una cruz con este detector geométrico, se realiza una nueva comprobación con una red neuronal que comprobará que no se trate de un punto del tablero que este oculto por una pieza que tenga delante. Esta red neuronal proporciona un porcentaje de probabilidad de que el punto sea parte del tablero, y admitiremos como validos aquellos casos en los que el porcentaje sea muy alto.



(a) Intersección que solo necesita del detector geométrico



(b) Intersección que requiere la red neuronal

Figura 40: Ejemplo de intersecciones

3.4. Fase CPS

La fase CPS tiene como objetivo encontrar los vértices del tablero, decidiendo del total de rectas encontradas cuales pertenecen al cuadrilátero de los límites del tablero.

Es importante tener en cuenta de que en el apartado anterior solo se detectaron los puntos que formaban intersecciones en la cuadrícula, lo cual no incluye a las líneas que marcan los límites del tablero, es por ello por lo que será necesario crear un margen de mínimo una casilla de grosor para no perder el tablero al ser recortado, este proceso se realiza en la última etapa de la iteración

Algorithm 4 Algoritmo Fase CPS:

Entrada: Líneas detectadas por SLID y puntos de la cuadrícula devueltos por LAPS.
Salida: Coordenadas en la imagen de las cuatro esquinas de la cuadrícula.

```
1: procedure (lineas, puntos_cuadricula)
2:     clusters ← DBSCAN(puntos_cuadricula)
3:     cluster_tablero ← max(clusters)
4:     lineas_candidatas ← cerca_borde(lineas,cluster tablero)
5:     lineas_verticales ← es_vertical(lineas_candidatas)
6:     lineas_horizontales ← es_horizontal(lineas_candidatas)
7:     max_valor ← −∞
8:     for v1,v2 ∈ lineas_verticales ⋀ h1, h2∈ lineas_horizontales do
9:         valor ← polyscore(v1, v2, h1, h2)
10:        if valor >max_valor then
11:            max_valor ← valor
12:            mejor_marco ← [v1, v2, h1, h2]
13:        end if
14:    end for
15:    return intersecciones(mejor_marco)
16: end procedure
```

La función polyscore es la encargada de tomar la decisión de que cuatro líneas forman los bordes internos del tablero, esta devuelve un coste el cual tiene su máximo cuando las cuatro escogidas dan el marco del tablero de ajedrez. Las cuatro rectas obtenidas determinarán el espacio central del tablero y, a partir de esto y de la distancia entre los puntos, se podrá deducir donde están los vértices del tablero en la última fase del proceso.

3.5. Fase final y recorte en vista cenital

Una vez hemos obtenido el marco del tablero y tenemos la matriz de puntos se procede a la transformación en perspectiva de la imagen, para esta perspectiva lo que hacemos es añadir los márgenes al marco central obtenido en la fase anterior para obtener lo que serían las esquinas de nuestro tablero, una vez con los puntos solo queda recortar convirtiendo las esquinas del tablero en las esquinas de la nueva imagen.

Figura 41: Detección de líneas, intersecciones y algoritmo DBSCAN aplicado

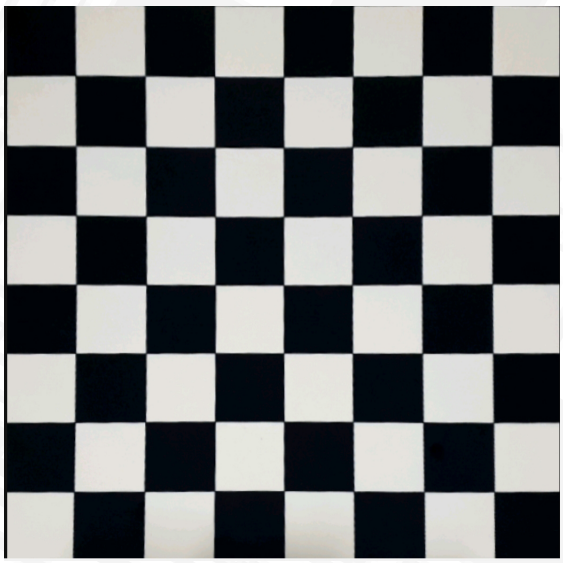


Figura 42: Imagen del tablero tras la tercera iteración

3.6. Rendimiento

Utilizando la aplicación en modo “profiling” para determinar el rendimiento y eficacia obtenemos que la aplicación es casi siempre capaz de encontrar el tablero o al menos los puntos de intersección en la aplicación, siendo la eficacia de detección de estos cercana a un 100 %. Sin embargo, en tableros con bordes adornados si presenta más dificultades a la hora de identificar todos los puntos de referencia del tablero. A pesar de eso, esto no tiene que presentar un problema grave debido a que con detectar un punto de la fila o columna donde no sea capaz de encontrar algún punto, disponemos de métodos para, suponer su posición ficticia del punto.

Para valorar su eficacia a la hora de identificación de tablero se ha creado una pequeña base de datos con todas las fotografías que hemos ido realizando con la aplicación a lo largo de la realización del proyecto luego hemos utilizado la misma aplicación para el recuento de puntos sobre el total que debería mostrar un tablero totalmente identificado ‘(x - 1) * (x - 1)’ . Siendo x el número de filas-columnas que forman el tablero partido del total de puntos).

Podemos por tanto afirmar que los resultados obtenidos fueron buenos, en especial con el tablero original que nos fue cedido, pero también para cualquier otro tablero de la misma estética que este.

Figura 43: Imágenes tomadas por la aplicación con diferentes dispositivos móviles

No obstante hemos podido apreciar situaciones, que describiremos a continuación, en las que la detección del tablero es más costosa o más imprecisa:

- En condiciones de poca luminosidad la detección del tablero se vuelve más imprecisa ya que debe haber un buen contraste entre los cuadrados blancos y negros para una óptima detección de las esquinas de estos. Además puede generar un mayor número de intersecciones que no pueden analizarse directamente con el detector geométrico volviéndose la operación de detección más costosa.
- En imágenes tomadas desde un ángulo superior a 30 grados con respecto del tablero la detección de las esquinas de las casillas es más complicada de resolver para el detector geométrico y aumenta el uso de la red neuronal junto con el tiempo de detección.

Las pruebas que hemos realizado en la plataforma Android muestran una mejora en los tiempos de detección con respecto al proyecto original, detectando el tablero en un dispositivo de gama media-alta en tan sólo 950ms, frente a los 3,84s del proyecto anterior.

