Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica, mención en Instrumentación electrónica v sistemas de control

Sistema de gazetracking con YOLOv8

Autor: Francisco José Clavijo Chacón

Tutor: Ramón González Carvajal

Dpto. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Trabajo de Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Sistema de gazetracking con YOLOv8

Autor:

Francisco José Clavijo Chacón

Tutor:

Ramón González Carvajal

Profesor titular

Dpto. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Carrera: Sistema de gazetracking con YOLOv8

Autor: Francisco José Clavijo Chacón
Tutor: Ramón González Carvajal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia
A mis maestros

Agradecimientos

Para mí el grado ha sido una etapa que ha tenido sus altibajos pero que me ha enseñado multitud de lecciones muy valiosas no solo para mi carrera profesional, sino también para mi vida personal. Ahora y tras numerosos tropiezos, puedo decir que voy a concluir esta etapa con orgullo con este trabajo de fin de grado, aunque no sin antes agradecer a todas las personas que me han apoyado durante la misma.

En primer lugar me gustaría agradecer a mi profesor en último año y tutor en este TFG, Ramón. Muchas gracias no sólo por tutelarme, sino por preocuparte tanto de tus alumnos como lo haces y por demostrar tanta pasión como profesor.

Por otra parte y de manera especial, me gustaría agradecer a mi compañero y amigo Jerónimo, cuyos consejos y apoyo me han ayudado en momentos difíciles y sin el cual yo no podría haber finalizado este trabajo.

A mis padres, gracias por ser el hombro en el que apoyarme y por intentar darme la mejor guía que han podido en cada momento.

Y por último agradecer con todo mi corazón a Tamara, mi compañera de vida, sin la cual no habría llegado a dónde estoy. Gracias por tu apoyo tan incondicional como sincero.

¡Muchas gracias!

Francisco José Clavijo Chacón
Sevilla, 2023

Resumen

El seguimiento del movimiento de los ojos o *gazetracking* es algo que lleva estudiándose desde comienzos del siglo XIX y que a día de hoy sigue intrigándonos. Esto es debido a que la cantidad de campos en los que se podrían aplicar los datos obtenidos de dichos estudios es inmensa, desde la biología y la medicina hasta el marketing y los negocios.

En el mercado existen muchos dispositivos que son capaces de realizar gaze tracking usando algoritmos que pueden tener más o menos éxito a la hora de identificar y seguir el ojo en tiempo real. Esta tarea requiere de una gran capacidad de cómputo, lo que lleva a un gran consumo de recursos y hace que dichos dispositivos sean difícilmente portátiles.

El presente proyecto tiene como objetivo el estudio de las redes neuronales para abordar el problema de seguimiento ocular. Para ello, se estudiarán las soluciones más prometedoras del estado del arte: las redes neuronales convolucionales (CNN) y las redes neuronales recurrentes.

Dado que la detección de objetos es algo que se lleva estudiando bastante tiempo, hemos encontrado varias herramientas que podríamos usar para llevar a cabo nuestro objetivo, aunque nosotros hemos escogido usar la tecnología de Ultralytics, el modelo YOLOv8 con sus diferentes variantes.

Abstract

Eye movement tracking, or gazetracking, is something that has been studied since the early 19th century and continues to intrigue us today. This is because the number of fields in which data from such studies could be applied is immense, ranging from biology and medicine to marketing and business.

There are many devices on the market that are capable of doing this using algorithms that may be more or less successful in identifying and tracking the eye in real time. This requires high computational power, which leads to high resource consumption and the latter makes such devices hardly portable.

The present project aims to implement a software solution to the gaze tracking problem using a neural network. One of the most promising solutions are convolutional neural networks (CNN) and recurrent neural networks.

Since object detection is something that has been studied for quite some time, we have found several tools that we could use to achieve our goal, although we have chosen to use the Ultralytics technology, the YOLOv8 model with its different variants.

Índice

Agradecimientos **9**
Resumen **11**
Abstract **13**
Índice **15**
Índice de Figuras **17**
1 Introducción **1**
1.1 Motivación *2*
1.2 Objetivos a cumplir *3*
1.2.1 Requisitos hardware 4
1.2.2 Requisitos software 4
2 Estado del arte **7**
2.1 Antecedentes del gazetracking *7*
2.2 Cómo funciona el ojo humano *9*
2.2.1 Sistema para el movimiento ocular 11
2.2.2 Movimientos oculares 12
2.3 El uso de redes neuronales en el seguimiento de la mirada *13*
2.3.1 Red neuronal convolucional 13
2.3.2 Cómo se implementan las redes neuronales convolucionales 16
2.3.3 Modelos neuronales más populares en la actualidad 18
2.3.4 Aplicaciones en la actualidad 21
2.4 Conclusión *22*
***3* Desarrollo de la solución** **24**
3.1 Desarrollo del hardware *24*
3.1.1 Diseño e impresión 3D 24
3.1.2 Uso en ordenador y en LattePanda 26
3.2 Desarrollo del software *28*
3.2.1 Adquisición de datos 28
3.2.2 Desarrollo del modelo YOLOv8 30
3.2.3 Optimización de YOLOv8 con ONNX y TFLite 40
4 Resultados **43**
4.1 Resultados del modelo .pt *43*
4.2 Ejecución del modelo en diferentes entornos *46*
4.2.1 Ejecución en ordenador personal 47
4.2.2 Ejecución en LattePanda 48
4.3 Conclusión y mejoras a futuro *50*
5 Referencias **53**
6 Anexos **56**

ÍNDICE DE FIGURAS

Ilustración 1: Número de usuarios de *smartphones* a nivel mundial. Fuente: Statista. 2
Ilustración 2: Número de publicaciones por año en el que vemos el aumento de las relacionadas con las CNN. Imagen de: https://www.researchgate.net/publication/339578120_An_Introductory_Review_of_Deep_Learning_for_Prediction_Models_With_Big_Data 3
Ilustración 3: Número de publicaciones sobre *eye tracking* por año. Fuente: Scopus. 9
Ilustración 4: Anatomía del ojo humano. Imagen de Wikipedia: *https://commons.wikimedia.org/wiki/File:Eyesection-es.svg* 11
Ilustración 5: Músculos extraoculares. Imagen de Wikipedia: *https://commons.wikimedia.org/wiki/File:MUSCULOS_OCULARES.JPG* 12
Ilustración 6: Estructura de una red neuronal convolucional. Imagen de: *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53* 14
Ilustración 7: Esquema de red neuronal. Imagen de: *https://www.aprendemachinelearning.com/aprendizaje-profundo-una-guia-rapida/* 15
Ilustración 8: L a famosa imagen "Lena" y sus valores de píxeles en código RGB. Imagen de: *https://www.researchgate.net/figure/RGB-color-values-of-some-pixels-in-Lenas-image_fig1_317351196* 16
Ilustración 9: Lena con un preprocesado aplicado a escala de grises. Imagen de: *http://kurauchi.com.br/post/means-medians-and-images/* 17
Ilustración 10: Muestra gráfica de cómo opera el Kernel. Imagen de: *https://developer.apple.com/documentation/accelerate/blurring_an_image* 17
Ilustración 11: Arquitectura de un detector de tipo SSD. Imagen de: *https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab* 19
Ilustración 12: Arquitectura de una red de tipo Faster R-CNN. Imagen de: *https://www.researchgate.net/figure/The-architecture-of-Faster-R-CNN_fig2_324903264/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6Il9kaXJlY3QiLCJwYWdlIjoiX2RpcmVjdCJ9fQ* 20
Ilustración 13: Conjunto de piezas que forman el cerramiento en Cura. 25
Ilustración 14: Conjunto de piezas que forman el cerramiento en Cura. Imagen de: https://www.thingiverse.com/thing:3860552 26
Ilustración 15: Conjunto de piezas que forman el cerramiento en Cura. Imagen de: *https://www.lattepanda.com/lattepanda-alpha* 27
Ilustración 16: Imagen de ojo izquierdo sacada del banco de datos 29
Ilustración 17: Imagen de un ojo derecho con gafas sacada del banco de datos. 30
Ilustración 18: Etiquetado de imágenes con labelme. 31
Ilustración 19: Datos generados por labelme. 32
Ilustración 20: Comparativa en tamaño y rapidez de las diferentes versiones de YOLO. Imagen de: *https://github.com/ultralytics/ultralytics/tree/main* 33
Ilustración 21: Tabla con los atributos de cada tipo de modelo.Imagen de:

https://github.com/ultralytics/ultralytics/tree/main 34
Ilustración 22: Inicio del entrenamiento del modelo 38
Ilustración 23: Finalización del entrenamiento del modelo 39
Ilustración 24: Esquema de la optimización entrada-salida al modelo .onnx. Imagen de: *https://docs.ultralytics.com/yolov5/tutorials/model_export/?query=tflite#export-a-trained-yolov5-model* 40
Ilustración 25: Gráficas de resultados de nuestro modelo entrenado 44
Ilustración 26: Detección de la pupila en imágenes durante la validación 46
Ilustración 27: Detección de la pupila en pc del modelo .pt 47
Ilustración 28: Detección de la pupila en pc del modelo .onnx 48
Ilustración 29: Detección de la pupila en LattePanda del modelo .pt 49
Ilustración 30: Detección de la pupila en LattePanda del modelo .onnx 49

1 INTRODUCCIÓN

"¿Herederán los robots la Tierra? Sí, pero serán nuestros hijos"

-Marvin Minsky-

En este primer capítulo, con el objetivo de colocar en situación al lector de la presente memoria, se va a proceder a dar una breve introducción para dar a conocer los antecedentes de este trabajo y proporcionar una ligera idea de lo que se ha llevado a cabo.

Podemos llegar a saber multitud de cosas a través de la posición de los ojos y del movimiento de los mismos. Por ejemplo, está demostrado que un alto porcentaje de personas suelen mirar hacia arriba y a la izquierda cuando mienten. De la misma forma, se puede sospechar el estado anímico de una persona o cómo le está afectando los estímulos que está recibiendo por la velocidad con la que cambia de posición sus pupilas. Cosas como estas se pueden aprovechar para multitud de aplicaciones como en terapia a la hora de tratar el conflicto mental de un paciente o en marketing para saber en qué se fijan primero las personas al mirar la estantería de un supermercado.

Hasta no hace mucho, los productos desarrollados que permiten la interacción de lo que nos rodea usando solo la mirada eran cosa del futuro. A día de hoy ya es una realidad y esto se lo debemos a la velocidad a la que conseguimos avances tecnológicos y a cómo integramos estos avances en nuestra sociedad. Una prueba de ello es lo increíblemente complejos que son los *smartphones* y en cambio, prácticamente todo el mundo tiene como mínimo uno y lo usa a diario.

En la siguiente gráfica podemos ver con datos reales la cantidad de personas que usan teléfonos móviles inteligentes alrededor de todo el mundo en los últimos años. También se puede observar como la tendencia ha sido creciente y se espera que así siga siendo.

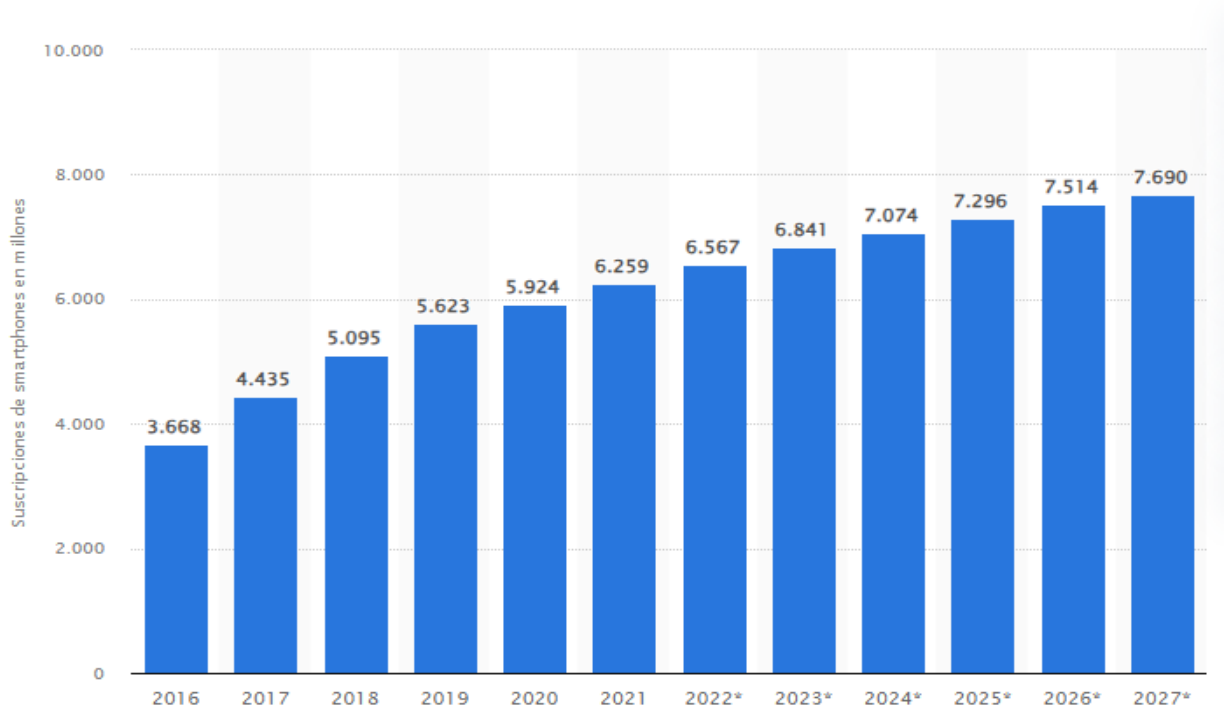

Ilustración 1: Número de usuarios de *smartphones* a nivel mundial. Fuente: Statista.

La cuestión es, que si esto ha sido así para los teléfonos inteligentes, ¿por qué no podría ser así para unas gafas que permitan controlar el ratón de un ordenador con la mirada? ¿O el uso de un dispositivo que altere el funcionamiento sensorial de la vista mientras se aplica estímulos para terapia emocional usando realidad aumentada y se recogen datos Las posibilidades que brindan el seguimiento ocular son inmensas, desde el campo de la medicina y el ocio hasta el simple hecho de cómo interactuamos con la tecnología.

Es por todo esto que lograr implementar una buena solución ante el problema del *gazetracking* en un dispositivo es tan importante y tiene tanta demanda en el mercado actual.

1.1 Motivación

El motivo principal por el que se ha querido llevar a cabo este proyecto es porque hay una alta demanda en el mercado tecnológico de este tipo de algoritmos y el uso de redes neuronales convolucionales está siendo cada vez más usado debido a las numerosas ventajas que posee.

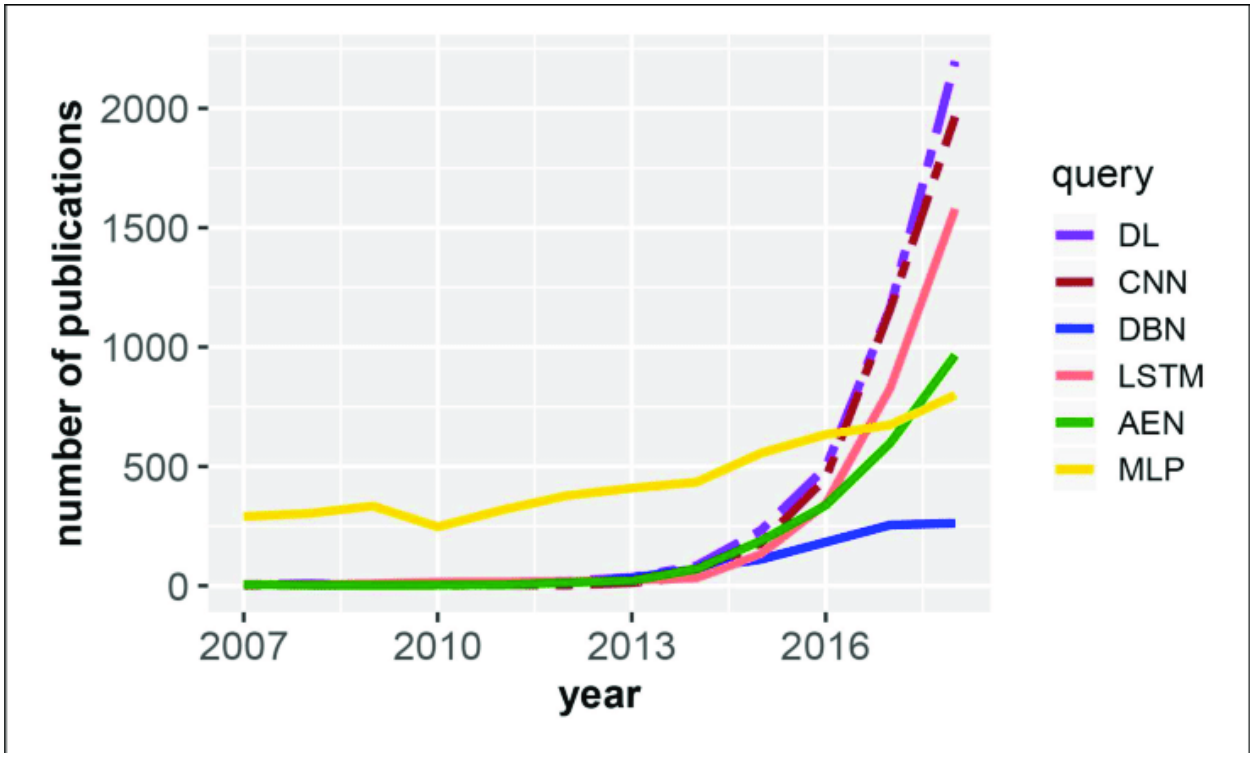

Ilustración 2: Número de publicaciones por año en el que vemos el aumento de las relacionadas con las CNN. Imagen de: https://www.researchgate.net/publication/339578120_An_Introductory_Review_of_Deep_Learning_for_Prediction_Models_With_Big_Data

Un dispositivo portátil que sea capaz de leer el movimiento de los ojos en tiempo real tendría un mercado tremendo en el mundo actual. La industria de los videojuegos lleva creciendo desde hace más de 20 años y algo así podría redefinir la forma en la que se juega. El mundo de la realidad virtual tendría unas mejoras muy significativas.

En segundo plano y como algo más personal, pienso que la implementación de este tipo de tecnologías en medicina para el uso terapéutico de trastornos mentales y el tratamiento de los conflictos psicológicos que sufren muchas personas podría revolucionar este campo. El diseño de un dispositivo que sea capaz de leer cómo le afecta un estímulo a un paciente mediante el visionado de sus ojos puede llegar a convertirse en una herramienta tremendamente útil para psicólogos y psiquiatras, hacienda incluso que la terapia se pudiera hacer desde casa, donde hasta las personas más introvertidas se sentirían más cómodas para poder expresarse y contar todo lo que sienten.

1.2 Objetivos a cumplir

El presente proyecto pretende desarrollar una solución ante el problema del seguimiento de la mirada mediante el uso de herramientas creadas a partir de inteligencia artificial. Para dicho fin, se va a hacer uso de la tecnología conocida como YOLO (You Only Look Once), concretamente de YOLOv8 desarrollado por Ultralytics.

El proyecto podría constar de 2 partes diferenciables, el hardware que usaremos para la creación del dispositivo y la parte del desarrollo e implementación de la red neuronal convolucional que

forma YOLOv8. No obstante, el alcance de este proyecto es la implementación de una solución software, dejando como posibles mejoras la creación de hardware para un desarrollo completo del dispositivo portátil.

En la parte hardware, vamos a proceder a usar una LattePanda, un miniordenador de placa única que nos ha facilitado el departamento. Por otra parte, vamos a desarrollar y fabricar un cerramiento para este hardware, con el objetivo de mantenerlo protegido y poder tener la posibilidad de añadir mejoras mediante la modificación de dicho cerramiento. Por último, decir que lo apropiado sería usar un *pupil labs* pero que, debido a la escasez de los mismos, hemos realizado todo el proyecto usando una cámara tipo webcam.

En la parte del software, hemos procedido a entrenar varias redes neuronales convolucionales, siendo todas de tipo YOLOv8, quedándonos con la que nos arrojó mejores resultados. Hemos usado un dataset bastante amplio de multitud de ojos diferentes y utilizado aprendizaje de tipo supervisado para el entrenamiento de las redes.

Por lo tanto, podríamos concluir este apartado con que para cumplir con el objetivo de este proyecto vamos a tener que afrontar una serie de requisitos en la parte de hardware (Rhard) y otros en la parte de software (Rsoft). Además, incluiremos objetivos que no será necesario cumplir para este proyecto pero que podemos admitirlos como posibles mejoras para conseguir un dispositivo completamente portable (Rhard/soft-opc).

1.2.1 Requisitos hardware

- Rhard-1: Diseño y fabricación del cerramiento para el dispositivo.
- Rhard-2: Empleo de LattePanda para el prototipo.
- Rhard-3: Uso de webcam para la implentación de la red neuronal.
- Rhard-opc: Entorno de alimentación para el dispositivo portable.

1.2.2 Requisitos software

- Rsoft-1: Desarrollo e implementación de una red neuronal tipo YOLO para el problema de *gazetracking*.
- Rsoft-2: Adquisición de datos fiables y diversos para el entrenamiento de la red neuronal.
- Rsoft-3: Optimización del modelo para su implementación en dispositivos con menos recursos.

2 ESTADO DEL ARTE

"Una máquina puede hacer el trabajo de 50 hombres ordinarios, pero no existe máquina que haga el trabajo de un hombre extraordinario"

-Elen G. Hubbard-

Para poder entender este proyecto, es necesario explicar los antecedentes del *gazetracking*, qué es una red neuronal de convolución y cómo se pueden implementar, cómo funciona el ojo humano y los algoritmos de *gazetracking* que se están usando en la actualidad.

En primer lugar se va a hacer un recorrido al lector por la historia del estudio del movimiento de los ojos. Así se podrá hacer una idea de la importancia que podría llegar a tener los datos obtenidos por este algoritmo.

En el siguiente apartado se va a abordar el tema de las redes neuronales convolucionales. De esta forma se proporciona un atisbo de la relevancia que están adquiriendo estas redes en el mundo de la inteligencia artificial y la automatización de procesos. Aquí se hablará de cómo se implementan las redes neuronales de forma física.

Finalmente se desarrollará un apartado dedicado a los modelos que existen en la actualidad, centrados en el seguimiento del movimiento y detección de objetos, donde se podrá ver que existen multitud de algoritmos que llevan a cabo esta función.

2.1 Antecedentes del *gazetracking*

El *gazetracking* o seguimiento de ojos se define como el proceso llevado a cabo para poder conocer el punto donde se fija la mirada o cómo se mueve el ojo respecto del movimiento de la cabeza del individuo. El proceso se utiliza en campos muy diversos tales como la psicología, el diseño de productos, la medicina o la robótica.

Se podría pensar que este proceso es algo relativamente actual, pero nada más lejos de la realidad. Los primeros rastreadores del movimiento ocular datan de finales del siglo XVIII cuando Wells, utilizando imágenes residuales, hace un estudio con el objetivo de describir el movimiento de los ojos (Porterfield, 1737).

A finales del siglo siguiente, se realizaron estudios (Javal y Lamare, 1879 y 1892 respectivamente) sobre el movimiento brusco de los ojos. Estos llevaron a cabo cuando Javal se

percató de que durante la lectura no se realiza un barrido suave del movimiento ocular a lo largo del texto como se suponía hasta la fecha, sino que los ojos se paraban brevemente en ciertas posiciones y de unas a otras la mirada realizaba movimientos bruscos (llamados sacadas). Estos autores establecieron que un mecanismo de acoplamiento de los ojos a los oídos para hacer audibles los movimientos de los ojos. Después de estos estudios y basándose en los mismos, Ahrens, Huey y Delabarre (1891, 1898 y 1898 respectivamente) fueron los primeros que intentaron registrar el movimiento de los ojos. Para ello usaron una capa de hollín sobre una superficie conectada mediante un cable al globo ocular.

En 1901, Dodge y Cline establecieron el primer método capaz de recoger movimientos oculares reflejando una luz de una fuente externa directamente en la fóvea. El aparato con el que realizaban este método usaba un método fotográfico y los reflejos de la luz sobre el ojo para grabar estos movimientos en dirección horizontal. Años después, en 1905, Judd, McAllister y Steel aplicaron la cinematografía para analizar el movimiento de los ojos. De esta forma, podía observar estos órganos fotograma por fotograma, algo que mejoró significativamente la calidad de los datos recogidos. Gracias a estos, Guy Thomas Buswell pudo, en 1935, crear el primer sistema de seguimiento ocular no intrusivo. Su método consiste en aplicar haces de luz que se reflejan en el ojo y luego los graba en una película. Guy Thomas realizó sus estudios diferenciando a los sujetos por edad y nivel educativo, de donde se pudo concluir que el movimiento ocular depende de estos factores de forma directa.

En 1939, Jung midió los movimientos oculares, tanto vertical como horizontales, mediante la colocación de diversos electrodos cerca de los ojos que era capaces de medir los campos eléctricos del globo ocular. Este método se llama electrooculografía (EOG) y permitió el procesamiento en tiempo real de los datos de la Mirada gracias a la tecnología analógica de la época.

En 1947, Paul Fitts usó cámaras de cine para registrar los movimientos oculares de los pilotos durante el aterrizaje. Este fue el primer estudio realizado de usabilidad con ayuda del seguimiento ocular y el objetivo era saber cómo los pilotos usaban los controles de vuelo.

En la década de los setenta se llevaron a cabo numerosos estudios en este campo y el auge de este método fue muy significativo. Los rastreadores oculares pasaban a ser mucho menos intrusivos gracias a las mejoras tecnológicas, sobre todo al desarrollo de la electrónica, aumentando su precisión y empezando a detector la localización del centro de la mirada gracias a la detección de la pupila y el reflejo de la córnea (Merchant, 1974). En este período la psicología se interesó notablemente en esta metodología para multitud de estudios, sobre todo enfocados en el estudio de la percepción y los fenómenos cognitivos.

En 1981, Bolt presenta su sistema de *eye tracking*, un sistema que permitía la interacción persona-ordenador. Esto se pudo hacer debido a la mejora en la potencia de los ordenadores, lo que permitió hacer el seguimiento ocular en tiempo real. Fue gracias a esto que los rastreadores oculares comenzaron a usarse en medicina con personas con discapacidad.

Desde la década de los 90 hasta ahora, debido a la mejora continua de la electrónica y la computación, los costes de los dispositivos de rastreo ocular se han abaratado notablemente y esto ha hecho que su uso en otros campos se dispare. Hoy en día se usan una infinidad de métodos para llevar a cabo el rastreo de los ojos, aunque uno de los métodos más usados se desarrolló en 2003 de la mano de Jacob y Karn. Su dispositivo permite el movimiento de la cabeza mientras se recoge el movimiento de las pupilas mediante el brillo de la misma y el

reflejo de la córnea.

En cuanto a sectores en los que se utiliza, la tecnología de seguimiento ocular se usa en multitud de campos: estudios sobre discapacidad (Hornof and Cavender, 2005), comercialización y marketing (Maughan, 2007), psicología (Jhonson, 2003), y diversos estudios aplicados a factores desarrollados por humanos. Este último apartado es muy amplio y va desde la gestión de sistemas de vuelo (Frische, 2011) y habilidades de conducción (Martens and Fox, 2007) hasta la investigación en la usabilidad y la interacción humano-ordenador (HCI).

Se adjunta una gráfica que muestra el número de publicaciones por año que se han hecho sobre *eye tracking*, obtenido de la base de datos Scopus. Aquí se puede ver cómo en los últimos años esta tecnología está tomando una gran relevancia en el mundo (se tienen datos hasta principios de 2020).

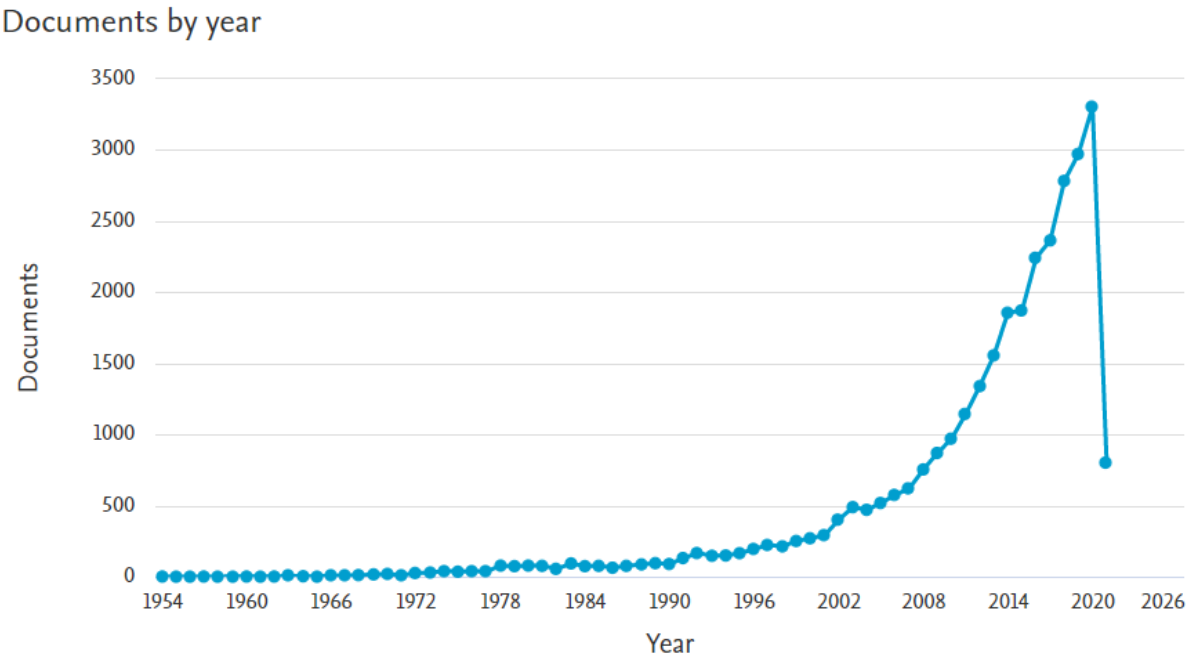

Ilustración 3: Número de publicaciones sobre *eye tracking* por año. Fuente: Scopus.

2.2 Cómo funciona el ojo humano

En este apartado se va a proceder a explicar de forma detallada cómo funciona el ojo humano ya que conocer esto puede ayudar al entendimiento de cómo encarar el sistema de captación del punto de la mirada y de los movimientos del ojo.

El ojo humano es un óptimo sistema convergente que posee la capacidad de formar imágenes invertidas sobre la capa sensible de la retina situada en el fondo del interior del globo ocular. La función principal del ojo es traducir las vibraciones electromagnéticas de la luz que le llegan en

impulsos nerviosos que se transmiten al cerebro, siendo este último el que lleva a cabo el proceso de interpretación de la visión.

La estructura anatómica del ojo se divide en 4 partes:

- El **globo ocular**, que consiste en una estructura esferoide con un marcado abombamiento en su superficie anterior, de unos 2.5 cm de diámetro aproximadamente. La parte exterior se compone de 3 capas de tejido:
    - La **esclerótica**. Es la capa más externa y se encarga de proteger el globo ocular. Ocupa unas cinco sextas partes de la superficie del órgano.
    - La **úvea**. Es la capa media y se diferencian tres partes en ella: la coroides, el cuerpo ciliar y el iris.
    - La **retina**. Es la capa más interna, sensible a la luz y donde se "proyectan" las imágenes invertidas que capta el ojo.
- La **córnea**. Consiste en una resistente membrana por la que la luz entra al interior del ojo. Por detrás, separando la córnea del cristalino, se encuentra el humor acuoso. Esta parte está conectada con el músculo ciliar, encargado de redondear la lente para cambiar la longitud focal.
- El **iris**. Estructura pigmentada que se encuentra suspendida entre la córnea y el cristalino, con una apertura en el centro llamada pupila. Rodeando esta última hay un músculo que se encarga de reducir o aumentar su tamaño con el objetivo de controlar la cantidad de luz que entra al interior del ojo.
- El **humor vítreo**. Es un líquido gelatinoso, de mayor consistencia que el humor acuoso, que rellena el interior del ojo y se encarga de darle forma.

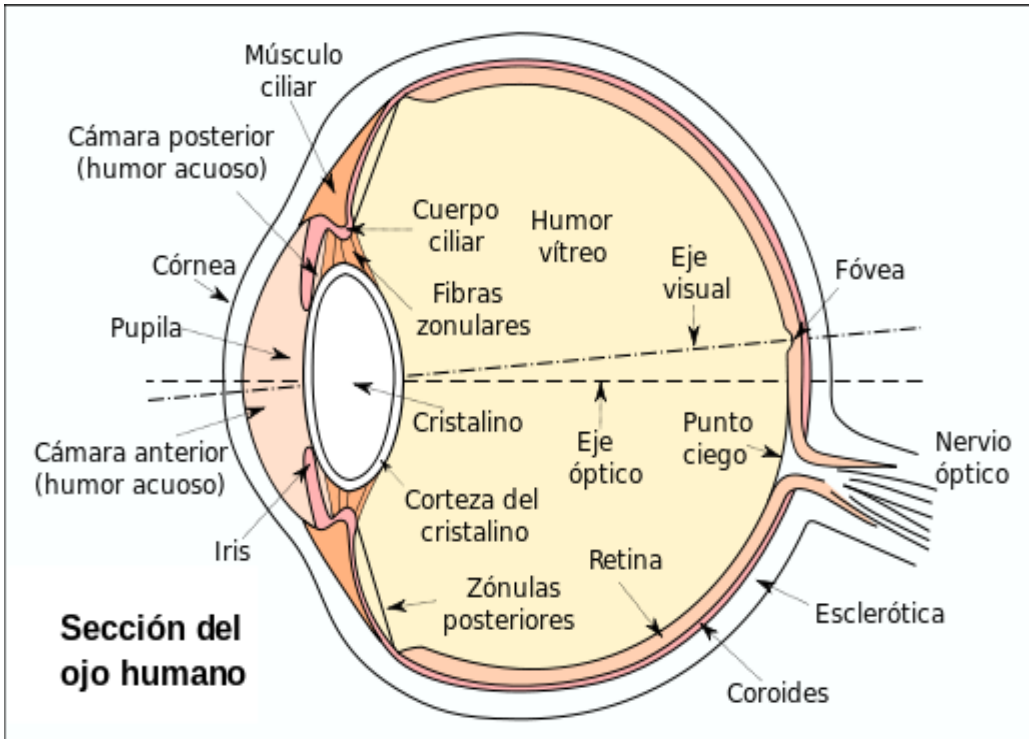

Ilustración 4: Anatomía del ojo humano. Imagen de Wikipedia: *https://commons.wikimedia.org/wiki/File:Eyesection-es.svg*

2.2.1 Sistema para el movimiento ocular

Si vemos cómo el ojo interpreta las señales de luz que le llegan, tenemos que la lente del cristalino hace que se forme en la retina una imagen invertida de los objetos a los que la luz llega y rebota entrando por la lente. Es decir, los rayos de luz se reflejan en los objetos y cuando estos rayos entran a través del cristalino se produce una imagen invertida en el fondo de la retina que se codifican y se transmiten al cerebro para que procese la información.

El campo visual se divide en varias zonas, aunque normalmente no somos conscientes debido a que el foco del ojo está en continuo movimiento y la retina se excita de distinta forma dependiendo de dicho foco. El movimiento del ojo posee seis grados de libertad, con tres rotaciones y tres traslaciones. En el ojo se encuentran seis músculos que se encargan del movimiento ocular, los músculos extraoculares. Del movimiento lateral el músculo recto medial y el lateral, mientras que de los movimientos verticales del ojo se encargan el recto superior e inferior y de los movimientos de torsión se encargan los oblicuos superior e inferior.

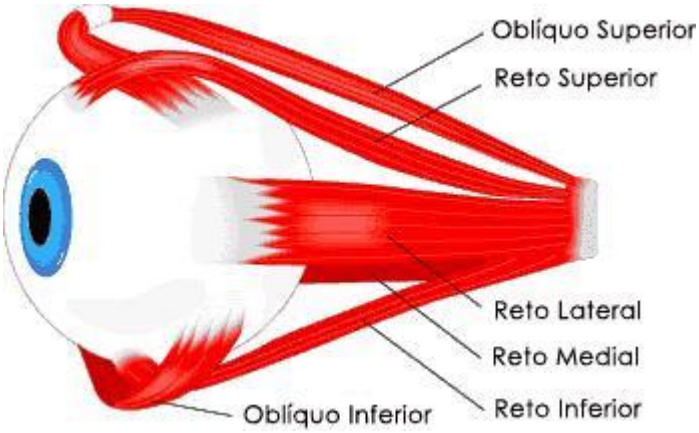

Ilustración 5: Músculos extraoculares. Imagen de Wikipedia: *https://commons.wikimedia.org/wiki/File:MUSCULOS_OCULARES.JPG*

El sistema oculomotor o sistema visual motor está formado por los músculos extraoculares, los nervios oculomotores y sus núcleos de origen. Los estímulos se integran en el tallo cerebral, provenientes de los movimientos binoculares, voluntarios y reflejos. La binocularidad es característica del movimiento ocular y por ello son necesarios los centros de coordinación. Estos se clasifican como centros corticales y centros subcorticales.

El sistema oculomotor posee cuatro funciones:

- Mantiene la fijación de la fóvea en el objeto en movimiento.
- Conduce los estímulos del campo periférico hasta el campo visual central.
- Capta y garantiza las imágenes en la fóvea.
- Dejar las imágenes en la fóvea cuando la cabeza se mueve.

2.2.2 Movimientos oculares

Existen 3 tipos de movimiento ocular (Gila et al., 2009):

- Los movimientos **vestíbulo-oculares** y los **reflejos optocinéticos**. Los primeros son movimientos automáticos que se producen para compensar los movimientos de la cabeza, mientras que los segundos sirven para estabilizar la imagen en la retina y tener la mirada fijada en un punto determinado.
- Los **movimientos sacádicos** y los de **seguimiento**. Los primeros son los movimientos voluntarios que se realizan para desplazar la mirada dentro del campo visual, mientras que los segundos son los que se realizan para perseguir con la mirada objetos móviles.
- **Temblor, microsacadas y derivas**, que son micromovimientos derivados de la fijación ocular.

El rastreo visual de una zona se realiza mediante una serie de movimientos sacádicos y los

movimientos de fijación entre ellos. Estos movimientos se pueden dar debido a acciones voluntarias o provocadas por estímulos que entran al campo de visión. También se dan movimientos sacádicos involuntarios durante la fase REM del sueño.

Existen algunos parámetros importantes a tener en cuenta si hablamos de los movimientos sacádicos. En la siguiente tabla de pueden ver:

Amplitud máxima	Lo máximo que se puede recorrer sin mover la cabeza son 30º
Duración	Esto depende de la amplitud del desplazamiento pero va desde los 40 ms hasta los 120 ms.
Velocidad máxima	La velocidad máxima se ha llegado a comprobar que está en 700º/s
Latencia	Es el tiempo entre la aparición de un estímulo y el inicio del movimiento sacádico. Se encuentra entre los 180 y los 300 ms.
Periodo refractario	Tiempo entre el final de una sacada y la disponibilidad del sistema para iniciar un nuevo movimiento. Está entre los 100 y los 200 ms.
Tiempo mínimo de fijación	Es la suma del periodo refractario y el tiempo de procesado cognitivo del objeto enfocado. Está entre los 200 y los 350 ms.

Analizando todo lo anterior, podemos concluir con que los movimientos a modelar necesarios para el rastreo visual son tres: sacádicos, de seguimiento y de fijación (Duchowski, 2007), por lo que será la identificación de estos tres movimientos los que se deben tener en cuenta debido a que son los que se corresponden con la atención visual voluntaria de una persona.

2.3 El uso de redes neuronales en el seguimiento de la mirada

2.3.1 Red neuronal convolucional

Se define como red neuronal convolucional a una red neuronal que ha sido diseñada de forma artificial mediante hardware y software. Las neuronas pertenecientes a dicha red corresponden a campos receptivos de forma muy parecida a como lo hacen las neuronas de un cerebro biológico.

El origen de las redes neuronales convolucionales fue en 1980 y vino de la mano de Kunihiko Fukushima. Años más tarde fue mejorado por Yann LeCun en 1998, cuando introdujo un método de aprendizaje que se basa en la retropropagación, mejorando el entrenamiento del sistema de forma muy significativa. En el año 2012 estas redes dieron otro salto gracias a Dar Ciresan entre otros, los cuales las implementaron en unidades de procesamiento gráfico y sus resultados fueron increíbles. A raíz de la pandemia provocada por Covid-19 se ha desarrollado una tecnología que utiliza estas redes neuronales para realizar un baremo de la cantidad de personas que se encuentran cerca de un objetivo conocido, tecnología conocida como *Crowd-counting*.

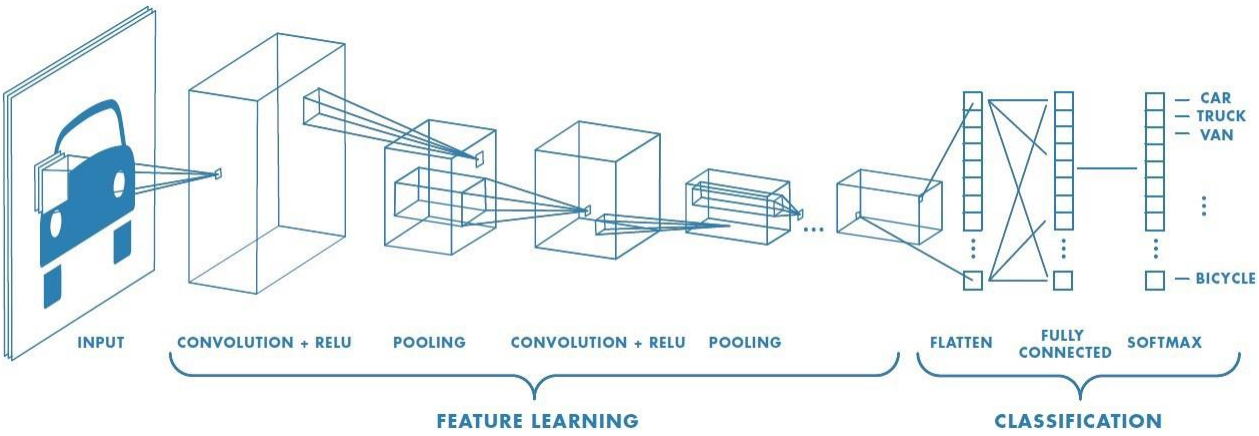

Ilustración 6: Estructura de una red neuronal convolucional. Imagen de: *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*

Estas redes con aprendizaje supervisado son multicapas y toman la inspiración para su funcionamiento del córtex visual de los animales. Existen multitud de tipos de redes neuronales y cada una de ellas tiene sus ventajas y aplicaciones, pero para el procesamiento de imágenes que es lo que nos ocupa, este tipo es el que mejor funciona. A grandes rasgos la arquitectura de estas redes consta de varias capas que se encargan de extraer las características de las entradas y de clasificar. La imagen se divide entonces en campos receptivos con el objetivo de detectar ciertas características de la imagen de entrada (líneas verticales, horizontal, etc…). Más tarde viene el *pooling* que consiste en reducir la dimensionalidad de las características extraídas dejando solo la información más importante. Luego se hace otra convolución y otro *pooling* que alimenta una red *feedforward* multicapa. Por último se tiene una salida de la red que es un grupo de nodos los cuales se encargan de clasificar el resultado.