4. Reconocimiento de piezas

Una vez detectado el tablero a partir de la imagen obtenida por la cámara, el siguiente paso es el reconocimiento y clasificación de las piezas. Para ello es necesario en primer lugar dividir el tablero en casillas individuales. Una vez realizada esta división se pasa a comprobar de forma individual si una casilla esta o no vacía y en el caso de tener una pieza de ajedrez, diferenciar cual de todas es.

Un problema que se plantea es que, como se puede ver en las fotografías, las piezas al estar en vista perspectiva no se ven únicamente en el espacio de delimita su casilla, causando que las piezas más altas, como la reina se muestren también en la casilla superior, lo que produce que, al dividir la imagen en casillas, en algunas piezas solo se muestre la base de la mismo, y que también aparezca la parte superior de las casillas situadas más abajo, lo que haría que no fuera posible distinguir entre las distintas piezas.

Una forma de solucionar este problema es tener en cuenta que la altura de las piezas es superior a la longitud de las casillas, de tal forma que no se tenga en cuenta únicamente lo que hay en el recuadro que delimita la casilla que queremos examinar. Para esto es necesario revertir el proceso de transformación en perspectiva de las iteraciones de tal manera que volvamos a tener una imagen de las figuras en perfil y altura. Esta idea está inspirada en el proyecto ChessVision [Din].

4.1. Reducción del tablero en casillas

Para la reducción por casillas hemos utilizado el mismo procedimiento que se sigue en LiveChess2FEN. Se guardan los cuatro puntos de las esquinas obtenidos al finalizar cada iteración en la fase LAPS (véase sección: 3.3) con el fin de realizar la inversión de la matriz de transformación y dividir el resultado de la última iteración en una cuadrícula 8×8 del tablero, de la cual obtendremos las esquinas de cada una de las casillas.

Una vez conocidas las esquinas de cada una de las casillas debemos recortar la imagen teniendo en cuenta la altura. Es importante recalcar que no todas las piezas tienen la misma altura por lo que es necesario ajustarla a la que mejor precisión obtenga, en nuestro caso utilizaremos un factor de 1,75 a la distancia entre los vértices superiores e inferiores de las casillas, haciendo que el recorte permita ver también la parte superior de la figura, la cual es lo diferencia una pieza de otra y permitiendo a

la red neuronal creada anteriormente clasificarlas correctamente.

Una vez se han identificado cada una de las 64 casillas, se procederá con su clasificación, siempre teniendo en cuenta que una casilla puede estar vacía. El resultado de la clasificación se devuelve utilizando la notación Forsyth-Edwards o "FEN" [For], ampliamente utilizada por los jugadores de ajedrez. No obstante, al estar la aplicación a desarrollar limitada a recibir toda la información mediante capturas no se puede tener toda la información de una partida como es el turno de los jugadores o si se pueden hacer ciertos movimientos como el enroque.

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

Figura 44: Ejemplo notación FEN (Fischer-Reshevsky)

4.2. Clasificación con redes neuronales

Como en LiveChess2FEN, la clasificación de las piezas se realiza utilizando una red neuronal convolucional (CNN). Se han explorado las mismas redes que se analizaron en el proyecto LiveChess2FEN. No obstante, estas redes han vuelto a ser entrenadas con el nuevo dataset que hemos elaborado. Para llevar a cabo dicho entrenamiento se ha utilizado el entorno de Google Colaboratory [Teaa] y los Jupyter Notebooks [Pér] de Anaconda3. Las redes se han definido y entrenado con Keras [Cho] y las librerías de Tensorflow [Teab]. Como se indicó en la introducción, se ha usado aprendizaje por transferencia para reducir los tiempos de entrenamiento.

El 80 % de las imágenes del dataset utilizado se destinaron al entrenamiento y

el 20 % restante se utilizó para datos de prueba y validación. Hay que tener cuidado con los porcentajes destinados al modelo y al test, ya que si utilizamos un porcentaje reducido la precisión será baja por lo que tendrá muchos fallos a la hora de identificar las piezas, y sin embargo, si el porcentaje es muy alto puede darse el caso de que produzca un sobre aprendizaje y no sea capaz de identificar en un futuro imágenes nuevas que sean diferentes a las utilizadas anteriormente.

```
from tensorflow import keras
from tensorflow.keras.preprocessing import image

from sklearn.metrics import confusion_matrix
from IPython.display import Image
```

In [3]:
```
physical_devices = tf.config.experimental.list_physical_devices('GPU')
print("Num GPUs Available: ", len(physical_devices))
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

```
Num GPUs Available:  1
```

In [4]:
```
from keras.applications import Xception
```

```
Using TensorFlow backend.
```

In [5]:
```
import numpy as np
```

In [6]:
```
xception_model=applications.xception.Xception()
```

```
WARNING:tensorflow:From C:\Users\katye\Anaconda3\envs\androidMobilnet\lib\site-packages\tensorflow\python\ops\init_ops.py:1251:
calling VarianceScaling.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future
version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
```

In [ ]:
```
xception_model.summary()
```

In [7]:
```
for layer in xception_model.layers:
    layer.trainable=False
```

In [8]:
```
from tensorflow.python.keras.models import Sequential
```

In [9]:
```
new_model = Sequential()
new_model.add(xception_model)
```

In [ ]:
```
new_model.summary()
```

In [11]:
```
new_model.add(Dense(120, activation='softmax'))
```

In [12]:
```
new_model.add(Dropout(0.5))
```

In [13]:
```
new_model.add(Dense(13,activation='softmax'))
```

Figura 45: Jupyter Notebook utilizado para el entrenamiento de las redes

En la siguiente captura se recogen los principales resultados de dichos entrenamientos, donde observamos que la red neuronal con un mayor porcentaje de acierto es Xception, seguida de MobileNetV2 y además podemos comprobar que la red neuronal que más tiempo de entrenamiento necesita es la VGG16.

	Porcentaje de acierto	**Tiempo de entrenamiento**
Xception	96,75 %	5418,00s
DenseNet201	90,90 %	3050,16s
NASNetMobile	81,69 %	757,55s
MobileNetV2	95,04 %	1143,82s
VGG16	83,13 %	6840,06s
VGG19	78,62 %	1919,25s

Cuadro 3: Resultados de los entrenamientos de las redes neuronales

Para la inferencia se ha utilizado Tensorflow Lite que al ser una versión reducida de Tensorflow genera una pérdida de precisión contra los modelos originales (entre otros motivos por usar float32 contra el float64). Para ello hemos comparado la precisión de cada modelo con su convertido.

La pérdida de precisión esperada se estima que es muy baja, por lo que no consideramos que utilizar TensorFlow Lite sea un motivo de ineficiencia. A continuación mostramos un cuadro que compara la pérdida de precisión de cada modelo.

	Porcentaje de acierto del modelo original	**Porcentaje de acierto del modelo convertido**
Xception	96,75 %	95,31 %
DenseNet201	90,90 %	88,57 %
MobileNetV2	95,04 %	91,93 %
VGG16	83,13 %	83,02 %

Cuadro 4: Resultados de los entrenamientos de las redes neuronales

Como en LiveChess2FEN, para aumentar la precisión de la clasificación se han añadido ciertas premisas específicas relacionadas con las reglas del ajedrez que ayudan a la toma de decisiones. Una premisa sencilla es que no es posible que haya más de un rey del mismo color en una partida.

Algorithm 5 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: procedure (vectores_probs)
2:     tablero ← {} * 64                          ▷ Lista vacía de las 64 casillas
3:     rey_blanco ← max_prob(vectores_probs, 'K' )
   //Fijamos los reyes, de igual forma para el rey negro
4:     for cada casilla ∈  vectores_probs do
5:         puntos_cuadricula.insertar(punto)
6:         if max_prob(casilla) = '_' then                          ▷ casillas vacías
7:             tablero[casilla] ← '_'
8:             por_rellenar ← por_rellenar − 1
9:         end if
10:    end for
   //Terminamos de rellenar el tablero en el orden dado por las probabilidades de
   las piezas
11:    while por rellenar >0 do
12:        pieza ← max_prob(tops)
13:        if ¬alcanzado_max(pieza, piezas_usadas) then
14:            tablero[pieza] ← pieza
15:            por_rellenar ← por_rellenar − 1
16:            piezas_usadas[pieza] ← piezas_usadas[pieza] − 1
17:        end if
18:        tops(pieza) ← ∅
19:    end while
20:    return tablero
21: end procedure
```

Si se está infiriendo partidas completas también se puede utilizar información adicional para mejorar la eficacia de la clasificación. Así por ejemplo, en cada turno solo se puede mover una pieza (excepto para enrocar o cuando hay una captura de piezas). Se puede por lo tanto asumir que es más probable que una pieza sea la misma que la que estaba en la misma posición en el turno anterior.

Para utilizar esta información de forma practica se debe dar mayor porcentaje a las piezas que fueron anteriormente elegidas en la misma posición y solo cambiar esta cuando se detecte un cambio claro en una nueva imagen. Este método es muy

práctico ya que la red neuronal diferencia con facilidad cuando se ha producido un movimiento y cambiado un espacio en blanco por el de una pieza y cuando una pieza ha sido capturada por otra por ser del color contrario.

Esta información se puede añadir al algoritmo de clasificación usando como entrada adicional la notación FEN de la anterior jugada, el vector de probabilidad utilizado para distinguir entre piezas y un algoritmo que conozca los movimientos posibles de cada una de las piezas de ajedrez.

Algorithm 6 Inferencia del tipo de pieza a partir de su movimiento:

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

1: **procedure** (FEN_anterior, vectores_probs)
2: $casillas \leftarrow casillas\_cambiadas(FEN\_anterior, vectores\_probs)$
3: $(casilla\_ini, casilla\_fin, accion) \leftarrow movinferido(FEN\_anterior, vectores\_probs, casillas)$
4: $piezas\_posibles \leftarrow piezas\_compatibles(casilla\_ini, casilla\_fin, accion)$
5: **end procedure**
6: **return** (piezas_posibles)

5. Desarrollo de la aplicación android

En este capítulo desarrollaremos el proceso de conversión de los algoritmos y optimizaciones mencionadas en los capítulos anteriores 3 y 4 a la aplicación Android y comentaremos como reescribimos a Java el código de Python.

5.1. Creación de la aplicación

En un principio tratamos de ejecutar el código de Python directamente sobre Android sin éxito debido a la incompatibilidad de este sistema con el lenguaje. Además cualquier intérprete desarrollado para esta plataforma debería estar desarrollado en lenguaje Java, teniendo así que interpretar el lenguaje Python para generar Java bytecode que debería ser interpretado de nuevo, lo que resultaría en una gran pérdida de rendimiento.

Es por ello que decidimos crear nuestro programa desde cero basándonos en los algoritmos desarrolladas en LiveChess2FEN. El entorno de trabajo escogido para la creación de la aplicación fue Android Studio [coa] debido a su versatilidad, documentación y ser creada por los mismos desarrolladores que el sistema operativo de Android (Google).

5.2. Diseño de la aplicación

El objetivo del diseño fue crear una aplicación sencilla e intuitiva de utilizar sin invertir demasiado tiempo ya que el objetivo principal era la funcionalidad de la aplicación.

El diseño de la aplicación consta de una vista de la cámara principal del dispositivo en la parte izquierda de la pantalla con la cual se puede ver en tiempo real la imagen capturada mientras que en el lado derecho podemos observar un dibujo descriptivo con la posición de cada una de las piezas en el tablero.

Para no tener problemas con las licencias, los iconos del tablero y de las piezas han sido creadas para el proyecto por Ángel Molina con Inkscape [Bry], siendo estos todos vectoriales para poder adaptarse a todas las resoluciones de pantalla de los diferentes teléfonos y tablets sin perder calidad de imagen.