Si se recurre a la programación tradicional, para poder predecir el resultado que se está buscando en base a una serie de entradas, habría que programar toda una serie de reglas de forma "artesanal". Estas reglas serán más numerosas a mayor número de entradas y condiciones tengan los resultados que estamos buscando. Como consecuencia, el código se volverá mucho más extenso y complejo, lo que puede llevar con toda probabilidad a errores, escasa escalabilidad del código y una dificultad tremenda a la hora de corregir dicho código.

Para poder evitar todo esto se recurre a la inteligencia artificial, concretamente a las redes neuronales para poder obtener una arquitectura de interconexiones que sea capaz de descubrir las relaciones existentes entre las entradas (capa de entrada), las capas intermedias (hidden layers) y las neuronas de salida (output layers). De esta forma el modelo que se crea tiene la capacidad de aprender por sí solo.

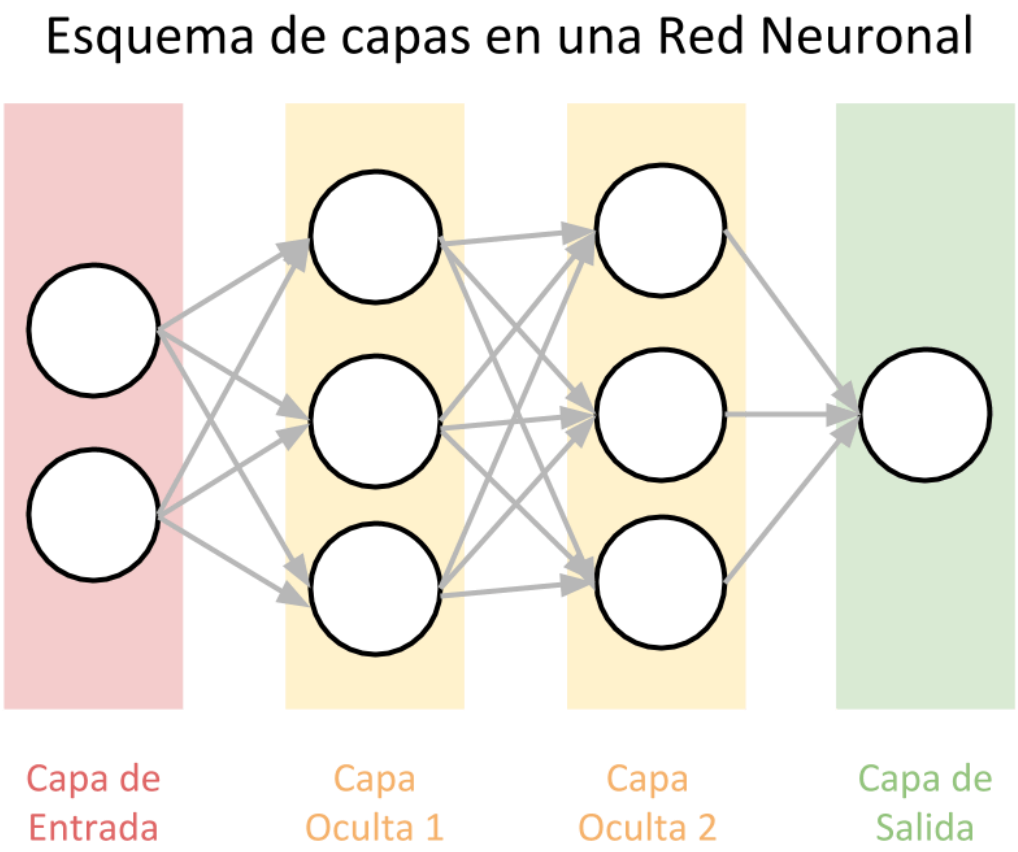

Ilustración 7: Esquema de red neuronal. Imagen de: https://www.aprendemachinelearning.com/aprendizaje-profundo-una-guia-rapida/

La capa de entrada se encarga de recibir los datos que va a manejar la red neuronal del exterior. Las capas intermedias o capas ocultas son las que se encargan de procesar la información que mandan las neuronas en la capa de entrada, haciendo cálculos matemáticos con dichos datos. Aquí se presenta uno de los retos a la hora de montar una red neuronal y es decidir cuántas capas tendrá la red y cuántas neuronas va a poseer cada capa. Por último se tiene la capa de salida que devuelve el resultado de la predicción realizada.

Ahora hay que hablar del cálculo de las predicciones. Cada conexión dentro de la red de neuronas lleva asociada un peso que determina la importancia que tendrá esa posible relación en la neurona al multiplicar dicho valor por el valor de entrada de esta neurona. Esto es muy importante ya que determina la probabilidad de los "razonamientos" que se están llevando a cabo por cada suceso que entra y sale de cada neurona.

Cada neurona artificial, como las neuronas biológicas, tiene una función de activación. Esta función determina si la suma de los valores que recibe la neurona supera el umbral que hace que la neurona se active. Si esto sucede, se dispara el valor de dicha neurona hacia la siguiente capa conectada. Existen multitud de funciones de activación que se pueden usar a la hora de implementar una red neuronal.

2.3.2 Cómo se implementan las redes neuronales convolucionales

Una red neuronal convolucional debe aprender por sí sola a reconocer ciertos objetos que se encuentren en las imágenes de entrada que le van llegando. Para ello lo primero que se va a necesitar son datos con los que la red pueda trabajar, es decir, un banco de muestras que contengan el mayor número posible de situaciones que debe tener en cuenta la red neuronal para poder predecir el resultado con exactitud.

Primero se debe tener en cuenta la información con la que se está tratando. No es lo mismo montar una red neuronal para calcular las probabilidades de que un equipo gane la liga de fútbol a intentar tratar de montar una red que sea capaz de identificar objetos en la visión que arroja una cámara. En este caso son imágenes y hay que mencionar que estas no son más que una matriz de pixeles. Se va a intentar escenificar las explicaciones con el ejemplo de la famosa "Lena".

Ilustración 8: L a famosa imagen "Lena" y sus valores de píxeles en código RGB. Imagen de: *https://www.researchgate.net/figure/RGB-color-values-of-some-pixels-in-Lenas-image_fig1_317351196*

Si se va a tratar con imágenes, una red neuronal va a tomar como valores de entrada los correspondientes a los pixeles de una imagen. Es decir, en el caso de que tuviéramos una imagen de 28x28 pixeles, es necesario montar 784 neuronas y eso si se tiene un solo color. Si hubiera varios colores se necesitaría tres canales (los correspondientes al RGB: red, Green and blue) y entonces se trataría de 28x28x3, es decir, 2352 neuronas solo en la capa de entrada.

Por este motivo es lógico pensar en realizar un preprocesado a la imagen y poder tener la misma en una escala de grises. Si hacemos esto, antes de comenzar las convoluciones se aplica un procesado a las imágenes. Si se le aplica a la ilustración 4 por ejemplo quedaría algo así:

Ilustración 9: Lena con un preprocesado aplicado a escala de grises. Imagen de: *http://kurauchi.com.br/post/means-medians-and-images/*

Después de esto se normaliza la imagen dividiendo todos los valores por 255, ya que como valores los pixeles pueden ir de 0 a 255 e una imagen y si se divide por el valor más alto que se pueda tener siempre quedará un valor que se encuentra entre 0 y 1.

Ahora es cuando comienza el proceso que distingue este tipo de red de las demás, las convoluciones. Estas consisten en tomar *neighbourhoods* dentro de la imagen de entrada, es decir, matrices más pequeñas (normalmente de 3x3 aunque esto se puede modificar para cogerlas más grandes) que van a ir operando junto con una matriz predefinida (como productos escalares) llamada Kernel. Esta matriz Kernel irá recorriendo todas las neuronas de la capa de entrada y va a generar una nueva matriz que será la siguiente capa de neuronas ocultas.

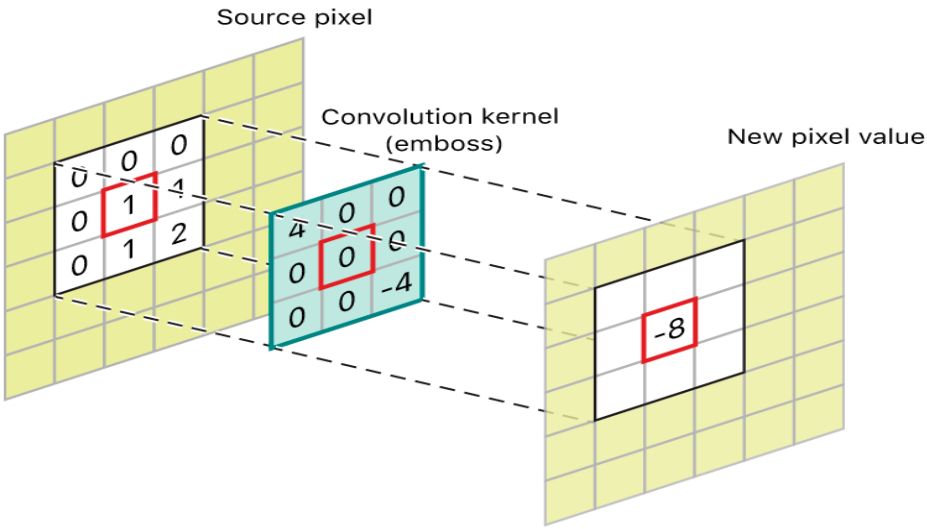

Ilustración 10: Muestra gráfica de cómo opera el Kernel. Imagen de: *https://developer.apple.com/documentation/accelerate/blurring_an_image*

Si la imagen fuera a color, el Kernel debería ser de 3x3x3, un filtrado de 3 Kernels de 3x3. Más tarde esos 3 se suman y conforman una salida como si de un solo canal se tratase y no de 3 distintos, pero este no es el caso para Lena ya que se ha pasado a escala de grises. La matriz Kernel al principio tomará valores aleatorios y se irá ajustando mediante *backpropagation*. Una alternativa para mejorar es hacer que siga una distribución normal siguiendo simetrías.

Una vez obtenido la convolución con el Kernel llega el momento de usar la función de activación. Para este tipo de redes neuronales la más usada es la conocida como ReLu, que proviene de Rectifier Linear Unit y se define como $f(x) = \max(0, x)$. Gracias a esto se obtiene un mapa de detección de características, lo que implica la primera detección de patrones que va a ir buscando la red en la imagen.

Una vez se tenga este mapa, hay que realizar un filtrado para poder reducir el número de neuronas. Hay que tener en cuenta que con una imagen de 28x28 pixeles, algo muy pequeño, habría una capa de entrada con 784 neuronas y que tras una primera convolución pasarían a ser 25088 neuronas (correspondiente a los 32 mapas de 28x28). Y tras cada convolución la siguiente capa se iría complicando de forma exponencial a niveles en los que la capacidad de computación necesaria sería demasiado grande.

Por ello se realiza un filtrado, conocido como *subsampling*. Para realizar este filtrado lo que se hace es dejar solo las características más significativas que detecta cada filtro. El más usado es el *Max-Pooling*, que consiste en dejar en una matriz del tamaño elegido solo los valores más grandes en cada subgrupo. Por ejemplo, si se hace un *Max-Pooling* de 2x2, se recorre cada una de las 32 imágenes características de 28x28 en matrices de dicho tamaño (4 pixeles cada vez) y se va preservando el valor más alto cada vez. De esta forma se obtienen 32 imágenes de 14x14, pasando de tener 25088 neuronas a 6272, reducción más que considerable.

Hasta aquí la red será capaz de detectar formas primitivas como curvas o rectas. Para poder ganar precisión, hay que realizar más convoluciones. Para el caso que nos ocupa de una imagen de entrada de 28x28 pixeles, se puede hacer dos convoluciones más de este tipo. Con la siguiente, aplicando el mismo proceso, con una entrada de 14x14x32 se obtiene una salida de 7x7x64 y con la última la salida sería de 3x3x128.

Por último, se toma la última capa y se "aplana" a una nueva capa tradicional, que será la última capa oculta de la que dispondrá la red. Para este caso, con una de 100 neuronas *feedforward* debería ser óptima. A esta capa se le aplica la función exponencial normalizada o "softmax", que normalizará la salida a una distribución de probabilidad sobre las clases de salida a predecir.

Por último se tendrá la capa de salida final, que tendrá la misma cantidad de neuronas que de objetos a detectar con sus respectivas probabilidades. Es decir, si en una imagen queremos clasificar perros y gatos (estas serían las 2 neuronas de salida) y se obtiene [0.3 0.7], lo que nos arroja la red neuronal es que con un 30 % de probabilidad será un perro y con un 70 % de probabilidad será un gato.

2.3.3 Modelos neuronales más populares en la actualidad

2.3.3.1 SSD (Single Shot MultiBox Detector)

Single Shot MultiBox Detector (SSD) es un modelo de detección de objetos que ha sido ampliamente adoptado en la comunidad de visión por computadora debido a su eficiencia y precisión en la detección de objetos en imágenes y videos. SSD se destaca por ser capaz de detectar objetos de diferentes tamaños y escalas en una sola pasada (single shot), lo que lo hace particularmente adecuado para aplicaciones en tiempo real.

SSD combina las ventajas de las redes de una sola etapa y de dos etapas para la detección de objetos. El enfoque de una sola etapa se caracteriza por la eficiencia en términos de velocidad, ya

que realiza la detección y la localización en una única pasada. A diferencia de algunos modelos de dos etapas, no requiere la generación previa de regiones propuestas.

El funcionamiento de SSD se basa en la idea de dividir la imagen de entrada en múltiples cuadrículas (grillas) de diferentes tamaños. Para cada cuadrícula, se generan múltiples cajas delimitadoras (bounding boxes) de diferentes relaciones de aspecto y tamaños. Estas cajas delimitadoras se utilizan para predecir la presencia de objetos y ajustar las coordenadas de las cajas para que coincidan con los objetos reales en la imagen.

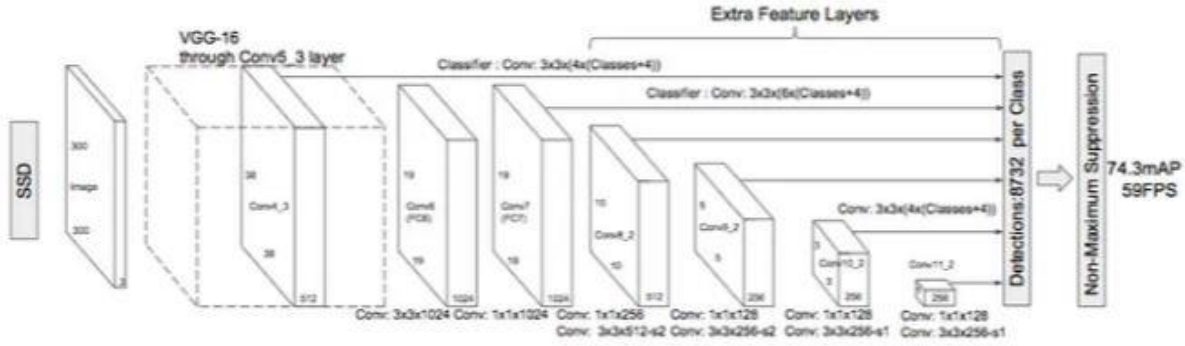

Ilustración 11: Arquitectura de un detector de tipo SSD. Imagen de: *https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab*

Como se puede apreciar en el diagrama anterior, la arquitectura de SSD se basa en la venerable arquitectura VGG-16, pero prescinde de las capas completamente conectadas. La elección de VGG-16 como red base se debe a su sólido desempeño en tareas de clasificación de imágenes de alta calidad y su popularidad en problemas donde el aprendizaje por transferencia contribuye a mejorar los resultados. En lugar de las capas completamente conectadas originales de VGG, se agregó un conjunto de capas convolucionales auxiliares (a partir de conv6 en adelante), lo que permite extraer características a múltiples escalas y disminuir progresivamente el tamaño de la entrada en cada capa subsiguiente. Esto facilita la detección de objetos en diferentes tamaños y escalas en una imagen, lo que es esencial para la detección de objetos en tiempo real.

2.3.3.2 Faster R-CNN

El modelo Faster R-CNN, que significa "Region-based Convolutional Neural Network" más rápido, es una innovadora arquitectura en el campo de la detección de objetos en imágenes. Este modelo ha demostrado ser altamente efectivo y preciso en la localización y clasificación de objetos dentro de imágenes o secuencias de video, y ha marcado un hito en esta área de la visión por computadora. Para lograrlo, se basa en una combinación de técnicas avanzadas, incluyendo redes neuronales convolucionales (CNN), regiones de interés (ROI) y redes neuronales recurrentes (RNN).

La arquitectura del modelo Faster R-CNN consta de dos componentes principales: una red convolucional para la extracción de características y un módulo de propuesta de regiones. La red convolucional, que puede ser una red base como VGG-16 o ResNet, se utiliza para extraer características de alto nivel de la imagen de entrada.

El módulo de propuesta de regiones es uno de los aspectos más innovadores de Faster R-CNN. Este módulo genera candidatos a regiones de interés en la imagen, lo que significa áreas donde es probable que se encuentren objetos. Utiliza una red neuronal especializada llamada RPN (Region Proposal Network) que evalúa múltiples ubicaciones y tamaños de posibles regiones de interés. Esto permite que el modelo se centre en áreas prometedoras y no en toda la imagen, lo que mejora significativamente la eficiencia computacional.

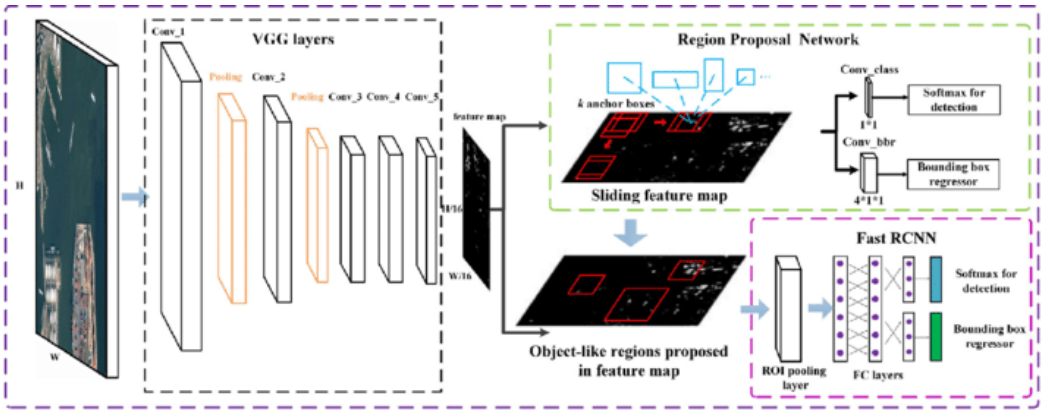

Ilustración 12: Arquitectura de una red de tipo Faster R-CNN. Imagen de: *https://www.researchgate.net/figure/The-architecture-of-Faster-R-CNN_fig2_324903264/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6Il9kaXJlY3QiLCJwYWdlIjoiX2RpcmVjdCJ9fQ*

La implementación de Faster R-CNN generalmente implica tres etapas clave: entrenamiento, inferencia y evaluación. Durante la fase de entrenamiento, se alimenta al modelo con un conjunto de datos etiquetado que contiene imágenes y las ubicaciones y categorías de los objetos en esas imágenes. El modelo ajusta sus pesos y parámetros para aprender a detectar y clasificar objetos.

En la etapa de inferencia, el modelo se utiliza para detectar objetos en nuevas imágenes o secuencias de video. Las regiones de interés se generan y se evalúan en función de las clases de objetos, lo que permite identificar y localizar objetos en las imágenes.

Finalmente, en la etapa de evaluación, se compara el rendimiento del modelo con un conjunto de datos de prueba para medir su precisión y rendimiento en la detección de objetos.

2.3.3.3 YOLO

El algoritmo YOLO, que significa "You Only Look Once" (Solo Mira una Vez), es un enfoque revolucionario en el campo de la detección de objetos en imágenes y videos. Desarrollado por Joseph Redmon y Santosh Divvala en 2016, YOLO ha redefinido la forma en que abordamos la tarea de detectar objetos en tiempo real con alta precisión y eficiencia computacional.

El concepto fundamental de YOLO es la detección de objetos en una sola pasada a través de la red neuronal. A diferencia de los enfoques tradicionales que dividían la tarea en múltiples etapas, como la generación de propuestas de regiones y la clasificación posterior, YOLO procesa la imagen completa de manera conjunta y predice simultáneamente las coordenadas de los cuadros delimitadores (bounding boxes) y las etiquetas de clase para todos los objetos en la imagen.