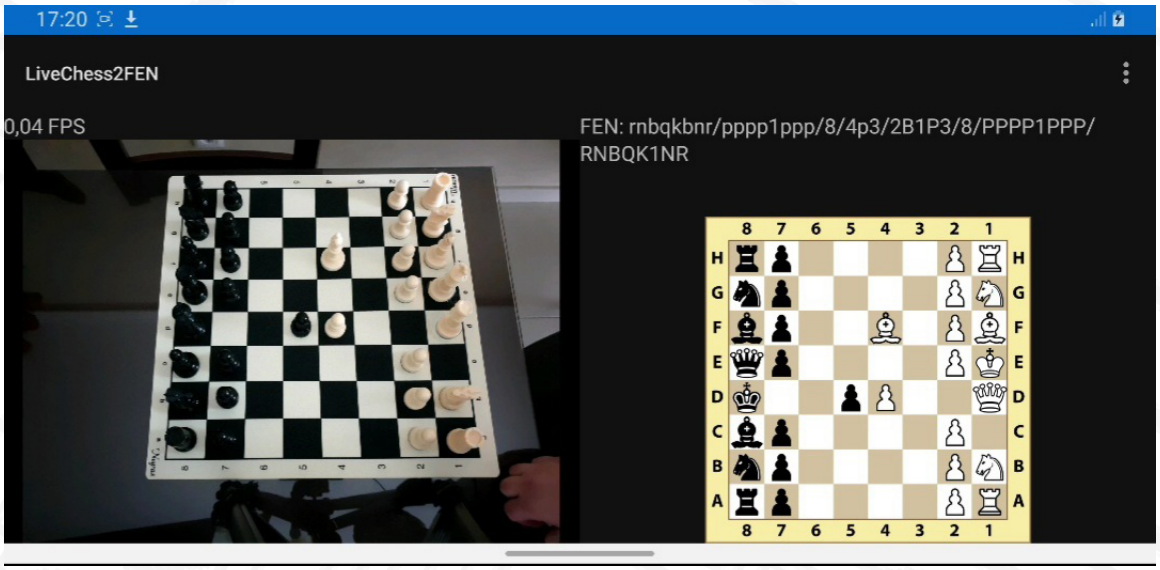


Figura 46: Ejemplo de vista de la aplicación

En la parte superior izquierda encontramos un botón de ajustes, a través del cual podemos acceder al menú de preferencias, desde dónde podemos seleccionar las redes neuronales que queramos utilizar para la detección del tablero y de las piezas y activar o desactivar el modo "profiling" que nos permite consultar el tiempo transcurrido en cada una de las etapas de la detección.

5.3. Desarrollo de la aplicación

Comenzamos el desarrollo de la aplicación con unos test unitarios de OpenCV y MLKit, desde los que desarrollaríamos más tarde la aplicación final.

En primer lugar nos centramos en el desarrollo del algoritmo de detección del tablero para el que hicimos uso de la librería OpenCV para Android programada en C++ e implementamos de manera minimalista los algoritmos de Bentley-Ottman [Val], DBSCAN [Fra] y PolyClipper [Mah] que eran necesarios para la parte de la detección del tablero. Asimismo implementamos los algoritmos de detección del tablero desde cero basándonos en los códigos desarrollados por Maciej A. Czyzewski [al].

Después nos centramos en el desarrollo de la red neuronal para la detección de esquinas del tablero, esta tarea resultó complicada ya que MLKit [cob] no soporta

redes neuronales con imágenes de tamaño menor a 128x128 píxeles y las imágenes de las esquinas eran de, tan solo, 21x21. Ampliar esas imágenes no era una opción ya que sólo aumentaría el tamaño del modelo y el coste de procesamiento por lo que tuvimos que descartarlo para esta red. En su lugar utilizamos la librería de TensorFlow para Android que sí nos permitía el uso de imágenes de este tamaño.

Una vez resuelta la detección del tablero, el siguiente paso fue realizar la detección de las piezas, de nuevo implementando desde cero los algoritmos ya desarrollados previamente y de nuevo añadiendo soporte multihilo. Para la detección de las piezas en un primer momento utilizamos MLKit que en este caso sí era compatible con nuestras necesidades, pero decidimos sustituirlo por la librería de TensorFlow para Android para así estandarizar las redes convolucionales y eliminar la librería MLKit (que es bastante pesada).

Finalmente implementamos una serie de algoritmos para la conversión de listas de piezas a notación FEN y viceversa, para la generación del tablero virtual con el fin de obtener un resultado más visual y comprensible para el usuario final.

5.4. Testing

Para poder analizar los resultados obtenidos por la aplicación desarrollamos un modo en el que se muestre una consola con los tiempos que invierte el procesador en las diferentes fases del programa, este modo es el llamado 'profiling' el cual se puede acceder desde el menú de opciones. Desde este modo podemos consultar:

- El nombre de la red neuronal que se esta utilizando para el reconocimiento de las piezas.
- El tiempo necesario para detectar el tablero.
- El tiempo que consume para dividir el tablero en casillas.
- El tiempo necesario para clasificar las piezas.

6. Resultados obtenidos por la aplicación

En este capítulo analizaremos los resultados obtenidos por la aplicación con las distintas redes neuronales. Se ha probado su eficacia con distintas fotografías tomadas desde distintas posiciones, con diferentes niveles de luz y dispositivos móviles.

6.1. Método de evaluación

Para evaluar la precisión de la aplicación se utilizó un trípode para teléfonos con el que mantenerlo fijo en una posición y así evitar afectar al rendimiento de la captura de imagen.Para las pruebas básicas, la lente de la cámara del dispositivo se colocó a 30º grados de inclinación respecto del tablero y la base del trípode (la cual esta a la misma altura que el tablero) se colocó a 60 centímetros de distancia (Figura 47). El objetivo de utilizar estos parámetros es simular la posición del teléfono con la aplicación a una posición similar a la que se suelen colocar el reloj en las partidas profesionales. Muchas de las retransmisiones en vivo también usan una cámara en esta posición.

Figura 47: Trípode utilizado colocando según el método de evaluación descrito

Para la evaluación de los tiempos hemos utilizado el modo profiling de la app y el historial de versiones de nuestra aplicación accesible desde nuestro repositorio de github [Ánga] con el cual podemos comparar el tiempo de speedup entre versiones y evaluar como cada una de estas han mejorado el rendimiento general de la aplicación. De la misma manera que podemos ver como cada red neuronal detecta el tablero y en cuanto tiempo con las diferentes imágenes a usar.

Las imágenes de prueba utilizadas son un conjunto propio de 25 imágenes tomadas del mismo tablero en distintas posiciones y ángulos, también utilizaremos las fotografías que se utilizaron como referencia en el proyecto LiveChess2FEN, que esta disponible en el github de dicho proyecto [Qui]. De este modo podemos comparar tambien entre el rendimiento que se obtiene con la versión anterior del proyecto en la Nvidia jetson nano, frente a la nuestra con dispositivos móviles.

6.2. Detección del tablero

Los algoritmos utilizados para la detección del tablero son esencialmente los mismos que se utilizaron en LiveChess2FEN. Por lo tanto, como cabria esperar, los resultados de precisión obtenidos que hemos obtenido son idénticos: 99 % de precisión para la detección de casillas y un 95 % de tasa de acierto a la hora de detectar el tablero completo [Mal].

Analizando si afecta al rendimiento el ángulo de inclinación de la cámara vemos que en esta fase no afecta en absoluto hasta que empiezas a probar en ángulos cerca de la vista de perfil en los cuales la forma cuadrada de las casillas empieza a deformarse demasiado pasando a ser un rectángulo con una altura apenas detectable para la lente. El nivel de luminosidad tampoco afecta drásticamente debido a que la diferencia de refracción entre los colores blanco y negro es muy alta y aunque no haya demasiada luminosidad, las casillas blancas resaltan mucho más que las de color negro a no ser que se tomen las fotos desde la penumbra (aunque también se puede aprovechar el flash de los teléfonos en estos casos y sigue funcionando bien la detección del tablero).

No obstante si hemos detectado que en los casos en los que tomas las fotos con el tablero girado no es capaz de detectar el tablero ya que no es capaz de detectar de forma correcta las intersecciones de las casillas cuando estas tienen forma de romboides para la cámara.

6.3. Clasificación de las piezas

Como vimos en la tabla del capítulo 4 (véase Tabla 3) algunas de las redes neuronales ofrecen mejores resultados a la hora de detectar las piezas, por lo que solo utilizaremos para la comparativa las redes Xception [FCh17], Densenet [G H17], MobileNet [MSa18] y VGG16 [K S], mientras que NasnetMobile [Zop] y VGG19 [tea] fueron descartadas porque antes de comenzar la comparativa ya son poco prometedoras teniendo menos de un 80 % de aciertos.

	Xception	**Densenet**	**MobileNetV2**	**VGG16**
Muestra LiveChess2FEN	94 %	90 %	93 %	86 %
Muestra propia	95 %	88 %	92 %	83 %
Tiempo detección	5,2s	5,1s	1,2s	0,76s

Cuadro 5: Precisión y velocidad con las diferentes muestras de imágenes para cada uno de los modelos

La tabla obtenida indica como las muestras utilizadas por David son las que mejores resultados presentan, esto se debía a que el recopilatorio de fotos que utilizó (además del a dataset) utilizaban un tablero cuya cuadrícula era de distinto color a las figuras, y las imágenes utilizadas no eran con un tablero con todas las piezas, lo cual aumenta el tiempo de detección al no haber tantas piezas que detectar y la precisión de detectarlas ya que hay más casillas vacías las cuales se detectan siempre correctamente.

Las fotografías en cenital obtienen una mejor tasa de acierto debido a que la imágenes al ser tomadas desde arriba permiten que cada pieza este en la misma casilla lo que reduce los fallos de la aplicación producidos a que una pieza obstruya la vista de otra o que confunda la posición de una figura con la de atrás suya.

Por último la fotografías de la muestra propia las cuales son fotografías tomadas con el método explicado anteriormente serían los más aproximados a los que se obtendrían en una partida real y es el que se utilizó también para evaluar el tiempo de detección. Respecto al tiempo de detección cabe destacar que es la media que necesitó en detectar las 25 fotografías que se utilizaron para nuestra propia muestra y cada tiempo utilizado para esa media fue el máximo que se mostró en la aplicación (el cual solía ser el primero ya que luego aprovecha la inferencia obtenida y mejora el rendimiento futuro)

7. Conclusiones y Trabajo Futuro

La visión artificial es un campo que cada vez toma mayor relevancia en nuestras vidas. El aumento de dispositivos con la capacidad de interactuar con el mundo exterior esta permitiendo una automatización masiva de actividades que anteriormente requerían de la interacción humana para su funcionamiento. El ajedrez fue uno de los primeros juegos en los que fue implementada la inteligencia artificial por lo que creímos interesante que también fuera uno de los primeros juegos en los que se consiguiese implantar modelos con visión artificial, más allá de la existente demanda de aplicaciones que sean capaces de sustituir los tableros actuales de gran coste sin perder la calidad y rendimiento que ofrecen.

La aplicación desarrollada es capaz de implementar una aplicación capaz de detectar un tablero y sus piezas con una tasa de refresco suficiente como para identificar movimientos en las partidas de ajedrez estándar siempre no se realicen movimientos en el tablero ni en el dispositivo de grabación, los cuales deben de permanecer estáticos.

El rendimiento obtenido es similar al que se obtuvo en el proyecto LiveChess2FEN. El nuevo Dataset ha mejorado el entrenamiento de las redes neuronales, pero tenemos que tener en cuenta las perdidas que se producen al pasar a Tensor Flow Lite, que no necesarias para poder ejecutar la aplicación en los dispositivos móviles utilizados. En el proyecto original LiveChess2FEN no se tenian estas restricciones, por lo que consideramos que nuestras aportaciones al proyecto han sido positivas.

Para la continuación con el desarrollo del proyecto seria conveniente ampliar el dataset con distintos tipos de piezas, ya que en este TFG solo hemos entrenado y validado los resultados con los mismos tipos de piezas.

La eficiencia de la aplicación también podría ser mejorada para permitir así su utilización en partidas tipo blitz o incluso bullet [Kid]). Pero para ello es necesario mejorar el rendimiento usando técnicas de aceleración hardware, que incluso podrían permitir el uso de dispositivos de gama baja.

La aplicación también podría haberse ampliado permitiendo la transmisión en vivo de la aplicación en servidores web con los cuales realizar un visualización en streaming de la misma forma que se realiza con las partidas profesionales en los campeonatos internacionales de ajedrez. Otras características que nos hubiera gustado implementar es la capacidad para detectar movimientos imposibles y advertirlos, utilizando por ejemplo la información obtenida por la inferencia de capturas consecutivas.

8. Introduction and motivation of this project

The image digitization of chess games is a problem still not solved in artificial intelligence. This technique is essential for the online broadcasting of chess games, at a professional level where you can broadcast live games digitally, or even compete against chess engines using a physical board.

Currently, one of the alternatives made are electronic boards such as those made by DGT which function as input devices capable of detecting the position of their pieces on the board. One of the drawbacks of these boards is their expensve price, since these can cost up to 500€ without any accessory, so they are not an option for many people. In addition, these boards only work with their own pieces and software.

Another interesting option are the new automatic robots which are able to play with a physical board. Many of them have been made with free software [Mey] so that they are able to work with a computer such as the well-known Raspberry pi at economic prices, unfortunately it also has a series of drawbacks such as it is not able to start a game from an intermediate game, or the complexity of using or installation of the robot which requires to have an advanced level of programming and robotics knowledge, that very few people have, so it is not accessible to most people.