Para lograr esto, YOLO divide la imagen de entrada en una cuadrícula y asigna cada objeto a la celda que contiene su centro. Cada celda de la cuadrícula predice cuántos cuadros delimitadores contendrá (generalmente uno o ninguno) y ajusta sus coordenadas y etiquetas de clase. Esto da como resultado una salida eficiente y compacta que describe todos los objetos detectados en la imagen.

El impacto de YOLO en la visión por computadora es innegable. Su capacidad para detectar objetos en tiempo real ha encontrado aplicaciones en una variedad de campos, desde vehículos autónomos y sistemas de vigilancia hasta la atención médica y la robótica. La eficiencia y la precisión de YOLO lo han convertido en una herramienta esencial para resolver problemas de detección de objetos. Es por esto que hemos escogido una variante de este tipo de modelo para realizar nuestro proyecto, así que nos centraremos en explicar en profundidad cómo funciona más adelante.

2.3.4 Aplicaciones en la actualidad

Aunque como ya se ha mencionado antes esta tecnología tuvo sus comienzos hace 3 siglos, las aplicaciones más interesantes y que han causado un verdadero impacto en la sociedad se han llevado a cabo en los últimos 20 años, con un auge de esta tecnología en los últimos diez.

Los usos que se le están dando actualmente a esta tecnología pueden entrar dentro de dos categorías diferentes, una pasiva y otra activa (también llamadas de diagnóstico). La primera proporciona la evidencia objetiva y cuantitativa de los procesos que se encuentran relacionados con la percepción del usuario mediante la visión. En la segunda categoría los dispositivos rastreadores tienen la misión de hacer que el usuario interactúe con la máquina. Algunos campos donde se usan son:

- **Investigación científica**. El área con más usos. Campos como la neuropsicología, oftalmología, lingüística, psicología cognitiva, entre muchos otros, usan este tipo de rastreadores en la actualidad, ya que permiten el análisis de los procesos cognitivos respecto a la atención visual del individuo.

- **Investigación de mercados**. Un campo en auge dado que este método ofrece datos sobre la respuesta de los clientes ante estímulos provocados. Hacer test de embalajes, de páginas web o de distribución de productos son algunos de los usos.

- **Experiencia del usuario**. Otro sector con un auge tremendo. La realidad virtual hace que este método esté presente en la industria de los videojuegos y de los videos interactivos, un sector que lleva creciendo más de 20 años.

- **Interacción humano-máquina**. Gracias al desarrollo de esta tecnología en este campo se ha mejorado notablemente la interacción con cualquier máquina donde se implemente. Un gran dentro de esta área es el desarrollo de dispositivos para discapacitados, permitiéndoles manejar sillas de ruedas motorizadas y computadores entre otros dispositivos.

- **Mejora del rendimiento deportivo**. En el entrenamiento para atletas profesionales, donde se estudian sus reflejos y tiempos de reacción, este método se aplica mediante dispositivos para mejorarlos. También se ha usado en baloncesto para mejorar los tiros a canasta de los jugadores.

- **Estudios de ergonomía**. En el sector aeroespacial y automovilístico esto es muy usado para multitud de funciones. Entre estas destacan el control del tráfico aéreo, para establecer los paneles de mando o para hacer estudios sobre conducción vial.

Y estos son solo los campos más importantes actualmente, aunque salen nuevas aplicaciones cada año y no hay duda de que al recorrido de mejora y de usos para el *eyetracking* aún le queda mucho por dar.

2.4 Conclusión

Tras el análisis del estado del arte realizado en este capítulo, se deriva la conclusión de que la mejor alternativa para un sistema de seguimiento ocular es la implementación de un modelo basado en redes neuronales convolucionales (CNN)

Entre las razones en las que nos apoyamos para esta decisión están:

- La literatura consultada denota una alta precisión en el reconocimiento de objetos mediante imágenes respecto de otros algoritmos.
- Las redes neuronales convolucionales han visto un gran desarrollo en la última década, lo que ha facilitado la consulta en la literatura sobre facetas avanzadas del campo de la inteligencia artificial.
- El desarrollo se puede llevar a cabo con una sola CNN, tanto para la captación de objetos como la salida de los resultados, por lo que se simplifica todo el proceso.
- Por último, debemos añadir que hay muchas redes neuronales de este tipo basadas en librerías de tipo open source, lo que facilita la consulta de desarrollos y motiva al uso y aportación de las mismas.

El siguiente punto lo vamos a centrar en los principios de las CNNs y en las partes más relevantes de su arquitectura. Más tarde, revisaremos varios enfoques con los que poder desarrollar dicha red para el seguimiento de la mirada.

3 DESARROLLO DE LA SOLUCIÓN

"Cualquier tecnología lo suficientemente avanzada es indistinguible de la magia"

-Arthur C. Clarke-

En este trabajo final de grado se ha llevado a cabo un prototipo de un dispositivo portátil para el seguimiento de la mirada mediante el uso de inteligencia artificial. Y, aunque el proyecto requiere de desarrollos de otras áreas ajenas al modelo de red neuronal, es este modelo la punta de lanza con la que podemos afrontar el problema y llevar a cabo nuestra solución.

3.1 Desarrollo del hardware

3.1.1 Diseño e impresión 3D

El primer apartado que vamos a tratar del hardware usado en el proyecto es el cerramiento que hemos fabricado para la placa. Y es que en la era de la innovación tecnológica y la personalización, no podíamos afrontar la aventura que es el desarrollo de un dispositivo (o al menos gran parte del mismo) portátil sin diseñarle su cerramiento apropiado a medida.

El cerramiento desempeña un papel esencial en cualquier proyecto que involucre componentes electrónicos, ya que no solo proporciona protección contra el entorno externo, sino que también contribuye significativamente a la estética y funcionalidad general del dispositivo. La fabricación de un cerramiento a medida es especialmente valiosa, ya que se adapta perfectamente a las dimensiones y requisitos específicos de la LattePanda que vamos a usar, asegurando un ajuste preciso y una presentación visualmente atractiva.

Para esta parte hemos usado 2 tipos de software diferentes, uno de diseño 3D llamado Fusión 360, especialmente indicado en industria para el diseño de piezas de todo tipo. Gracias a este software podemos diseñar todo lo que seamos capaces de imaginar para, posteriormente, introducir nuestros diseños en un laminador que es capaz de facilitarnos el archivo que necesita nuestra impresora 3D para la fabricación del producto. Nosotros aquí no hemos tenido que hacer grandes cambios, ya que hemos usado un modelo previamente diseñado por el autor "ProjectSBC", dentro de la plataforma "Thingverse".

En nuestro caso el laminador que hemos usado se llama Cura. Este software es capaz de dividir en capas los archivos previamente diseñados y transformar estos en los famosos ".gcode", archivos necesarios que introduciremos mediante una tarjeta SD en nuestra impresora 3D. En nuestro caso hemos usado una Ender 3 S1, impresora de la que ya disponíamos en casa.

Además de las piezas, cuyas imágenes mostraremos a continuación, hemos usado varios imanes de neodimio para facilitar la apertura y el cierre de la caja sin necesidad de cerraduras ni pestañas extras fáciles de romper.

Vamos a asegurarnos a continuación de proporcionar una visión del conjunto de piezas usadas para crear el cerramiento a medida:

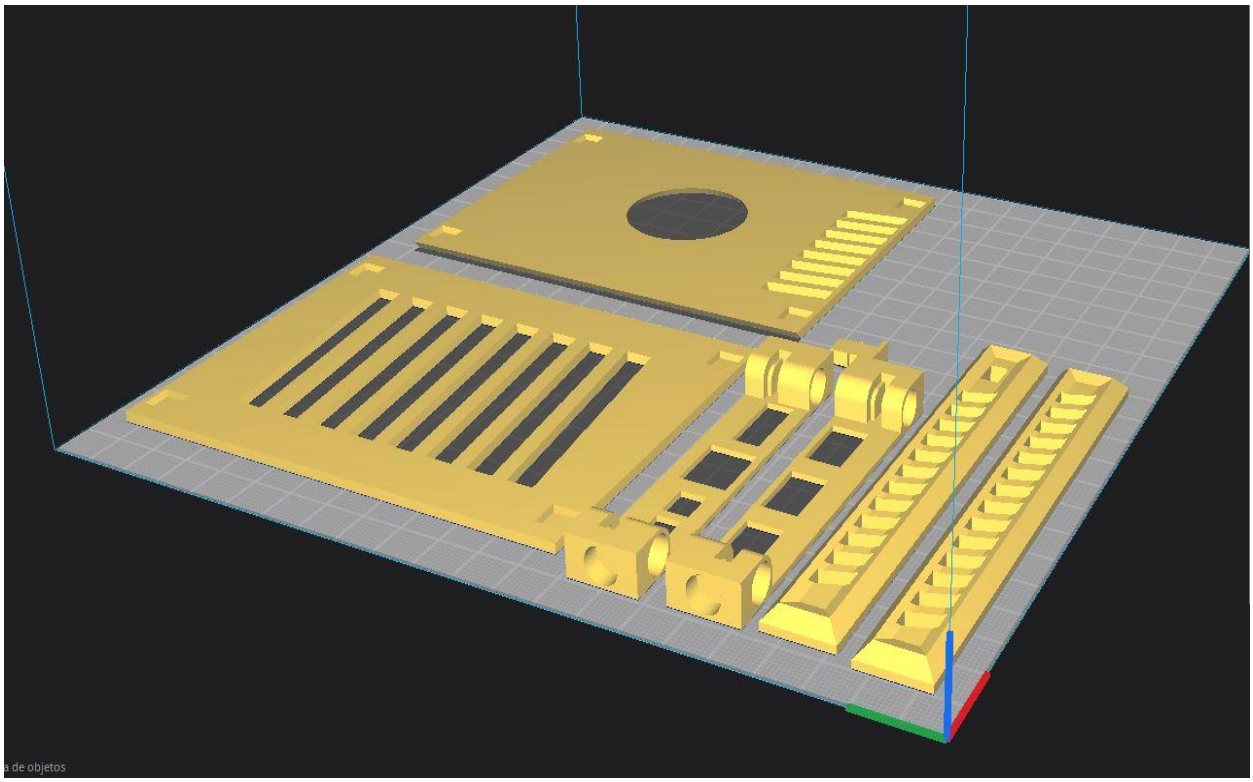

Ilustración 13: Conjunto de piezas que forman el cerramiento en Cura.

Las 2 piezas de mayor tamaño son las que cierran por la cara del ventilador de la placa y su cara opuesta. Se puede observar como se han dejado los huecos necesarios para la debida refrigeración de la misma, permitiendo así el correcto funcionamiento del ventilador. Se pueden observar además cuatro orificios cuadrados en las esquinas de ambas piezas, que es donde hemos pegado los imanes para la apertura y cierre de la caja.

Si seguimos recorriendo la imagen desde las piezas descritas, nos encontramos con las piezas frontales de la carcasa, piezas que tienen igualmente orificios donde alojar los imanes de neodimio y que además poseen aperturas para todas las entradas de periféricos que tiene la placa. A su lado se puede observar una pequeñapieza hecha como pulsador del botón de encendido.

Por último tenemos las 2 piezas que forman la cobertura lateral del cerramiento. Estas piezas están diseñadas con el objetivo de no permitir una acumulaión excesiva de temperatura, ofrecer protección extra a la placa eletrónica y poder colocar el botón para el pulsador de encendido.

Ilustración 14: Conjunto de piezas que forman el cerramiento en Cura. Imagen de: https://www.thingiverse.com/thing:3860552

Añadir como último punto que gracias a esta personalización del cerramiento y construcción modular, sería sencillo crear diferentes partes que tengan como objetivo añadir más funcionalidades, como circuito de alimentación autónoma, pantallas, ampliación de GPU, etc.

3.1.2 Uso en ordenador y en LattePanda

Este apartado será breve ya que abordaremos en mayor profundidad el uso del modelo cuando expliquemos los resultados que nos ha arrojado el mismo en cada uno de los dispositivos. Por ello, nos centraremos en qué dispositivos hemos usado para el desarrollo e implementación del modelo YOLOv8.

En primer lugar, tenemos que mencionar que el equipo con el que hemos entrenado la red neuronal es mi ordenador portátil personal. Se trata de un MSI GL63, el cual posee una GPU NVIDIA GeForce GTX 1050 Ti. Esta gráfica no es demasiado potente y está un poco limitada en cuanto a recursos comparada con sus versiones superiores más actuales, pero ha sido

suficiente para entrenar la red neuronal.

Por otra parte, disponemos de un “ordenador de bolsillo” que nos ha facilitado el departamento, una LattePanda ALPHA. Este equipo no se ha usado para entrenar la red debido a que los recursos son mucho más limitados que los de mi propio ordenador, pero sí hemos podido ejecutar el modelo de YOLOv8 una vez entrenado en el portátil.

Ilustración 15: Conjunto de piezas que forman el cerramiento en Cura. Imagen de: *https://www.lattepanda.com/lattepanda-alpha*

Esta placa consta, entre otros atributos y aparte de lo indicado en la propia imagen, de una CPU Intel Core m3-8100y, núcleo de 4 hilos, 8GB LPDDR3 de memoria RAM. Con esto queremos reflejar que si bien no es igual que un ordenador actual en cuanto a recursos y potencia, estamos hablando de un mini ordenador con suficientes recursos como para ejecutar modelos previamente entrenados.

3.2 Desarrollo del software

En el ámbito de la visión por computadora y la inteligencia artificial, la detección precisa de objetos es esencial para una amplia variedad de aplicaciones, desde la seguridad hasta la asistencia médica. En este contexto, el presente Trabajo de Fin de Grado (TFG) se centra en una solución innovadora desarrollada mediante el entrenamiento de una red neuronal convolucional conocida como YOLOv8 (You Only Look Once, versión 8) para la detección de la pupila.

La pupila, como parte fundamental de la anatomía del ojo humano, desempeña un papel crucial en diversos campos, incluyendo la oftalmología, la investigación del comportamiento visual y la interacción hombre-máquina. La capacidad de detectar y rastrear la pupila con precisión y eficiencia en tiempo real es esencial para comprender y abordar una serie de desafíos en estas áreas. Sin embargo, esta tarea no es trivial debido a la variabilidad en la apariencia y las condiciones de iluminación.

En este contexto, el desarrollo de un sistema de detección de pupila basado en YOLOv8 representa una contribución significativa. YOLOv8 es una arquitectura de red neuronal profunda que se ha destacado por su capacidad para lograr un equilibrio óptimo entre precisión y velocidad de detección. La elección de YOLOv8 como base para este proyecto se fundamenta en su eficiencia y su historial de éxito en la detección de objetos en tiempo real.

Esta sección del proyecto se podría dividir en 3 partes que encajan con el flujo de trabajo que hemos llevado a cabo: la adquisición de datos, gracias a los cuales hemos podido entrenar la red, el entrenamiento del modelo YOLOv8, y su optimización con TFLite, esto último de cara a poder ejecutar nuestro modelo en entornos que no dispongan de la misma cantidad y calidad de recursos de los que puede disponer un ordenador personal.

3.2.1 Adquisición de datos

Para poder tener éxito en cualquier proyecto que use la inteligencia artificial tenemos que saber que la adquisición de datos es un pilar fundamental. Debemos usar datos de calidad y en grandes cantidades, dado que si no poseemos una buena base de datos con estas características a nuestro modelo le será imposible obtener un entrenamiento apropiado y esto lo veremos en los resultados.

La calidad y cantidad de datos recopilados son factores críticos que influyen directamente en la capacidad de una red neuronal para aprender y generalizar patrones con precisión. En el caso de la detección de la pupila, la adquisición de datos desempeña un papel crucial, ya que estos datos actúan como el "combustible" que alimenta el proceso de entrenamiento de la red.

Una de las características que hacen a un banco de datos ser de calidad, es que cada uno de los archivos que se usen para entrenar la red neuronal sea representativo de lo que se quiere llegar a conseguir. Es decir, si tenemos que hacer un modelo cuyo objetivo es la detección de perros, debemos introducir fotografías en las que salgan perros.

Además dichos datos deben ser diversos. Para el ejemplo puesto antes, si solo entrenamos la red con fotos de perros de raza labrador, cuando en la fuente de datos que está examinando el modelo aparezca un chihuahua probablemente no lo catalogue como un perro puesto que no se parece lo suficiente a los datos que ha estado analizando la red durante su entrenamiento.

La calidad de los datos también es un factor crítico. Los datos ruidosos, inconsistentes o incompletos pueden dar lugar a un entrenamiento deficiente de la red y, en última instancia, a una detección poco fiable. Por lo tanto, el proceso de adquisición debe llevarse a cabo de manera meticulosa, garantizando la integridad y coherencia de los datos recopilados.

Todo esto convierte a la fase de adquisición de datos en un componente de nuestro proyecto crucial. En nuestro caso hemos encontrado un laboratorio que posee varias bases de datos que nos han servido para nuestro proyecto. Lo podemos encontrar en "*biometrics*". Las imágenes que hemos obtenido son todas en escala de grises y tienen una resolución de 640x480.