Artificial vision offers an increasingly attractive alternative for this type of problem due to its low cost and to the fact that it can be implemented in a large number of devices. To be specific, through the development of artificial intelligence applications, any computer, from desktop computers using a webcam, to mobile phones for home use, could be able to perform these tasks, currently only possible through expensive boards, which affects notably to the realization of tournaments and championships of this practice that is chess.

Futhermore, the use of this technology would allow us to remember games played physically and even to be able to compete against an artificial intelligence without the need for a monitor showing the board, giving the player the option of playing with a convencional chessboard.

Most projects that use this technology are specialized to work only under previous parameters such as the exact position of the camera or with the specific board with which it was trained to provide artificial vision. However, currently more versatile versions have been created that are able to deal with this problem and obtain accurate results, although none has yet been able to work in Android environments

(because there are no conversions of all the Libraries or frameworks of TensorFlow of Python [**tensoflow**] essential for the realization and conversion of 2D images of boards to reading format for computers, also known as FEN terminology [**FEN**]).

That is the reason for we decide to develop this project in order to be useful to future developers and people around the world when it comes to finding alternatives to current options.

The antecedent of our work is LiveChess2FEN, a framework made by David Mallasén Quintana as part of his Final Degree Project in the 2019-20 academic year [**quintana2020livechess2fen**; **memoria˙david**] which includes numerous investigations in projects related to the detection of chess boards and pieces.

A practical limitation of LiveChess2FEN is that it is implemented entirely in Python and uses an Nvidia Jetson Nano as a platform [Nvi], so all its code, both the algorithms used by it and those self-generated by the frameworks used cannot be used to create mobile applications. In addition, the available code of LiveChess2FEN is also not structured to work directly through video in such a way that images can be captured in the same way that they are analyzed as a game passes. Finally, the training of the neural networks used in LiveChess2FEN was done with a small sample of images, since no database of well-labeled chess pieces is publicly available. Improving training with a larger database is therefore another interesting aspect to explore.

8.1. Objectives

Neural network training needs to be effective training with large amounts of preprocessed data that allow parameter inference. This training is very expensive, but fortunately, there are a lot of pre-trained networks available that we can use. However, these networks do not offer a satisfactory solution to the specific problem we pose, in our case the inference of chess pieces in an image.

In these cases, transfer learning is usually used, a common methodology in Deep Learning in which instead of training networks from scratch, a network previously trained for a task is used as a starting point, in our case recognition of objects. It is in this *second* learning that a database of labeled chess pieces is needed. However, the samples of chess pieces in public databases are very limited. This is why our first objective is precisely the creation of our own database which takes images from various angles of each of the pieces that make up the board. To simplify the creation of a large database, we considered the design and implementation of a system capable of quickly taking these photographs automatically.

The next objective would be to continue with the work of the previous student David Mallasén and adapt his algorithm and code to the continuation of our project in such a way that it can be used for our objectives.

The last objective will be from the union of both previous objectives to try to join the code with the database to develop an application and assess the results obtained, the latter is important because David carried out his project with a Jetson nano [Nvi], a device which is moderately expensive and requires a computer knowledge superior to that of other simpler computers. That is why we finally decided to opt for an Android application since it would solve the two previous problems, being usable by any mobile phone (and desktop or laptop computer with Windows [Cha]11) and not being necessary to adapt the user to any new device, since it is only needed to install the program to use it.

To facilitate the future extension of the project, all the source code developed is available in the references of this document [Ángb] [Ánga]

8.2. Work plan and document organization

This would be the work plan to follow that we would use to try to meet the objectives detailed above as successfully as possible.

- Develop a semi-automatic system for photographing chess pieces that is also based on free hardware and software using a webcam and an Arduino Uno [Mas] microcontroller. The device is connected via 2 USB ports to a computer with Linux [Tor] and, after placing a piece, start taking pictures from different angles to cover as many possibilities as possible.
- Use as far as possible, the advances of the previous student David Mallasén together with our new database to create the new algorithm for the recognition of the board and for the identification of figures.
- Once the code has been rebuilt and the algorithms are finished, we have to make an application capable of being easily used by anyone regardless of their programming knowledge will be developed.

The dissertation of the project is divided into 10 chapters the first is the introduction to the project and the last 3 are for the English translation and the contributions of the members of the group, the rest correspond to the entire process of elaboration of our artificial intelligence application, from the creation of the database with the ChessPieceScanner to the analysis of performance of the application. All the sections are ordered as we advanced with the TFG project so for chapter the progress of the previous one was needed.

9. Conclusions and future work

Artificial vision is a field that is becoming increasingly important in our lives during last years. The greater number of devices with the ability to interact with the outside world is increasingly allowing massive automation of activities that previously required human interaction for their operation. Chess was one of the first games in which artificial intelligence was implemented, so we thought it was interesting that it was also one of the first games in which models with artificial vision were implemented.Beyond the existing demand for applications that are able to replace the current high-cost boards without losing the quality and performance they offer.

The application developed is able to detect a board and its pieces with a refresh rate sufficient to identify movements in standard chess games as long as no movements are made on the board or on the recording device, which must remain static.

The performance obtained is similar to that obtained in the LiveChess2FEN project. The new Dataset has improved the training of neural networks, but we have to take into account the losses that occur when moving to Tensor Flow Lite, which are not necessary to be able to run the application on the mobile devices used. In the original LiveChess2FEN project there were no such restrictions, so we consider that our contributions to the project have been positive.

For the continuation with the development of this project, the dataset could be expanded or include new datasets with different boards to give the possibility of using more boards with different shapes of pieces because we have just trained our database and validate the results with the same chessboard.

The efficiency of the application could also be improved to allow faster games (such as blizt or bullet modes [Kid] mode) and the use of low-end devices.

The application can be extended allowing the live transmission of the application on web servers with which to perform a streaming visualization in the same way as it is done with professional games in international chess championships. Other features that we would have liked to implement is the ability to detect impossible movements and warn them, using for example the information obtained by the inference of consecutive captures.

10. Contribuciones

En este último apartado haremos un breve resumen de las contribuciones personales de cada uno de los integrantes del TFG.

10.1. Ángel Molina Núñez

Mi participación en el proyecto fue principalmente destinada a la implementación y traducción de código, tanto propio como el originalmente utilizado por David Mallasén Quintana en el proyecto original LiveChess2FEN.