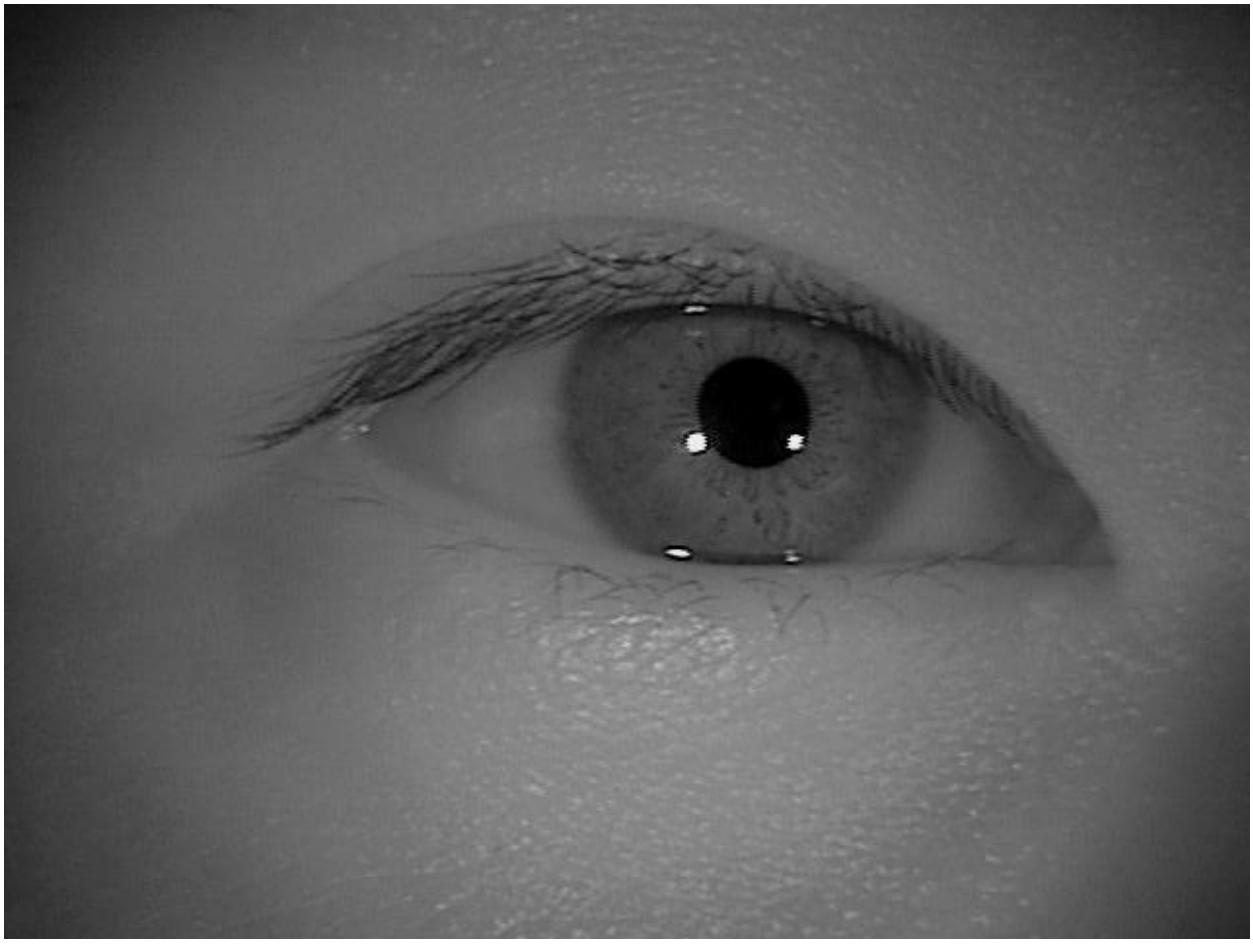

Ilustración 16: Imagen de ojo izquierdo sacada del banco de datos

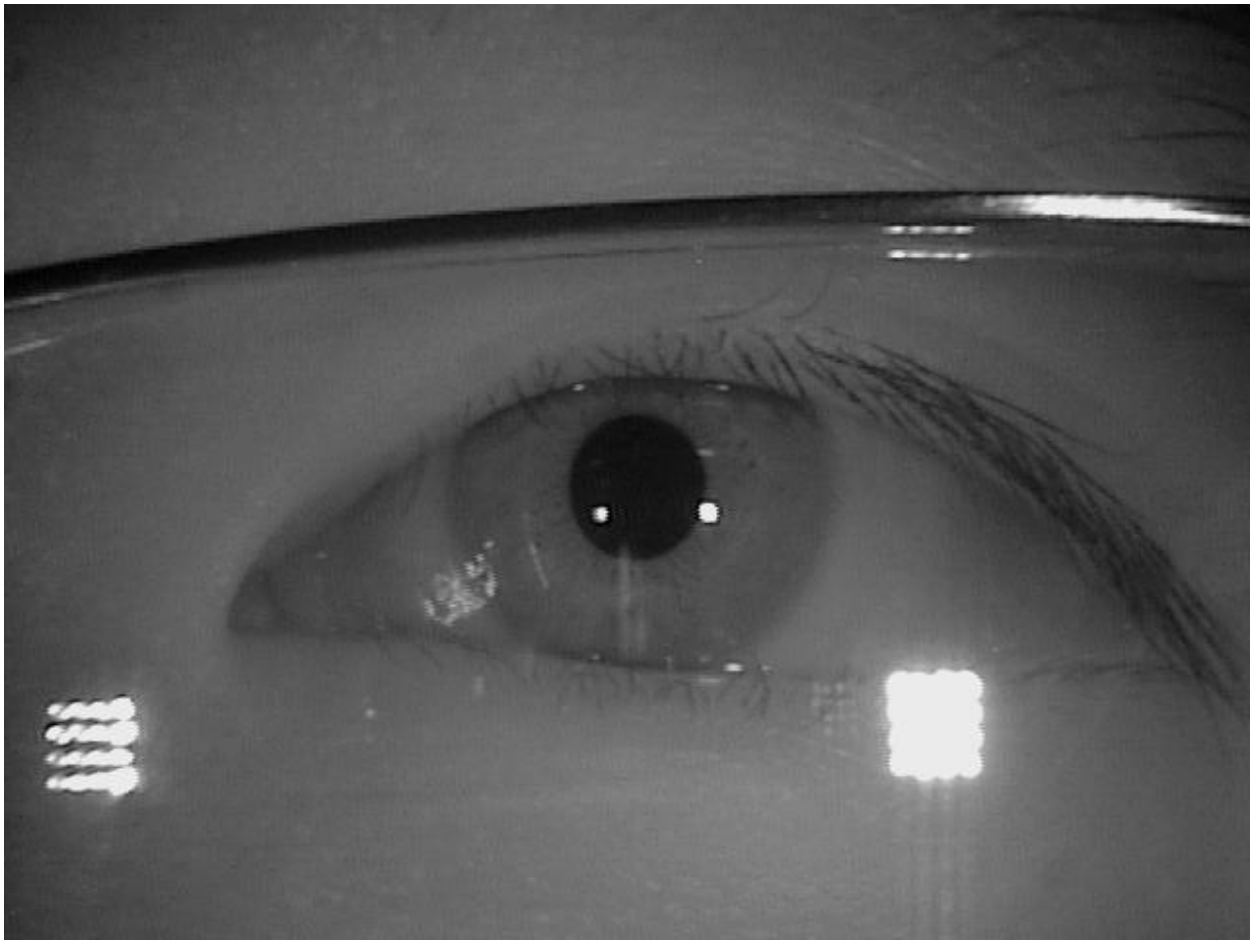

Ilustración 17: Imagen de un ojo derecho con gafas sacada del banco de datos.

En cuanto a diversidad los datos usados contienen ojos de 75 personas de diferentes etnias, además de usuarios de gafas, expuestos a un gran foco y a oscuridad para obtener pupilas de diferentes tamaños. Es por todo esto que hemos considerado que el banco de datos elegido es bastante consistente, variado y representativo.

3.2.2 Desarrollo del modelo YOLOv8

3.2.2.1 Tipo de entrenamiento del modelo

Una vez tenemos los datos con los que vamos a entrenar al modelo, debemos tener en consideración un aspecto muy importante antes de intentar implementar una herramienta que use inteligencia artificial y es el desarrollo desde 0 de una red neuronal o el uso de herramientas ya creadas que tienen como objetivo facilitar todo el proceso. En nuestro caso hemos elegido usar la tecnología de Ultralytics, ya que ofrecen una herramienta super útil basada en el algoritmo YOLO que nos facilita bastante esta sección del proyecto.

Este algoritmo ha demostrado una eficacia y eficiencia increíbles a la hora de detectar objetos, y concretamente su última versión arroja unos resultados estupendos y más que fiables. Es por todo esto que hemos elegido esta herramienta.

Lo primero que debemos abordar en este punto es el tipo de entrenamiento que necesitamos para

nuestro modelo. Para este caso, debemos usar entrenamiento supervisado.

Para ponernos en contexto, el entrenamiento supervisado es un método en el que un modelo de inteligencia artificial se entrena utilizando un conjunto de datos etiquetado. En otras palabras, se proporciona al modelo tanto las entradas como las salidas deseadas (etiquetas) durante el proceso de entrenamiento. El objetivo principal es que el modelo aprenda a mapear las entradas a las salidas correctas y, posteriormente, pueda realizar predicciones precisas sobre nuevos datos no vistos. Por ejemplo, en la clasificación de imágenes, se proporcionan imágenes junto con etiquetas que indican las categorías a las que pertenecen. El modelo se entrena para reconocer patrones en las imágenes y asignar la categoría correcta a nuevas imágenes.

En cambio, el entrenamiento no supervisado implica la ausencia de etiquetas en el conjunto de datos de entrenamiento. El modelo se expone únicamente a las entradas y debe encontrar patrones o estructuras en los datos por sí mismo. Esto puede incluir la identificación de grupos de datos similares (clustering) o la reducción de la dimensionalidad de los datos. Un ejemplo común es el clustering de usuarios en grupos según sus comportamientos en un sitio web, sin conocimiento previo de las categorías.

Primero intentamos un entrenamiento que mezclase ambos, procesando cada imagen a mediante el uso de filtros, detectando la pupila en cada de estas y generando un archivo asociado a cada una de ellas de tipo .txt, con las coordenadas de la geometría que rodea la pupila. Esto se encuentra en el archivo anexo ‘deteccion_pupila.py’. No obstante, lo desechamos después de varios intentos por falta de precisión y optamos por el entrenamiento supervisado.

Para ello hemos tenido que etiquetar cada una de las imágenes que hemos querido usar para el entrenamiento. Aquí hemos usado una herramienta llamada “labelme”. Esta herramienta ofrece una interfaz con las imágenes de un directorio y te da la posibilidad de realizar un etiquetado de la zona que deseamos, dándole entidad al cuerpo que queremos que nuestro modelo aprenda (en nuestro caso la pupila).

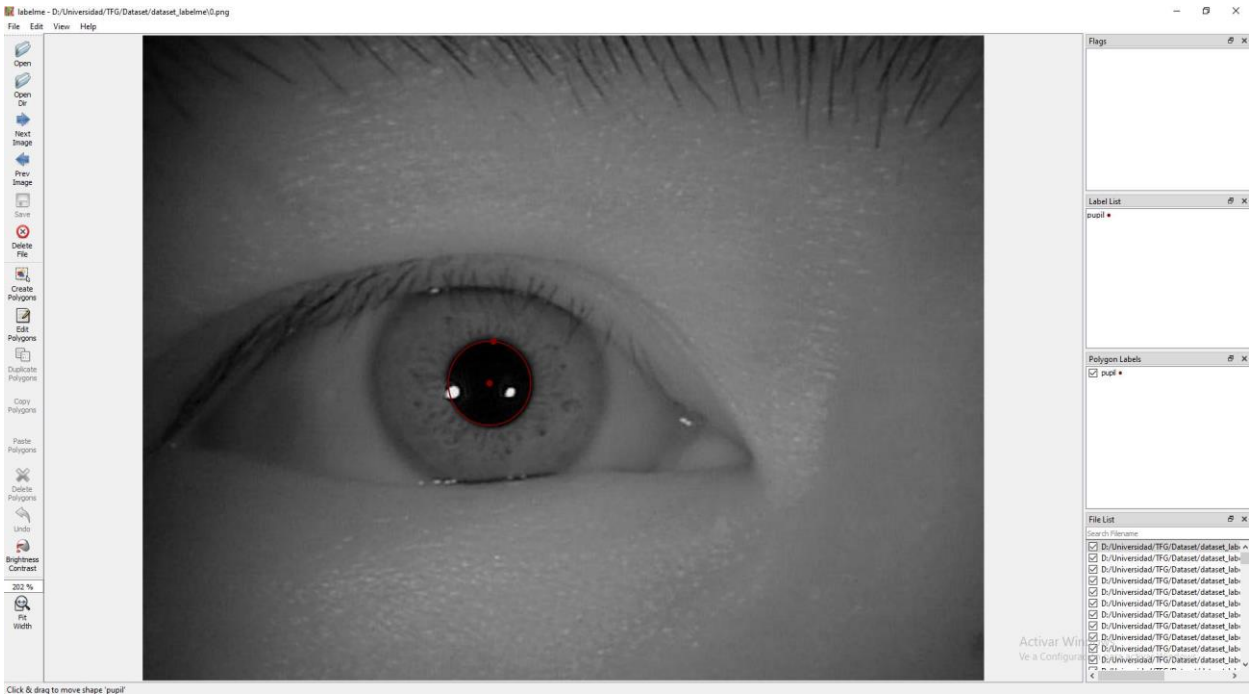

Ilustración 18: Etiquetado de imágenes con labelme.

En la imagen podemos ver uno de los ojos que hemos escogido para el etiquetado, con un

marcado en la pupila visible por la línea roja y su centro. Este marcado genera un archivo .json que contiene todos los datos que ha sacado la herramienta de esta imagen, y te lo genera con el mismo nombre. Este archivo contiene las coordenadas dentro de la imagen, entre otras cosas.

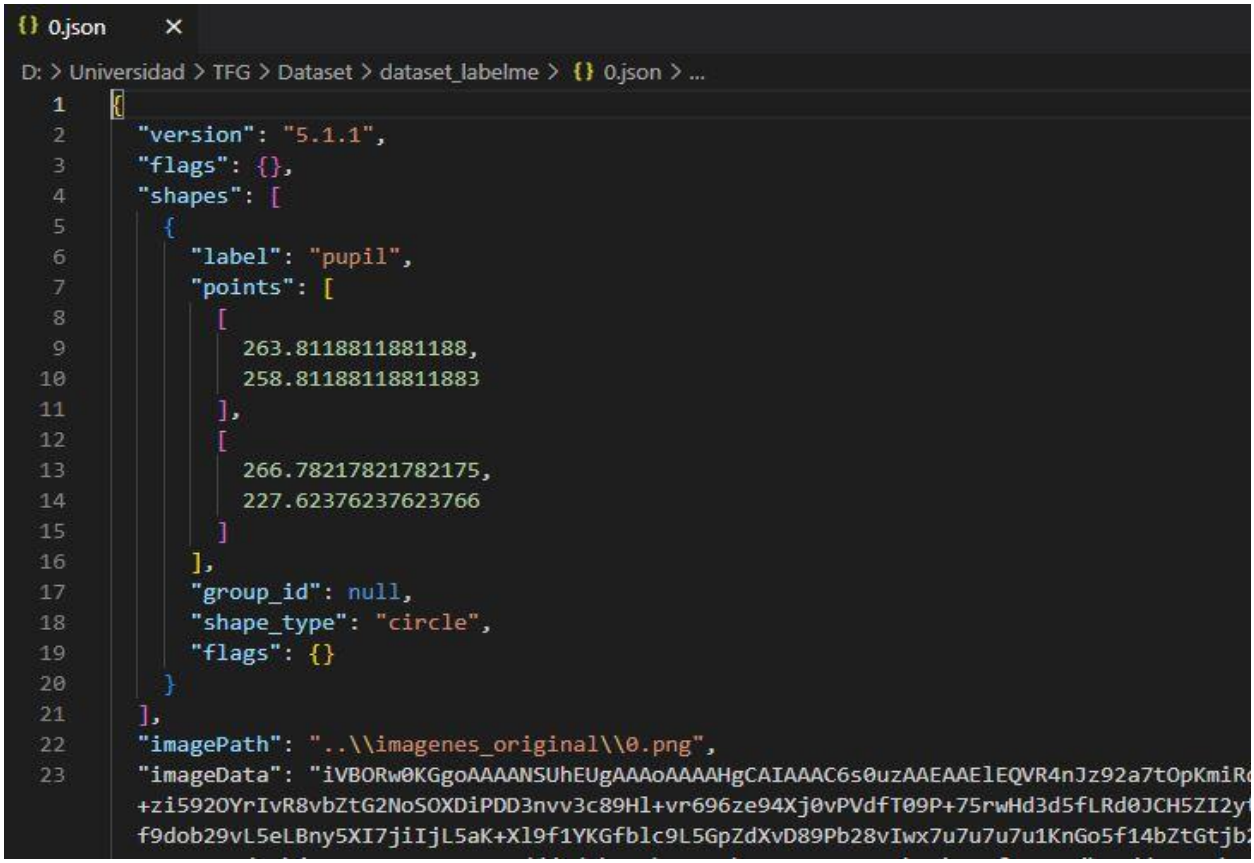

Ilustración 19: Datos generados por labelme.

Una vez que tengamos el etiquetado de forma supervisada de cada una de las imágenes de nuestro conjunto de datos, lo que nos va a quedar es una imagen, nombrada con un número siguiendo el formato 000, 001, etc. Además, asociada a cada una de estas imágenes tendremos el archivo correspondiente al de la imagen, con sus datos y nombrados siguiendo la misma serie.

Una vez que tenemos esto, debemos pasar los datos creados por la herramienta al tipo de datos que puede interpretar nuestro modelo de YOLOv8. Básicamente tenemos que convertir los archivos .json en archivos .txt que contienen las coordenadas guardadas en los archivos .json. Esto lo hemos hecho usando una herramienta que hemos descargado desde *https://github.com/rooneysh/Labelme2YOLO* llamada Labelme2YOLO. Tras ejecutarla, obtenemos todos los archivos .txt que necesitamos con las coordenadas de cada una de las imágenes. Tras esto, lo tenemos todo para empezar a entrenar nuestro modelo.

Para la organización de directorios y el almacenamiento de todo lo que necesitamos para entrenar la red, hemos automatizado un proceso por el cual leemos los datos, los renombramos, cambiamos su extensión y los almacenamos en el lugar y la forma que necesitamos, todo esto usando un script llamado “CarpetaInicial.py” que adjuntamos en el anexo.

3.2.2.2 Características del modelo

En este apartado vamos a centrarnos en la propia tecnología desarrollada por Ultralytics, contando cómo es, cómo funciona y lo que nosotros hemos hecho para usarla una vez hemos conseguido todo lo descrito hasta ahora.

La herramienta desarrollada por Ultralytics conocida como YOLO fue creada por Joseph Redmon y Santosh Divvala en 2016. Desde entonces, han ido sacando nuevas versiones que han mejorado a la anterior tanto en rapidez y precisión como en eficiencia. Adía de hoy, ya van por la versión YOLOv8, la mejor que han conseguido hasta la fecha.

Estas versiones son una implementación de alta eficiencia de la popular arquitectura YOLO (You Only Look Once), arquitectura de la que ya hemos hablado en el estado del arte de este proyecto. Con YOLOv8, Ultralytics ha llevado la detección de objetos a un nuevo nivel dentro de las soluciones en este campo llevadas a cabo mediante inteligencia artificial, consiguiendo un equilibrio excepcional entre velocidad y precisión.

Es por esta razón que la tecnología de Ultralytics ha ganado popularidad en este campo durante los últimos años. El desarrollo de este modelo se basó en 3 pilares clave:

- **Mejora en la eficiencia**: Se realizaron optimizaciones significativas para garantizar que conforme iban avanzando las versiones, estas ofrecieran una velocidad de detección aún más rápida sin comprometer la precisión. Esto lo convierte en una opción atractiva para aplicaciones en tiempo real y en dispositivos con recursos limitados.
- **Precisión mejorada**: La arquitectura de red y los métodos de entrenamiento se afinaron con cada nueva versión para lograr una mayor precisión en la detección de objetos. Esto significa que cada nueva versión es capaz de detectar y clasificar objetos con una precisión mayor que su predecesora.
- **Facilidad de uso**: Ultralytics se centró en hacer que YOLO sea más accesible para los desarrolladores, proporcionando una implementación sencilla y una documentación detallada. Esto ha fomentado su adopción en la comunidad de desarrollo.

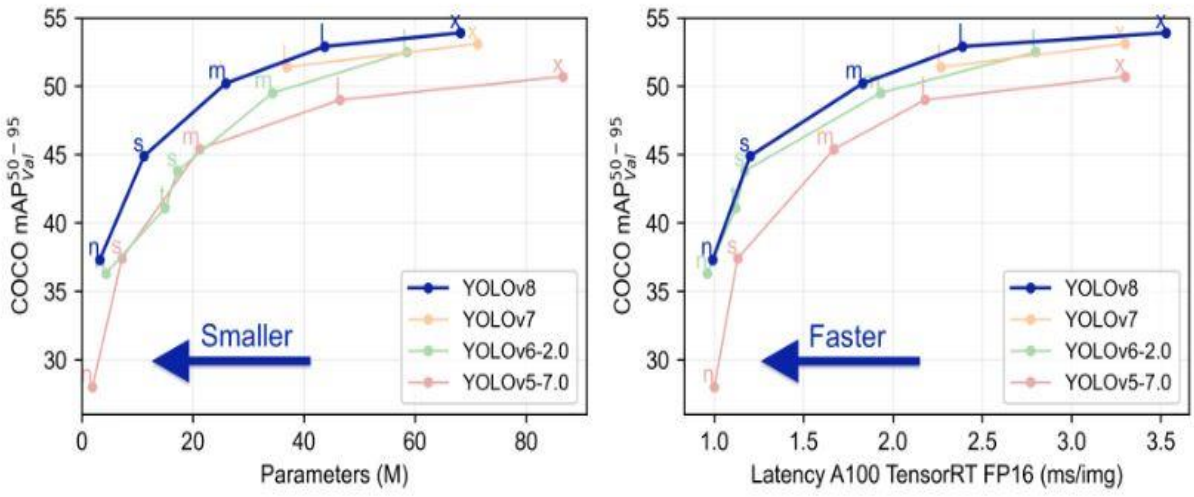

Ilustración 20: Comparativa en tamaño y rapidez de las diferentes versiones de YOLO. Imagen de: *https://github.com/ultralytics/ultralytics/tree/main*

La imagen nos muestra un par de gráficas en las que se comparan los diferentes modelos preentrenados de cada una de las versiones que han desarrollado de YOLO, tanto en tamaño como en rapidez.

A simple vista se pueden observar dos conclusiones bastante importantes; una es que los modelos han mejorado su rendimiento a pesar de mantener el mismo número de parámetros (su tamaño, algo de vital importancia para la posterior ejecución del modelo), y dos es que la mejora de rendimiento no se ha traducido en una pérdida de rapidez, algo que es sumamente importante, sobretodo para las aplicaciones que tienen estos modelos en visión por computador.

En cuanto a nuestro modelo, hemos usado los modelos preentrenados pequeño (modelo n) y mediano (modelo m), ya que consideramos que poseen las características que son necesarias para llevar a cabo el objetivo de este proyecto, tanto en consumo de recursos como en rapidez.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Ilustración 21: Tabla con los atributos de cada tipo de modelo.Imagen de: *https://github.com/ultralytics/ultralytics/tree/main*

En la imagen anterior podemos ver cada uno de los atributos más importantes de cada uno de los modelos que han sido desarrollados dentro del entorno YOLOv8:

- **Tamaño de Entrada (Pixels):** Los modelos YOLOv8 están diseñados para adaptarse a diferentes tamaños de entrada, lo que los hace versátiles para una variedad de aplicaciones. YOLOv8n, el modelo más pequeño, utiliza imágenes de 640 píxeles de ancho y alto, mientras que YOLOv8x, el modelo más grande, opera con la misma resolución de 640 píxeles. La elección del tamaño de entrada depende de la necesidad de precisión y velocidad.
- **Puntaje mAPval:** La precisión es un factor crucial en las aplicaciones de detección de objetos. Los modelos YOLOv8 ofrecen diferentes niveles de precisión, desde el 37.3% de YOLOv8n hasta el 53.9% de YOLOv8x. Esto significa que YOLOv8x tiene una mayor capacidad para reconocer objetos en comparación con YOLOv8n, que se enfoca en aplicaciones donde la precisión es menos crítica.
- **Velocidad de Inferencia:** La velocidad de inferencia es esencial para aplicaciones en tiempo real. YOLOv8n es el modelo más rápido, con una velocidad de inferencia de 0.99

ms en CPU ONNX y 3.2 ms en A100 TensorRT. Por otro lado, YOLOv8x es más lento debido a su mayor tamaño y complejidad, con tiempos de 3.53 ms y 68.2 ms, respectivamente. La elección del modelo debe equilibrar la velocidad y la precisión.

- **Número de Parámetros y FLOPs:** Los modelos más grandes, como YOLOv8x, tienen un mayor número de parámetros (3.53 millones) y realizan más operaciones de punto flotante por segundo (68.2 mil millones). Estos modelos son adecuados para aplicaciones donde se necesita la máxima precisión pero se dispone de suficiente capacidad de procesamiento.