En un primer momento fui el encargado de probar el entorno de David sobre una Nvidia Jetson Nano para comprobar cómo funcionaba la aplicación de Python, ya que esta sería nuestra base de trabajo.

Desarrollé el dispositivo ChessPieceScanner construido sobre Arduino mencionado en el capítulo 2, la redacción del mismo y la aportación de las imágenes utilizadas en la memoria sobre su instalación y montaje, una vez construido me encargué de diseñar el software y de la creación semiautomática de la base de datos de las piezas de ajedrez que utilizaríamos posteriormente para el entrenamiento de las redes neuronales para la detección de las piezas y el tablero.

Mi cometido principal fue el desarrollo de la aplicación en lenguaje Java desde cero, utilizando la librería de C++ OpenCV [Com] para procesamiento de imágenes y con conectores para su uso en Java. Comencé desarrollando el algoritmo del tablero e incorporando las librerías necesarias para su funcionamiento incluyendo únicamente las partes fundamentales de éstas. Posteriormente me centré en la adición de la librería de TensorFlow para Android y el desarrollo de la detección de piezas sobre el tablero. Finalmente, añadí una serie de reglas básicas para mejorar el resultado de la detección de las piezas y consiguiendo aumentar la precisión de la aplicación. Además se incluyó en los algoritmos soporte multihilo -ya que los dispositivos Android suelen tener procesadores de entre 4 y 8 núcleos en oposición a la Jetson Nano- con el objetivo de acelerar los tiempos de detección y conseguir una aplicación capaz de funcionar a tiempo real. Todo el código fuente se encuentra disponible en mi repositorio de GitHub [Ánga].

Asimismo creé el icono de la aplicación, de cada una de las piezas y la imagen del tablero virtual.

Generé un entorno de entrenamiento basado en cuadernos Jupyter para el entrenamiento de modelos TensorFlow y su posterior conversión a modelos TensorFlow Lite de modo que fueran válidos para la aplicación Android desde los que se entrenaron posteriormente todos los modelos utilizados y que yo mismo utilicé para el entrenamiento de una red neuronal secuencial para la detección de las esquinas del tablero con las mismas imágenes que utilizó David del repositorio de Artur Laskowski en Github [al]. También utilicé este mismo entorno para el entrenamiento de la primera red neuronal basada en MobileNetV2 para la detección de piezas del tablero.

Grabé los videos de demostración del funcionamiento del dispositivo ChessPieceScanner y de la aplicación LiveChess2Fen.

Redacté algunos fragmentos de la memoria además de los ya mencionados anteriormente y revisión general de la misma.

10.2. Carlos Plaza García-Abadillo

Mi participación principal esta relacionada con el desarrollo de la memoria que hemos realizado utilizado Latex [Lam80] mediante el uso de la plataforma online de Overleaf [Lim11b], la cual incluye para ayuda a los usuarios principiantes (como yo cuando comencé esta memoria) documentación detallada de las diferentes librerías y comandos utilizados para realizarla (enlace de la guía básica que utilicé para las consultas más básicas [Lim11a]).

También realicé la traducción de los primeros apartados al inglés y realice las adaptaciones oportunas para cumplir con los criterios estimados por la normativa vigente de presentación de memorias de la universidad, añadiendo los documentos bibliográficos utilizados de los compañeros y propios en cada uno de los apartados.

Utilización de diversas librerías y frameworks empleados anteriormente por David en Python mediante Anaconda y la creación de un entorno de trabajo. Esto fue necesario para comprobar si era posible utilizar el trabajo de David sin la necesidad de la Jetson Nano [Nvi] y si sería posible reutilizar las diferentes partes de código que había elaborado para su uso posterior, de la misma forma que encontrar alternativas para otros lenguaje de programación que se iban a utilizar y la posibilidad de utilizar herramientas de traducción de código que finalmente se descartaron debido a ser inviables por la gran cantidad de trabajo que supondría y las posibles pérdidas de rendimiento.

Realicé las pruebas de rendimiento, utilizando distintas fotografías tomadas por mí o recogidas de los proyectos consultados por David entre distintos dispositivos y tomé las conclusiones obtenidas de los resultados obtenidos tanto en la comparación de dispositivos utilizados como en las versiones de aplicaciones utilizadas, dejando los resultados en las distintas tablas y gráficas del apartado 6, así como sacar las conclusiones sobre el trabajo futuro que se puede realizar para optimizar los resultados obtenidos.

10.3. Katya Alejandra Rengel Lazcano

Cuando comenzamos con el proyecto, tuvimos reuniones para ver opciones de cómo mejorar el proyecto que nos fue asignado. Por mi parte investigué acerca del formato PGN (Portable Game Notation) y la notación FEN (notation Forsyth-Edwards) [Cas] y la utilización de una base de datos OpenSource [MI], para poder utilizar en las diferentes partidas mediante las bases de datos y la búsqueda de información sobre Python para Android (Kivy, etc), para ver si podíamos implementar el proyecto en una aplicación móvil.

Una tarea que me fue asignada fue la de realizar los entrenamientos con los diferentes modelos que nos ofrece Tensorflow, para ello tuve que aprender el lenguaje Python, porque las aplicaciones que utilizamos para entrenar fueron Google Colaborate y Jupyter de Anaconda3.

Como fue la primera vez que utilice estas aplicaciones, la configuración y la codificación me fue difícil y tuve que ver tutoriales y realizar un curso "Machine Learning con Android utilizando Tensorflow" de Udemy [Iba] , así logré tener una base más fuerte para realizar el entrenamiento de los modelos.

Al principio utilice Google Colaborate para el entrenamiento del reconocimiento de las piezas de ajedrez, pero tuve el problema de que la memoria no era suficiente para la base de datos tan grande que teníamos, así que busque otra aplicación y después de probar Spyder [Adc] y Jupyter de Anaconda [WO], decidí que los realizaría con los NoteBooks de Jupyter.

En Anaconda3 tenemos un entorno básico pero nosotros necesitábamos uno específico porque teníamos que utilizar diferentes librerías para los entrenamientos, así que otra dificultad que tuve fue la de configurar y probar entornos con diferentes versiones de librerías, una vez que di con la configuración correcta pude comenzar a

programar.

El primer modelo que implementé fue Xception, ya que era el que mejor resultado le había dado a David Mallasén en su proyecto. Este es un modelo pre-entrenado, integrado en Keras, que puede reconocer 1000 categorías de objetos con una precisión muy alta. Ya con estos conocimientos, me fue más fácil implementar los otros modelos (VGG16, VGG19, NasNet y DenseNet).

Para la implementación tuve que investigar las características de cada modelo, ya que cada modelo tiene características específicas como el tamaño, etc. Al principio los entrenamientos tardaban mucho y al utilizar la GPU los tiempos de entrenamiento se redujeron bastante.

Finalmente aporte parte de la descripción del entrenamiento de las redes neuronales utilizas para la clasificación de piezas.

Referencias

[Lam80] Leslie Lamport. *Latex - A document preparation system*. 1980. URL: https://www.latex-project.org/.

[J M00] J. Kittler J. Matas C. Galambos. *Robust detection of lines using the progressive probabilistic Hough transform*. 2000, págs. 119-137.

[Lim11a] WriteLatex Limited. *guide LaTeX editor*. 2011. URL: https://es.overleaf.com/learn.

[Lim11b] WriteLatex Limited. *The easy to use, online, collaborative LaTeX editor*. 2011. URL: https://es.overleaf.com/.

[FCh17] F.Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. URL: https://arxiv.org/abs/1610.02357.

[G H17] L. Van Der Maaten y K. Q. Weinberger G. Huang Z. Liu. *Densely connected convolutional networks*. 2017. URL: https://ieeexplore.ieee.org/document/8099726.

[MSa18] M.Zhu M.Sandler A.Howard. *Mobilenet Inverted residuals and linear bottlenecks*. 2018. URL: https://ieeexplore.ieee.org/document/8578572.

[Adc] Nick Adcock. *spyder, open source scientific environment*. URL: https://www.spyder-ide.org/.

[al] Maciej A. Czyzewski et al. *Chessboard and piece recognition with neuronal networks*. URL: https://github.com/maciejczyzewski/neural-chessboard.

[Ánga] Katya Rengel Ángel Molina Carlos Plaza. *Android LiveChess2FEN*. URL: https://github.com/angmolin/Android-LiveChess2FEN.

[Ángb] Katya Rengel Ángel Molina Carlos Plaza. *ChessPieceScanner*. URL: https://github.com/angmolin/ChessPieceScanner.

[Bry] Nathan Hurst Bryce Harrington Ted Gould. *Inkscape, editor de gráficos vectoriales libre*. URL: https://inkscape.org/es/.

[Can] J. Canny. *A computational approach to edge detection*. 6. IEEE, págs. 679-698.

[Cas] Pedro Castro. *FEN y PGN*. URL: https://sites.google.com/site/motoresdeajedrez/ajedrez-con-ordenador/fen-y-pgn.

[Cha] Windows Dev Channel. *Soporte APK Windows 11*. URL: https://blogs.windows.com/windows-insider/2021/10/20/introducing-android-apps-on-windows-11-to-windows-insiders/.

[Cho] François Chollet. *Keras*. URL: https://keras.io/.

[coa] Google co. *IDE android studio SDK Tools*. URL: https://developer.android.com/.

[cob] Google co. *Machine learning for mobile developers*. URL: https://developers.google.com/ml-kit.

[Com] Intel Company. *Librería OpenCV*. URL: https://opencv.org/.

[Dav] Alberto Antonio del Barrio García y Manuel Prieto Matías David Mallasén Quintana. *LiveChess2FEN: a Framework for Classifying Chess Pieces based on CNNs*. arXiv: 2012.06858 [cs.CV].

[DGT] DGT. *Tableros electrónicos*. URL: http://www.digitalgametechnology.com/index.php/products/electronic-boards.

[Din] J. Ding. *Chess Board and Piece Recognition*. URL: https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf.

[For] Forsyth-Edwards. *Sistema de notación Forsyth-Edwards*. URL: https://es.wikipedia.org/wiki/Notaci%C3%B3n_de_Forsyth-Edwards.

[Fra] Christopher Frantz. *Algoritmo de DBSCAN*. URL: https://github.com/chrfrantz/DBSCAN/.

[Iba] Freddy Daniel Alcarazo Ibañez. *Machine Learning con Android utilizando Tensorflow*. URL: https://www.udemy.com/course/machine-learning-con-android-utilizando-tensorflow-lite/.

[K S] A. Zisserman K. Simonyan. *VGG16 – Convolutional Network for Classification and Detection*. URL: https://neurohive.io/en/popular-networks/vgg16/.

[Kid] Harvey Kidder. *Ajedrez rápido*. URL: https://es.wikipedia.org/wiki/Ajedrez_r%C3%A1pido#Tipos_de_ajedrez_r%C3%A1pido.

[Mah] Tobias Mahlmann. *clipper*. URL: https://github.com/lightbringer/clipper-java.

[Mal] David Mallasén. *Técnicas de aceleración para el reconocimiento de piezas de ajedrez*. URL: https://eprints.ucm.es/id/eprint/61946/.

[Mas] David Cuartielles Massimo Banzi. *Arduino Uno*. URL: https://www.arduino.cc/en/Main/arduinoBoardUno&gt.

[MI] Slashdot Media y Dice Inc. *Study Chess Like You Mean It*. URL: https://sourceforge.net/.

[Mey] Joey Meyer. *Raspberry Turk chess robot*. URL: https://github.com/joeymeyer/raspberryturk.

[Nvi] Nvidia. *NVIDIA® Jetson Nano™*. URL: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-nano/.

[Pér] Fernando Pérez. *jupyter notebooks*. URL: https://jupyter.org/.

[Qui] David Mallasén Quintana. *LiveChess2FEN*. URL: https://github.com/davidmallasen/LiveChess2FEN.

[Teaa] Google Team. *Google Colaboratory*. URL: https://colab.research.google.com/.

[Teab] Google Brain Team. *TensorFlow*. URL: https://www.tensorflow.org/.

[tea] VGG team. *VGG19 – Convolutional Network for Classification and Detection*. URL: https://gist.github.com/baraldilorenzo/8d096f48a1be4a2d660d.

[Tor] Linus Torvalds. *Linux operating system*. URL: https://www.linux.org.

[Val] Petr Valenta. *Algoritmo de bentley-ottmann*. URL: https://github.com/valenpe7/bentley-ottmann.

[WO] Peter Wang y Travis Oliphant. *Anaconda, data science plataform*. URL: https://www.anaconda.com/.

[Zop] Barret Zoph. *Learning Transferable Architectures for Scalable Image Recognition*. URL: https://arxiv.org/abs/1707.07012.