En resumen, los modelos YOLOv8 ofrecen una gama de opciones para satisfacer diversas necesidades. Ultralytics pone a disposición de los desarrolladores desde YOLOv8n, que prioriza la velocidad, hasta YOLOv8x, que busca la máxima precisión. Nosotros hemos priorizado un poco la velocidad por encima de la precisión, dado que se trata de un video en *streaming* y esto de por sí ya requiere un gran consumo de recursos, y por eso hemos escogido el modelo m.

Es importante resaltar que nos vamos a centrar en los modelos preentrenados que han sido desarrollados para la detección de objetos, dado que Ultralytics también ha desarrollado modelos para la resolución de los problemas de segmentación y clasificación de objetos, así como diferentes modelos que estiman la posición de puntos clave a partir de ciertas referencias (*keypoints*).

En cuanto a las claves de YOLOv8 que la hacen mejor respecto a sus versiones anteriores:

1. **Arquitecturas avanzadas de Backbone y Neck:** YOLOv8 utiliza arquitecturas de Backbone y Neck de última generación para una extracción de características mejorada. Esto se traduce en un aumento en la precisión de la detección y una mejora en el rendimiento general del modelo.

2. **Cabeza Ultralytics sin anclajes**: YOLOv8 adopta un cabezal Ultralytics dividido sin anclajes, es decir, sin cajas predefinidas a la hora de detectar objetos. Esto contribuye a una mayor precisión y a un proceso de detección más eficaz en comparación con los enfoques basados en anclaje.

3. **Óptimo equilibrio entre precisión y velocidad:** YOLOv8 ha sido diseñado con un enfoque en mantener un equilibrio óptimo entre precisión y velocidad. Esto lo hace adecuado para tareas de detección en tiempo real en diversas áreas de aplicación, donde cada milisegundo cuenta.

4. **Variedad de modelos preentrenados:** YOLOv8 ofrece una amplia gama de modelos preentrenados para satisfacer diversas necesidades de rendimiento y tareas específicas. Esto facilita la selección del modelo adecuado para casos de uso particulares, acelerando la implementación.

Estas características hacen que YOLOv8 sea una opción atractiva para aplicaciones de detección de objetos en tiempo real en una variedad de dominios, desde seguridad y vigilancia hasta vehículos autónomos y análisis de video. Su enfoque en el equilibrio entre precisión y velocidad, junto con sus modelos preentrenados, lo convierten en una herramienta versátil y poderosa para resolver desafíos de visión por computador.

3.2.2.3 Entrenamiento del modelo

Para poder entrenar el modelo, una vez obtenidos los datos, hemos tenido que preparar el entorno de entrenamiento. Hemos usado Anaconda para esto, instalando dicho software en los dos terminales en los que vamos a ejecutar la red, mi ordenador portátil y la LattePanda.

Una vez listo, hemos creado un nuevo entorno llamado "yolov8" en el que hemos instalado las dependencias que vamos a necesitar más adelante por medio de la consola.Para usarla tenemos que irnos a sección "Home" en el menú de la izquierda y abrir CMD.exe Prompt.

Luego hemos procedido a la ejecución del comando 'pip install ultralytics'. Gracias a esto tenemos todas las librerías necesarias para entrenar y ejecutar los modelos que hemos usado.

Hecho esto, vemos que la versión de Python que poseemos sea la 3.10.9 o superior. Si es así, debemos comprobar que tenemos "torch.cuda" disponible, es decir, que podemos usar la plataforma CUDA.

CUDA es una plataforma de cómputo paralelo desarrollada por NVIDIA que permite aprovechar el poder de las unidades de procesamiento gráfico (GPU) para realizar cálculos de manera altamente eficiente. En el contexto de PyTorch y otros marcos de trabajo de aprendizaje profundo, CUDA se utiliza para acelerar el entrenamiento y la inferencia de modelos de redes neuronales en hardware compatible con GPU.

Cuando se ejecuta PyTorch en un sistema con una GPU NVIDIA compatible, CUDA se utiliza automáticamente para realizar operaciones matriciales y cálculos en paralelo, lo que acelera significativamente el rendimiento del entrenamiento y la inferencia de modelos de redes neuronales. Algunas de las operaciones más intensivas en cálculos, como la multiplicación de matrices y las operaciones de convolución, se ejecutan considerablemente más rápido en una GPU que en una CPU convencional.

Pero volvamos a nuestra consola. Esto lo haremos ejecutando en la misma consola los comandos:

(yolov8) C:\Users\Fran>python

>>> import torch

>>> torch.cuda.is_available()

En caso de que la respuesta al último comando sea 'True', es que lo tenemos instalado y operativo. Si no es así, debemos ejecutar el comando "pip3 install –upgrade torch torchvision torchaudio –extra-index-url https://download.pytorch.org/whl/cu117". Esto nos instalará automáticamente todo lo que necesitamos para poder usar CUDA.

Una vez tenemos esto, hemos conseguido todo lo necesario para poder poner a entrenar nuestro modelo con la base de datos que hemos conseguido. Para ello, debemos configurar los directorios de entrenamiento y validación de la red, así como el archivo 'dataset.yaml' que ha generado el software labelme con dichos directorios y comprobar que este posee sólo la etiqueta con la entidad que hemos creado, en nuestro caso "pupil".

Para los directorios, debe haber 2, uno para entrenamiento con el nombre 'train' y otro para la validación de la red con el nombre 'tests'. Dentro de cada uno de estos debe haber también 2

carpetas, una llamada ‘images’ con las imágenes y otra llamada ‘labels’ con los archivos .txt que contienen las coordenadas de las etiquetas que creamos en cada una de las imágenes. Recomendamos el uso del 90 por ciento del total de las imágenes para el entrenamiento de la red y un 10 por ciento para su validación.

Una vez tengamos listo, ponemos a entrenar el modelo. Para ello, ejecutamos el comando “yolo task=detect mode=train epochs=100 data=dataset.yaml model=yolov8m.pt imgsz=640 batch=4”. El comando tiene los siguientes elementos:

- **yolo**: Este es el comando principal que indica que se va a realizar una tarea relacionada con YOLO (You Only Look Once). En este caso, se trata de la detección de objetos.
- **task=detect**: Este parámetro especifica la tarea que se va a llevar a cabo, que es la detección de objetos en imágenes.
- **mode=train**: Indica que el modelo se encuentra en modo de entrenamiento. Durante el entrenamiento, el modelo se ajusta a los datos del conjunto de entrenamiento para aprender a detectar objetos.
- **epochs=100**: Este parámetro establece la cantidad de épocas de entrenamiento. Una época se refiere a una pasada completa a través de todo el conjunto de datos de entrenamiento. En este caso, se realizarán 100 épocas de entrenamiento. Decidimos que era un buen número de épocas ya que cuando probamos con 200, el entrenamiento paró en 177, arrojándonos que no se encontraron mejoras en las últimas 50 épocas.
- **data=dataset.yaml**: Especifica el archivo de configuración del conjunto de datos que se utilizará para el entrenamiento. Este archivo YAML contiene información sobre las rutas de los datos, las clases de objetos, el tamaño de las imágenes y otros detalles del conjunto de datos personalizado. Este es el archivo que hemos tenido que modificar previamente para añadir las rutas de nuestros directorios con los datos.
- **model=yolov8m.pt**: Indica el modelo preentrenado que se utilizará como punto de partida para el entrenamiento. En este caso, se utiliza el modelo YOLOv8m como base para el entrenamiento.
- **imgsz=640**: Este parámetro establece el tamaño de las imágenes de entrada durante el entrenamiento. Las imágenes se redimensionarán a un tamaño de 640x640 píxeles antes de ingresar al modelo, dado que originalmente poseen un tamaño de 640x480 en nuestro caso.
- **batch=4**: Indica el tamaño del lote (batch size) que se utilizará durante el entrenamiento. El tamaño del lote se refiere al número de ejemplos de entrenamiento que se procesarán simultáneamente en cada paso de entrenamiento. En este caso, se utiliza un tamaño de lote de 4. Intentamos primero con 16 y, dado que mi ordenador no pudo soportarlo por falta de memoria, lo volvimos a intentar con 8, pero tampoco resultó efectivo. Al final lo dejamos en 4 porque fue el mayor tamaño de lote que nos permitió antes de arrojarnos el error por falta de memoria.

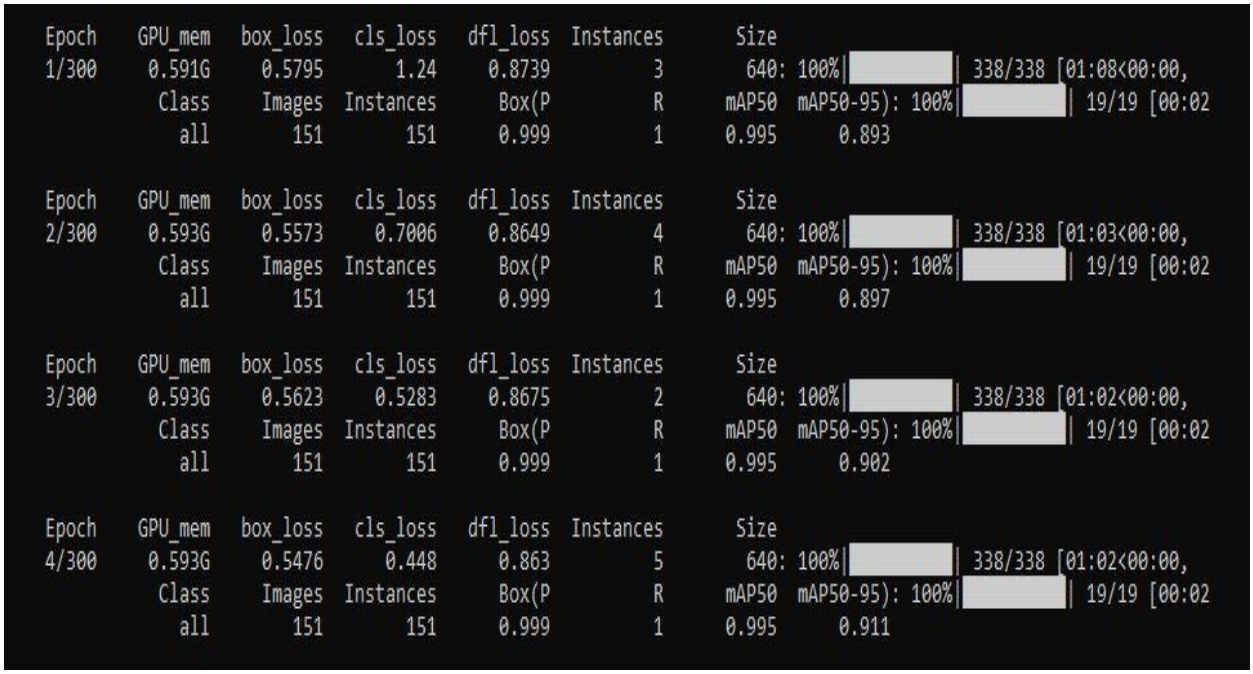

Ilustración 22: Inicio del entrenamiento del modelo

Tras unas 10 horas entrenando con nuestro banco de datos etiquetado de forma supervisada, el entrenamiento finaliza proporcionando 2 modelos en forma de archivos '.pt'. Estos modelos son "best.pt" y "last.pt", aunque para nosotros solo será importante el primero.

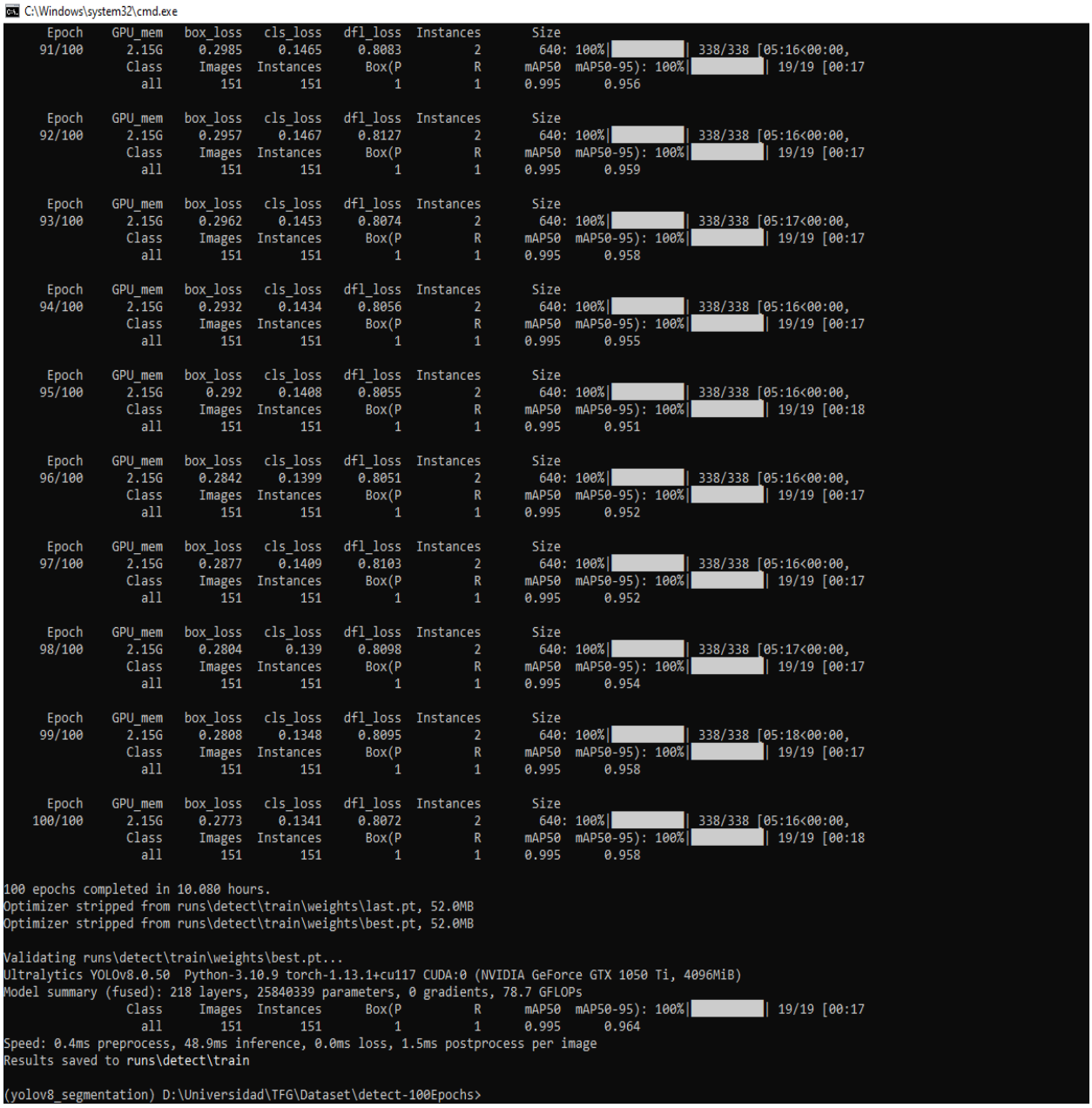

Ilustración 23: Finalización del entrenamiento del modelo

En la imagen podemos observar el final del entrenamiento del modelo, el número de épocas, el tamaño de los 2 modelos resultantes (es el mismo), los resultados de la validación, las pérdidas o el tiempo medio de inferencia entre otras cosas. No obstante, de los resultados hablaremos más adelante.

Como conclusión podemos decir que el entrenamiento del modelo se ha completado en un tiempo de 10.08 horas, más que decente teniendo en cuenta la tarjeta gráfica con la que los hemos entrenado (la podemos ver en la imagen, una GTX 1050). Además la velocidad que nos ha proporcionado es buena, junto con un modelo de 52 MB bastante compacto. Teniendo en cuenta todo lo mencionado, podemos decir que el entrenamiento de nuestro modelo se ha completado con éxito.

3.2.3 Optimización de YOLOv8 con ONNX y TFLite

Las modelos de Ultralytics tienen soluciones para la optimización de cara a la implementación de los modelos una vez entrenados en diferentes plataformas y dispositivos. Esto lo han conseguido gracias al traspaso de los formatos de los modelos que generamos a partir de los preentrenados, concretamente a los formatos '.onnx' y 'tflite'. Gracias a esto han conseguido que ejecutar sus modelos en dispositivos que poseen menos recursos que un ordenador convencional de última generación sea posible.

Al transformar un modelo YOLO entrenado por Ultralytics al formato .onnx (Open Neural Network Exchange), se desencadenan una serie de procesos clave que mejoran significativamente la versatilidad y la eficiencia del modelo. En primer lugar, Ultralytics se encarga de exportar todas las capas y componentes del modelo YOLO a un formato compatible con ONNX, lo que incluye las capas de detección, la arquitectura de la red neuronal y los parámetros de peso y sesgo. Este proceso implica la extracción, mapeo y conversión de capas, pesos y configuración del modelo YOLOv8 al .onnx.

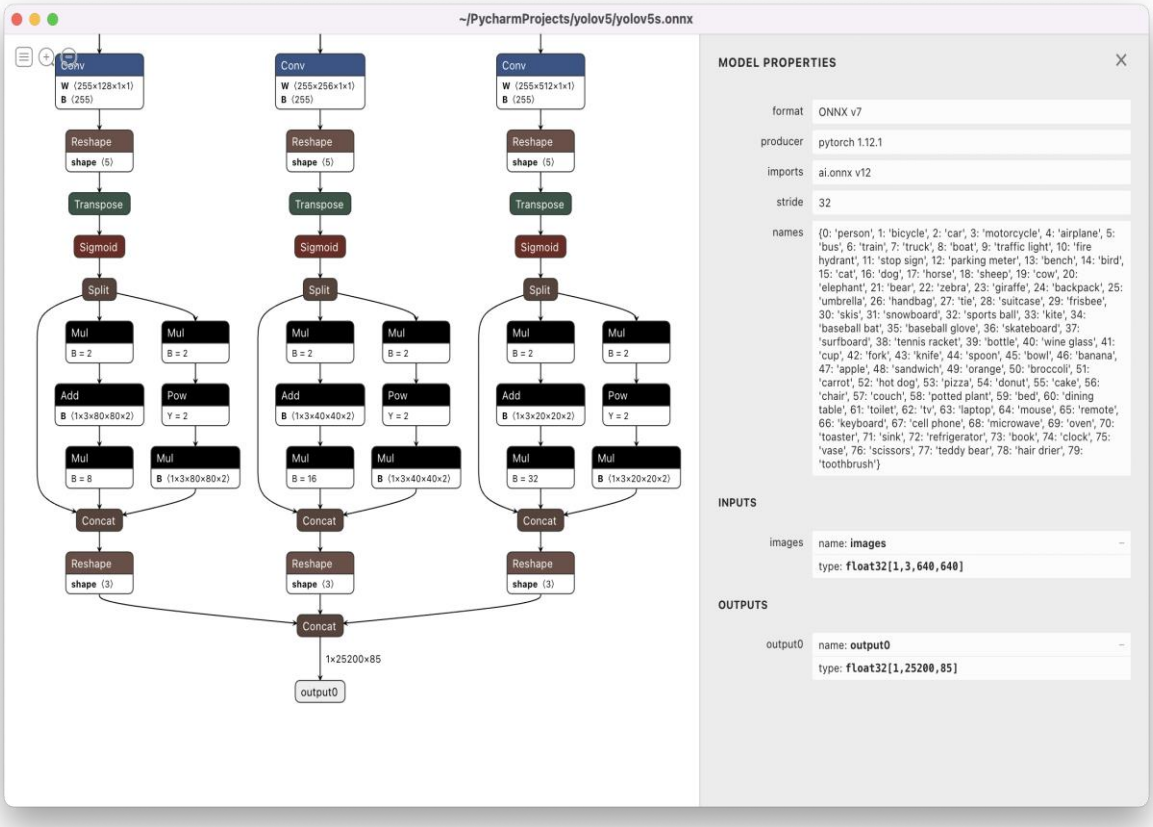

Ilustración 24: Esquema de la optimización entrada-salida al modelo .onnx. Imagen de: https://docs.ultralytics.com/yolov5/tutorials/model_export/?query=tflite#export-a-trained-yolov5-model

Algo muy similar ocurre con la exportación de nuestro modelo .pt al modelo .tflite. La conversión a .tflite permite que el modelo sea altamente eficiente y adecuado para su ejecución en dispositivos con recursos limitados, como dispositivos móviles, sistemas embebidos y

aplicaciones de IoT (Internet de las cosas). Esto es posible debido a la optimización específica de TensorFlow Lite para la inferencia en hardware de bajo consumo energético.

Además, .tflite es un formato estándar compatible con TensorFlow, lo que facilita la integración y el despliegue en aplicaciones de TensorFlow existentes. Esto permite a los desarrolladores implementar modelos de detección de objetos en sus proyectos de manera más sencilla y eficiente.

Cuando la exportación está completa, el modelo resultante puede ser implementado en multitud de entornos diferentes y de forma eficiente, desde móviles hasta sistemas embebidos. Para poder realizar esta exportación, lo único que debemos hacer es ejecutar el script anexo “import.py”.

4 RESULTADOS

"La grandeza de un hombre se mide por la medida en que deja huella positiva en la vida de los demás"

-Sam Houston-

En esta sección se analizan los resultados obtenidos del entrenamiento y de la cuantización de la red neuronal, así como la comparación de su ejecución en distintos entornos. Vamos a ir a través del modelo principal que hemos obtenido del entrenamiento a partir del modelo preentrenado de Ultralytics hasta las exportaciones realizadas a los formatos .onnx y .tflite. Por último, se presentan las conclusiones del trabajo realizado.

4.1 Resultados del modelo .pt

Antes de ejecutar el modelo en cualquierade los 2 entornos de los que disponemos, ordenador portátil y LattePanda, deberíamos analizar los resultados arrojados por el propio proceso de Ultralytics.

Tras entrenar el modelo, el proceso ejecutado para entrenar la red nos arroja una serie de resultados dentro del directorio en el que lo hemos entrenado. Esta carpeta lleva la ruta "…/runs/detect/train" y ofrece gráficas de los resultados, datos en formato .csv ejemplos del análisis que ha hecho el propio modelo durante la validación.

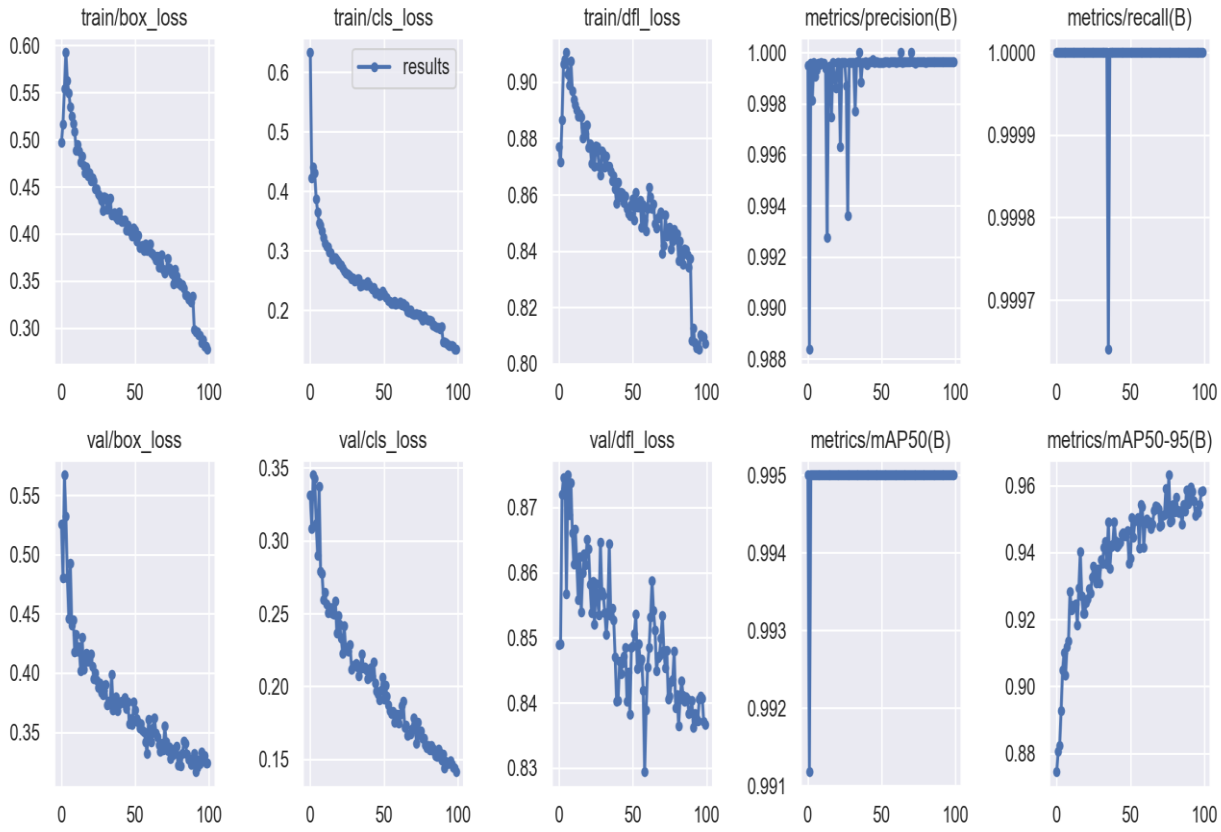

Ilustración 25: Gráficas de resultados de nuestro modelo entrenado

Estas gráficas son:

- **train/box_loss**: Esta gráfica muestra la pérdida (loss) asociada a la regresión de las coordenadas de las cajas delimitadoras (bounding boxes) durante el entrenamiento. Indica cuán bien el modelo está prediciendo las ubicaciones de los objetos en la imagen. Una disminución en esta pérdida significa que el modelo está mejorando en la precisión de la ubicación de las cajas delimitadoras.

- **train/cls_loss**: Representa la pérdida relacionada con la clasificación de objetos en las cajas delimitadoras durante el entrenamiento. Muestra cuán bien el modelo está prediciendo las clases de los objetos detectados. Una disminución en esta pérdida indica una mejora en la precisión de la clasificación.

- **train/dfl_loss**: Esta gráfica muestra la pérdida asociada a la detección de objetos en el frente y fondo (foreground-background). Ayuda a evaluar cómo el modelo discrimina entre objetos reales y falsos positivos durante el entrenamiento.

- **metrics/precision(B):** La precisión (precision) es una métrica que evalúa cuántas de las detecciones positivas realizadas por el modelo son realmente correctas. La "B" aquí se refiere a la métrica calculada en el conjunto de entrenamiento. Un valor alto de precisión indica que el modelo está haciendo menos falsas alarmas.

- **metrics/recall(B):** La recuperación (recall) es una métrica que evalúa cuántos de los objetos reales se detectaron correctamente. La "B" denota que esta métrica se calcula en

el conjunto de entrenamiento. Un valor alto de recuperación indica que el modelo está capturando la mayoría de los objetos.

- **val/box_loss**: Similar a "train/box_loss", esta gráfica muestra la pérdida de regresión de cajas delimitadoras, pero se calcula en el conjunto de validación en lugar del conjunto de entrenamiento. Ayuda a evaluar el rendimiento del modelo en datos no vistos durante el entrenamiento.
- **val/cls_loss**: Similar a "train/cls_loss", muestra la pérdida de clasificación, pero en el conjunto de validación. Evalúa la precisión de la clasificación en datos de validación.
- **val/dfl_loss**: Similar a "train/dfl_loss", representa la pérdida de detección de frente y fondo en el conjunto de validación.
- **metrics/mAP50(B):** El promedio de precisión en el margen del 50% (mean Average Precision at 50%) es una métrica que evalúa la precisión de la detección en un margen de coincidencia del 50%. Cuanto mayor sea este valor, mejor es la capacidad del modelo para detectar objetos.
- **metrics/mAP50-95(B):** Es similar a "metrics/mAP50(B)", pero calcula el promedio de precisión en un margen de coincidencia que va desde el 50% hasta el 95%. Mide la capacidad del modelo para detectar objetos en diferentes niveles de solapamiento con las cajas delimitadoras verdaderas.

Estas gráficas y métricas son esenciales para evaluar el rendimiento y la capacidad de detección de un modelo YOLO durante el proceso de entrenamiento y validación. Ayudan a identificar áreas de mejora y a comprender cómo se desempeña el modelo en diferentes aspectos de la detección de objetos.

Podemos observar en las gráficas cómo todas tienden a valores de rendimiento mejores conforme vamos aumentando de épocas, llegando a límites bastante buenos para nuestro entrenamiento.

Además podemos ver cómo en la validación consigue acertar la pupila de forma exacta en imágenes estáticas para diferentes ojos, indicativo de que el entrenamiento ha ido por buen camino.

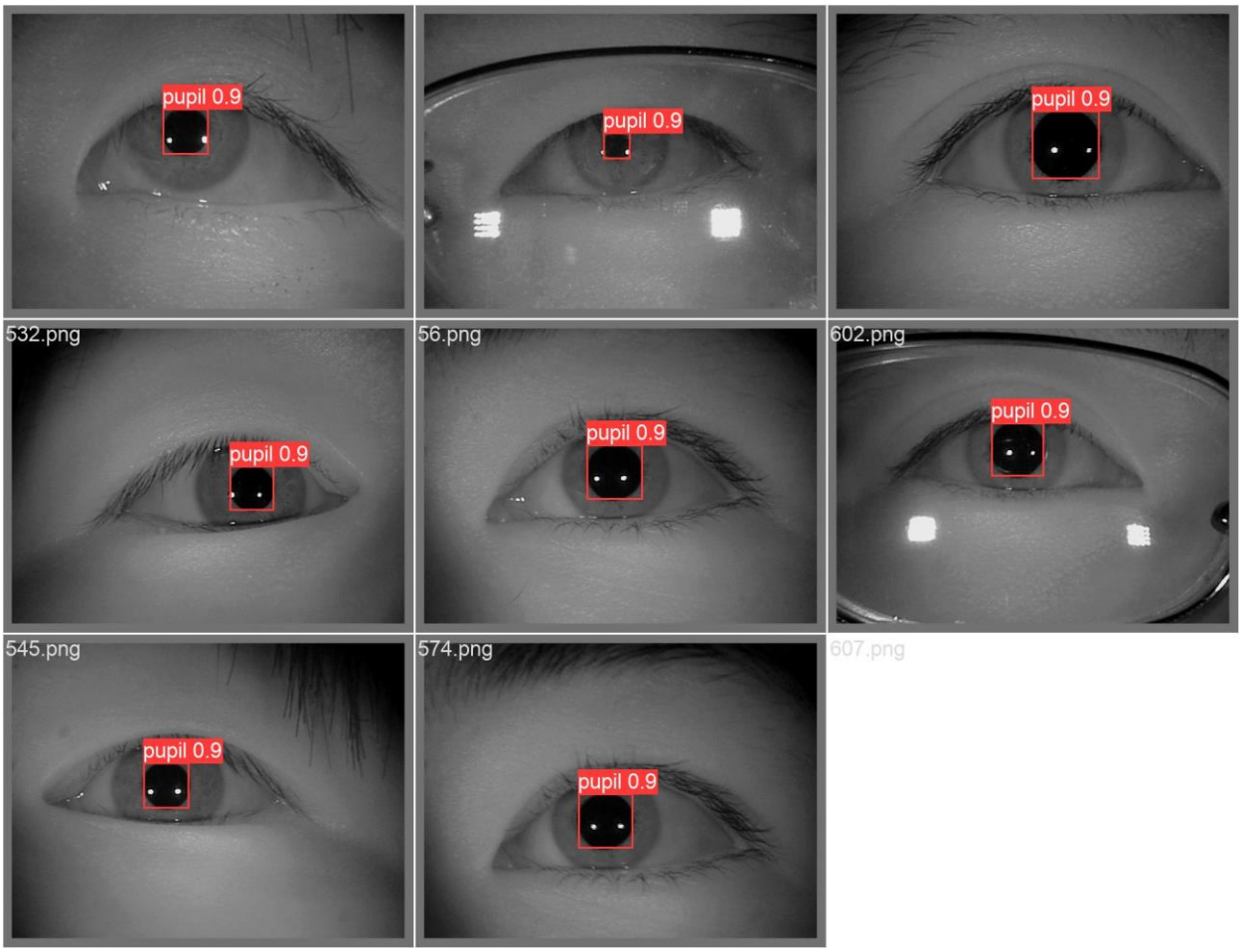

Ilustración 26: Detección de la pupila en imágenes durante la validación

Todo esto nos hace pensar que nuestro modelo ha conseguido un muy buen desempeño para la tarea en la que ha sido entrenado. No obstante, no adelantemos acontecimientos y veamos cómo se desenvuelve para el propósito que vamos buscando, la detección de la pupila en tiempo real.

4.2 Ejecución del modelo en diferentes entornos

Para hablar sobre los resultados del proyecto, lo primero que debemos hacer es ejecutar nuestro modelo en los diferentes entornos en los que nos hemos propuesto hacerlo. Como hemos mencionado antes, disponemos de 2 entornos diferenciados en los que queremos hacer funcionar nuestro dispositivo, uno en nuestro ordenador portátil, con el objetivo de ver cómo funciona nuestro modelo con abundancia de recursos, y otro en una LattePanda, con el objetivo de ver cómo se desenvuelve nuestro modelo en un entorno con menos recursos pero que podríamos llegar a incorporar en un sistema portable para las personas junto a un pupil lab.

La ejecución se ha llevado a cabo en ambos casos mediante el script que se llama “run_color.py” y que hemos adjuntado en el anexo. Este script no solo se encarga de ejecutar el modelo que le proporcionemos, sino que accede a la dirección de memoria asignada a la cámara de nuestro dispositivo mediante la función “cv2.VideoCapture” y además pone un identificador ID al objeto identificado y dibuja la trayectoria de su movimiento.

4.2.1 Ejecución en ordenador personal

Hemos ejecutado en cada entorno 3 versiones del mismo modelo: '.pt', '.onnx' y '.tflite'. En la ejecución de la primera hemos obtenido tiempos de inferencia muy buenos, en torno a los 45 ms y obtenemos una muy buena precisión. Aquí podemos observar una captura del video grabado a mi propio ojo de la ejecución del modelo:

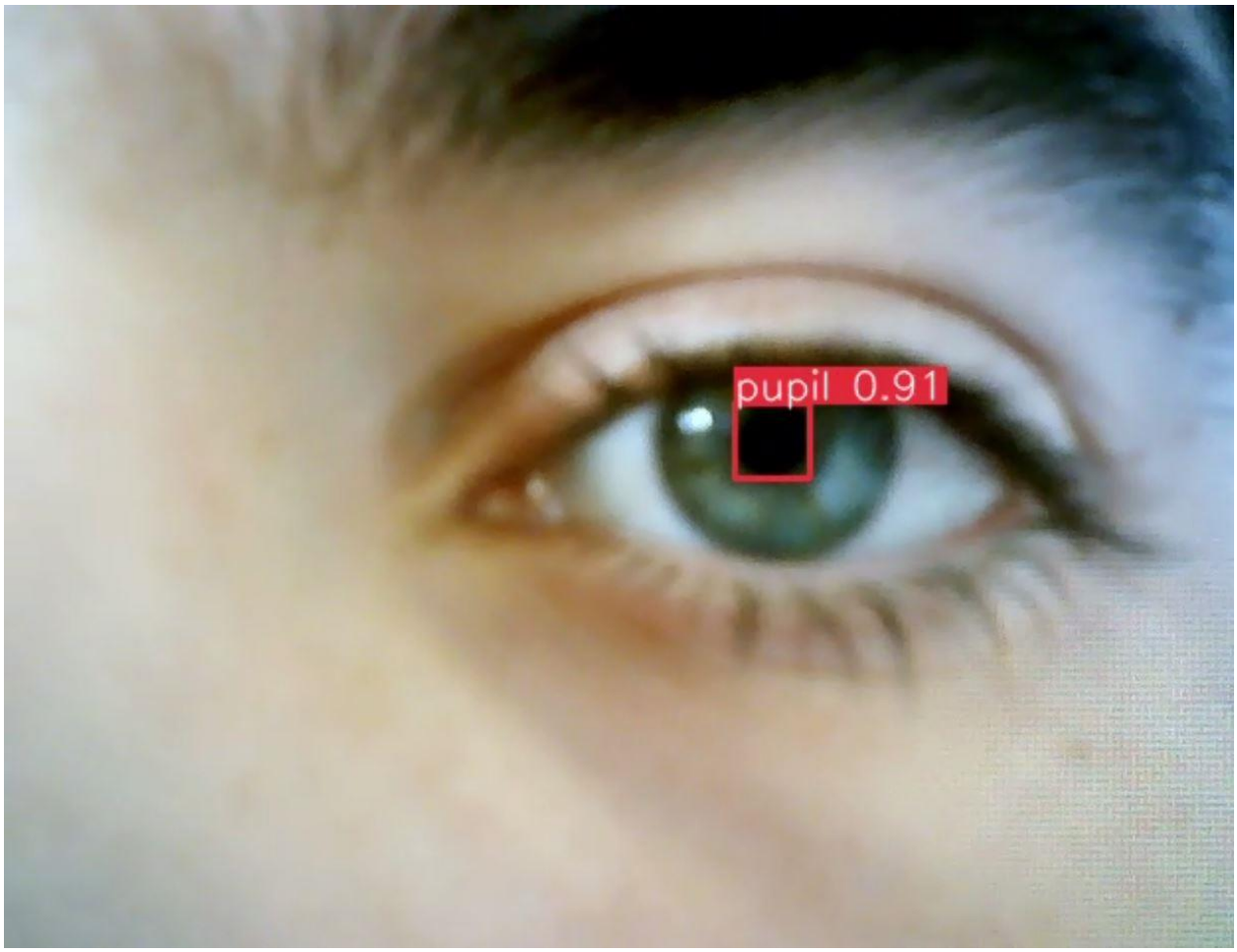

Ilustración 27: Detección de la pupila en pc del modelo .pt

Para el modelo .onnx hemos tenido tiempos de inferencia peores, aunque siguen siendo aceptables, del orden de unos 60 ms y, aunque también perdemos algo de precisión, esta sigue siendo más que aceptable.

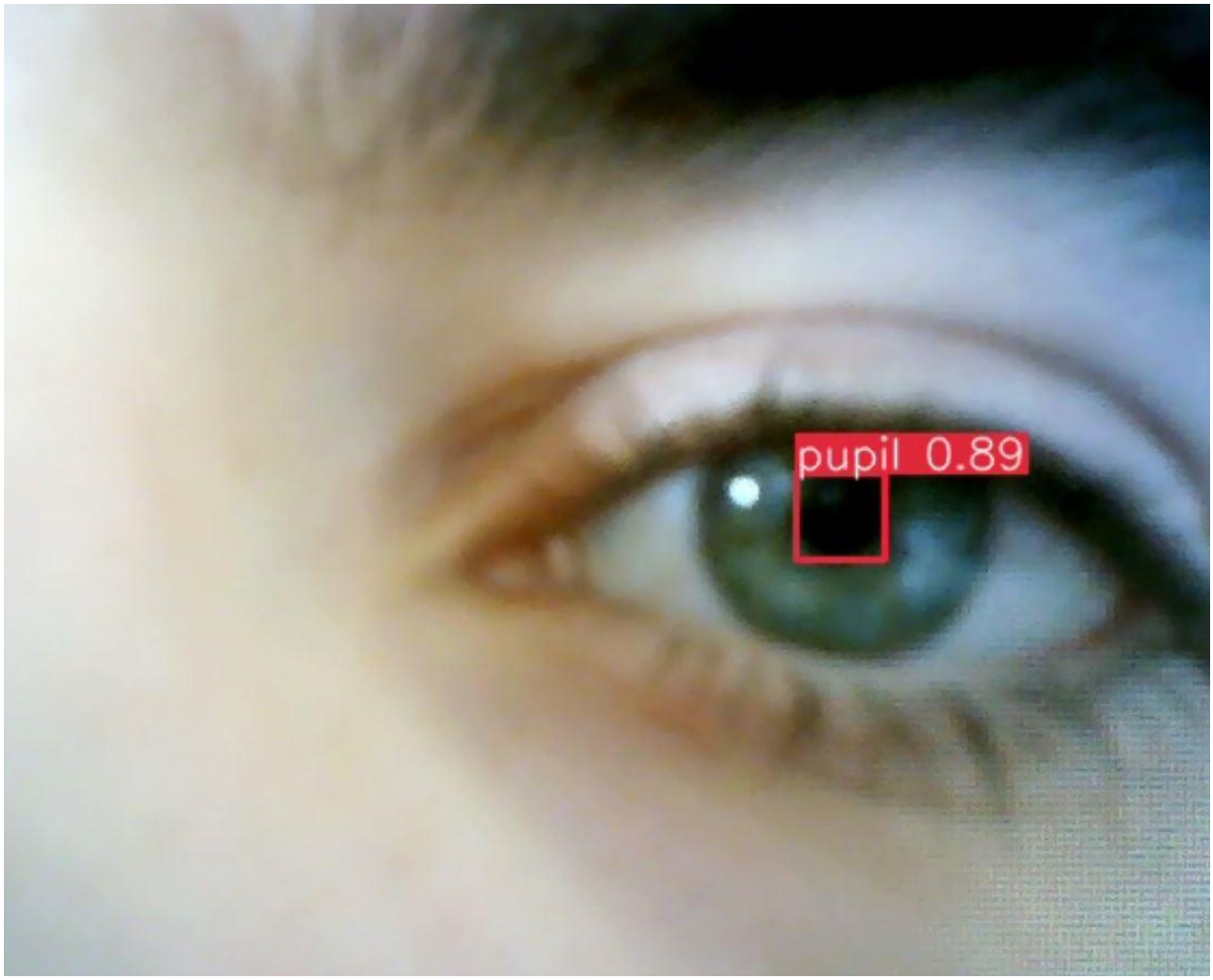

Ilustración 28: Detección de la pupila en pc del modelo .onnx

No obstante, para el modelo exportado a .tflite los resultados son muy malos.Los tiempos de inferencia suben a 1.7 ms, algo inaceptable dado que con esos tiempos apenas es capaz de reconocer nuestra pupila en ningún momento.

Creemos que esto podría deberse a que la cuantización realizada durante la exportación a este tipo de formato no se realiza de forma óptima, reduciendo el tamaño del modelo pero eliminando en el proceso mucha información crucial.

4.2.2 Ejecución en LattePanda

Aquí hemos vuelto a ejecutar nuestro modelo en sus 3 formatos. Para el primero hemos obtenido tiempos de inferencia de 700 ms, algo que no está nada bien y dificulta enormemente la posibilidad de deteción de cualquier objeto durante el video. Aun así, en ciertos frames sí que lo ha detectado, llegando a una precisión máxima de 0.76.

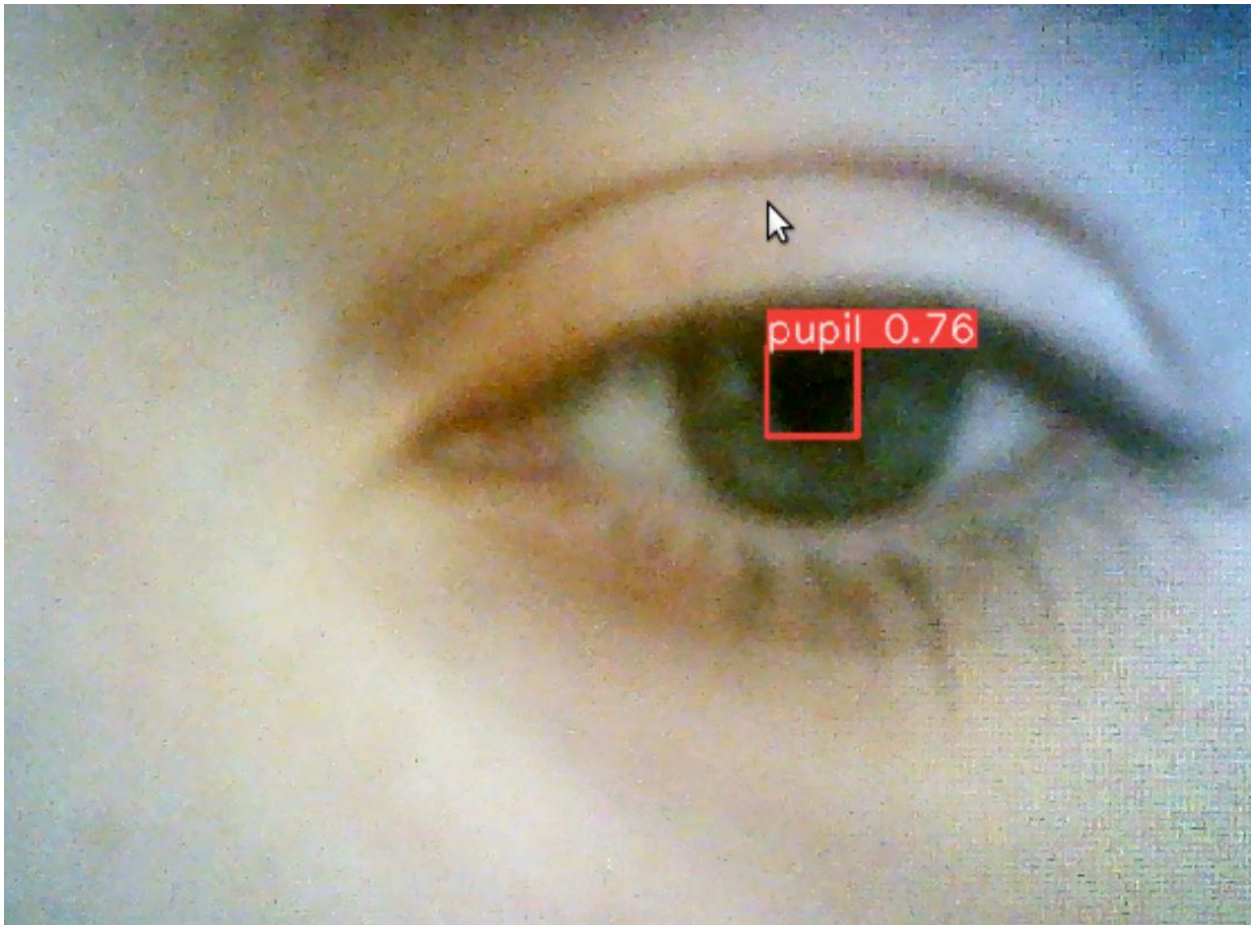

Ilustración 29: Detección de la pupila en LattePanda del modelo .pt

Para el segundo formato, el .onnx, tenemos unos tiempos de inferencia en torno a los 900 ms, tiempos que son muy mejorables y que dificultan la detección de objetos en tiempo real. No obstante, al igual que con el modelo original, se ha conseguido detectar en ciertos frames con una precisión que si bien no entra dentro de un rango muy bueno, tampoco llega a ser muy malo, llegando a un valor máximo de 0.70.

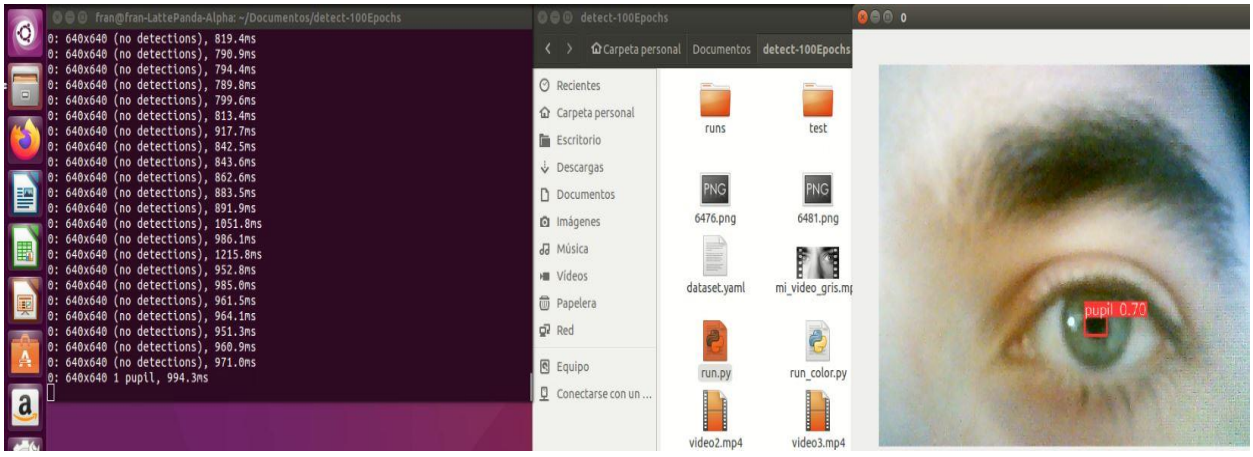

Ilustración 30: Detección de la pupila en LattePanda del modelo .onnx

Para el formato .tflite ni siquiera vamos a adjuntar una imagen del resultado debido a que los

tiempos son totalmente inadmisibles, rondando los 2.2 s como tiempo de inferencia, algo que imposibilita la detección de objetos.

4.3 Conclusión y mejoras a futuro

Como conclusión a este proyecto podemos decir que los tiempos y precisión de la ejecución realizada en el propio ordenador son bastante buenos para los formatos .pt y .onnx de nuestro modelo, por lo que en este aspecto podemos catalogar a YOLOv8 como un éxito en cuanto a soluciones para la detección y seguimiento de la pupila en tiempo real.

En cambio, no ocurre lo mismo para el formato .tflite, quedando totalmente descartado de nuestro abanico de posibilidades. No es así para la ejecución en el dispositivo LattePanda, puesto que si bien hemos conseguido cierto nivel de detección en grabación a tiempo real, no podríamos admitir una aplicación práctica para detección de pupilas en un dispositivo portátil con estos resultados.

Además, es bastante admisible admitir que tenemos pruebas más que suficientes para afirmar que YOLOv8 es una solución al problema de la detección de objetos y, concretamente, del seguimiento de la mirada en nuestro caso, muy buena, versátil y fácil de usar.

Hay que tener en cuenta que estos resultados han sido recogidos con un equipo en general bastante deficiente, algo que ha afectado mucho a la capacidad de entrenamiento de la red, una cámara de baja calidad, lo que ha hecho que el video captado tenga una resolución muy baja y esto dificulta la detección de la pupila para el modelo, y en unas condiciones lumínicas muy determinadas que probablemente afectarán al dispositivo portátil que pueda nacer de la semilla sembrada por este proyecto.

Como posibles mejoras, podríamos incidir en la realización del mismo proyecto con un equipo que posea una mejor gráfica, con el objetivo de poder aumentar la cantidad de datos que toma la red al entrenarse sin que el tiempo de entrenamiento llegue a más de un par de días al superar las 100 épocas de entrenamiento. Algo que mejoraría mucho el desempeño sería una buena calidad de imagen a la hora de captar el video en tiempo real, por lo que una cámara de mayor calidad como la de un pupil labs ayudaría mucho a que los resultados fueran mejores.

Además, se plantea la posibilidad de generar un dataset propio con el fin de entrenar la red neuronal con las entradas más semejantes a las que usaría el sistema final. De este modo se podría aumentar la precisión y se podría adaptar también el tamaño de las imágenes, reduciendo el número de valores de entradas y, como consecuencia, el número de operaciones a realizar.

Por otra parte, añadir como mejora lo que fue catalogado como requisito opcional dentro del apartado de hardware, un sistema de alimentación para hacer LattePanda autónomo. Con esto y el pupil labs integrados en este proyecto, podríamos alcanzar un auténtico producto totalmente portátil y fácil de manejar para el usuario, con un buen diseño y con resultados más que satisfactorios.

De este modo, animo con los resultados de este trabajo a que el departamento y otros compañeros de mi titulación continúen el desarrollo de una solución portátil para seguimiento ocular. Proponiendo como principales vectores de investigación la generación de un dataset de pupilas propio con el sistema pupil-labs, el desarrollo hardware de un sistema autónomo

embebido con aceleración hardware para ejecutar redes neuronales de convolución y el entrenamiento de nuevas redes neuronales que puedan reducir el tiempo de inferencia y mejorar su precisión.

5 REFERENCIAS

1. [Cruz, Y.J.; Rivas, M.; Quiza, R.; Villalonga, A.; Haber, R.E.; Beruvides, G. (December 2021). «*Ensemble of convolutional neural networks based on an evolutionary algorithm applied to an industrial welding process*». *Computers in Industry.* **133**. doi:10.1016/j.compind.2021.103530.

2. ↑ Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". *Biological Cybernetics* **36** (4): 193–202.

3. ↑ LeCun, Yann; Léon Bottou; Yoshua Bengio; Patrick Haffner (1998). "Gradient-based learning applied to document recognition". *Proceedings of the IEEE* **86** (11): 2278–2324.

4. Ultralytics. (2016). Ultralytics GitHub Repository. *https://github.com/ultralytics*

5. Kurdthongmee, W., Kurdthongmee, P., Suwannarat, K., & Kiplagat, J. K. (2022). "A *YOLO Detector Providing Fast and Accurate Pupil Center Estimation using Regions Surrounding a Pupil*". Emerging Science Journal, 6(5), 985-997."

6. Guestrin, E. D., & Eizenman, M. (2006). "*General theory of remote gaze estimation using the pupil center and corneal reflections".* IEEE Transactions on Biomedical Engineering, 53(6), 1124–1133. doi:10.1109/TBME.2005.863952.

7. Cognolato, M., Atzori, M., & Müller, H. (2018). "*Head-mounted eye gaze tracking devices: An overview of modern devices and recent advances".* Journal of Rehabilitation and Assistive Technologies Engineering, 5, 1-13. doi:10.1177/2055668318773991.

8. Sulé Armengol, E. (2019). Project of implementing an intelligent system into a Raspberry Pi based on deep learning for face detection and recognition in real-time. [Degree Thesis, Universitat Politècnica de Catalunya BarcelonaTech]. https://upcommons.upc.edu/bitstream/handle/2117/173058/MEMORIA.pdf?isAllowed=y&sequence=3

9. Ramírez Invernón, Marc. (2015). Método de actuación para el análisis de la percepción visual con Eye-Tracking. [Proyecto final de carrera, Universitat Politecnica de Catanlunya BarcelonaTech]. https://upcommons.upc.edu/handle/2117/77871

10. Reyes Gentil, María. (2017). Registro de patrones de lectura con dispositivos de eye Tracker de bajo coste y estudio de su aplicación para la recomendación de diagnóstico de patologías. [Trabajo fin de máster, Universidad Autónoma de Madrid]. https://repositorio.uam.es/handle/10486/677535d

11. Karthika Renuka, K. R., & Thenmozhi, R. (2006). Using Genetic Algorithm for Eye Detection and Tracking in Video Sequence. International Journal of Computer Applications, 2(7), 25-28. https://www.researchgate.net/publication/228970533_Using_Genetic_Algorithm_for_Eye_Detection_and_Tracking_in_Video_Sequence

12. ProjectSBC. (2019). MagClick Modular LattePanda Alpha Delta Case [Archivo 3D]. Thingverse. https://www.thingiverse.com/thing:3860552

13. Aruba Networks. (2023). ¿Qué es la arquitectura Spine-Leaf? HPE aruba networking. https://www.arubanetworks.com/es/faq/que-es-la-arquitectura-spine-leaf/

14. Takuya Akashi, Yuji Wakasa, Kanya Tanaka, Stephen Karungaru, Minoru Fukumi. (2007). "Using genetic algorithm for eye detection and tracking in video sequence*". Journal of Systemics Cybernetics and Informatics.* https://www.researchgate.net/publication/228970533_Using_Genetic_Algorithm_for_Eye_Detection_and_Tracking_in_Video_Sequence

15. Tárrega Artieda, Asís. (2012). Tracking ocular mediante un Sistema óptico monocular con marcas naturales. [Trabajo final de máster, Universidad politécnica de Valencia]. https://m.riunet.upv.es/bitstream/handle/10251/27241/TFM.pdf?sequence=1&isAllowed=y

16. Agulló Ocampos, Jerónimo. (2021) Análisis de aceleración hardware para autenticación biométrica multimodal. [Trabajo fin de máster, Universidad de Sevilla].

17. J. P. Singh, S. A. S. Jain and U. P. Singh, "Vision-Based Gait Recognition: A Survey," *IEEE Access,* vol. 6, no. doi: 10.1109/ACCESS.2018.2879896, pp. 70497-70527, 2018.

18. T. Cootes, C. Taylor, D. Cooper and J. Graham, "Active Shape Models-Their Training and Application," *Computer Vision and Image Understanding,* no. 61, pp. 38-59, 1995.

19. K. Zhang, Z. L. Zhanpeng Zhang and Y. Qiao, "Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks," *IEEE Signal Processing Letters,* vol. 23, no. 10, pp. 1499-1503, 2016.

20. V. Bazarevsky and e. al., "BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs," pp. 1-4, 2019.

21. K. He, X. Zhang, S. Ren and J. Sun., "Deep residual learning for image recognition," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* pp. 770-778, 2016.

22. Frank Emmert-Streib, Shailesh Tripathi. (2020). "An introductory review of deep learning for prediction models with big data". Frontiers in Artificial Intelligence, vol. 3, no. 1.pp. 1-22. doi:10.3389/frai.2020.00004

6 ANEXOS

ANEXO A – Codigo en Python ‘deteccion_pupila.py’

```python
import cv2
import json
import os
import numpy as np

def detect_pupil(img):
    # Convertir la imagen a escala de grises
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Aplicar un filtro Gaussiano para reducir el ruido
    gray = cv2.GaussianBlur(gray, (5, 5), 0)

    # Aplicar un umbral adaptativo para binarizar la imagen
    thresh = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 2)

    # Aplicar una operación morfológica de apertura para eliminar pequeños
objetos
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)

    # Buscar los círculos en la imagen mediante la transformada de Hough
circular
    circles = cv2.HoughCircles(thresh, cv2.HOUGH_GRADIENT, dp=1,
minDist=10, param1=60, param2=18, minRadius=20, maxRadius=40)

    # Eliminar los reflejos de las gafas en la imagen
    if circles is not None:
        circles = np.uint16(np.around(circles))
        for circle in circles[0]:
            # Extraer la región de la imagen que cubre el círculo
            x, y, r = circle
            mask = np.zeros_like(gray)
            cv2.circle(mask, (x, y), r, 255, -1)
            masked = cv2.bitwise_and(gray, gray, mask=mask)

            # Aplicar la eliminación de reflejos en la región de la imagen
            blurred = cv2.medianBlur(masked, 5)
            edges = cv2.Canny(blurred, 50, 150, apertureSize=3)
            lines = cv2.HoughLines(edges, 1, np.pi/180, 100)
            if lines is not None:
                for line in lines:
                    rho, theta = line[0]
                    if abs(theta) > np.pi/3 and abs(theta) < 2*np.pi/3:
                        a = np.cos(theta)
                        b = np.sin(theta)
```

```
                        x0 = a*rho
                        y0 = b*rho
                        x1 = int(x0 + 1000*(-b))
                        y1 = int(y0 + 1000*(a))
                        x2 = int(x0 - 1000*(-b))
                        y2 = int(y0 - 1000*(a))
                        cv2.line(masked, (x1, y1), (x2, y2), (0, 0, 0), 5)

            # Reemplazar la región de la imagen original con la región procesada
            gray[mask != 0] = cv2.medianBlur(masked, 5)[mask != 0]

        # Devolver el centro y radio del círculo más grande encontrado
        max_radius = 0
        max_circle = None
        for circle in circles[0]:
            if circle[2] > max_radius:
                max_radius = circle[2]
                max_circle = circle
        return max_circle[0], max_circle[1], max_circle[2]
    else:
        return None

def main():
    input_folder = 'D:/Universidad/TFG/Dataset/img_procesado_deteccion'  # Carpeta de entrada
    output_folder = 'D:/Universidad/TFG/Dataset/img_procesado_deteccion/procesadas'  # Carpeta de salida

    # Si la carpeta de salida no existe, la crea
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    for filename in os.listdir(input_folder):
        # Lee la imagen
        img = cv2.imread(os.path.join(input_folder, filename))

        # Detecta la pupila
        pupil_coords = detect_pupil(img)

        if pupil_coords is not None:
            # Dibuja el círculo en la imagen
            cv2.circle(img, pupil_coords[:2], pupil_coords[2], (0, 255, 0), 2)

            # Guarda la imagen con el círculo dibujado
            cv2.imwrite(os.path.join(output_folder, filename), img)

            # Crea un archivo .json con las coordenadas del cuadrado circunscrito al círculo
            json_data = {
                'x': int(pupil_coords[0] - pupil_coords[2]),
                'y': int(pupil_coords[1] - pupil_coords[2]),
                'width': int(pupil_coords[2] * 2),
                'height': int(pupil_coords[2] * 2)
            }
            with open(os.path.join(output_folder,
```

```
f"{os.path.splitext(filename)[0]}.json"), 'w') as f:
                json.dump(json_data, f)

        else:
            print(f"No se pudo detectar la pupila en {filename}")


if __name__ == '__main__':
    main()
```

Anexo B – Código python 'Carpeta_inicial.py'

```
import os
import shutil
import cv2

root_path = "D:/Universidad/TFG/Dataset"
output_path = "D:/Universidad/TFG/Dataset/imagenes"

counter = 0
for casia_foldername in os.listdir(root_path):
    casia_folder_path = os.path.join(root_path, casia_foldername)
    if os.path.isdir(casia_folder_path):
        for foldername in os.listdir(casia_folder_path):
            folder_path = os.path.join(casia_folder_path, foldername)
            if os.path.isdir(folder_path):
                for inner_foldername in os.listdir(folder_path):
                    inner_folder_path = os.path.join(folder_path,
inner_foldername)
                    if os.path.isdir(inner_folder_path):
                        for filename in os.listdir(inner_folder_path):
                            if filename.endswith('.jpg') or
filename.endswith('.jpeg') or filename.endswith('.png'):
                                filepath = os.path.join(inner_folder_path,
filename)
                                img = cv2.imread(filepath)

                                # cambiar extension a jpg
                                output_filename = str(counter) + '.jpg'
                                output_filepath = os.path.join(output_path,
output_filename)

                                cv2.imwrite(output_filepath, img)
                                counter += 1
```

ANEXO C – Código Python ‘import.py’

```
1 from ultralytics import YOLO
2
3 model = YOLO("yolov8m_custom.pt)
4
5 model.export(format="onnx")
```

ANEXO D – Código Python para ejecutar el modelo ‘run_color.py’

```python
import cv2
from ultralytics import YOLO
import numpy as np

def apply_brightness_filter(image, alpha, beta):
    # Aplica el filtro de brillo a la imagen
    filtered_image = cv2.addWeighted(image, alpha, np.zeros(image.shape, image.dtype), 0, beta)
    return filtered_image

def mostrar_video_camara_grises_yolo():
    # Crear objeto para capturar el video de la cámara
    cap = cv2.VideoCapture(0)

    # Verificar que la cámara se haya abierto correctamente
    if not cap.isOpened():
        print("No se pudo abrir la cámara")
        return

    # Crear objeto YOLO
    model = YOLO("yolov8m_custom.tflite")

    # Leer cada frame del video de la cámara, mostrarlo en tiempo real y realizar la predicción con YOLO
    while True:
        # Leer el siguiente frame
        ret, frame = cap.read()

        # Verificar que se haya leído el frame correctamente
        if not ret:
            print("No se pudo leer el siguiente frame")
            break

        # Aplicar filtro de brillo a la imagen
        filtered = apply_brightness_filter(frame, alpha=1.0, beta=-50.0)

        # Realizar la predicción con YOLO en el frame filtrado en BGR
        results = model.track(filtered, show=True, save=True, conf=0.5)

        # Esperar a que se presione la tecla 'q' para salir
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Liberar los recursos
    cap.release()
    cv2.destroyAllWindows()

# Llamar a la función para mostrar el video de la cámara en escala de grises y realizar la predicción con YOLO
mostrar_video_camara_grises_yolo()
```