GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

Escuela de Ingeniería de Fuenlabrada

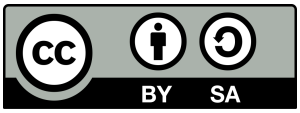
Curso académico 2024-2025

Trabajo Fin de Grado

Estimación de la dirección de la mirada en imágenes del rostro

Tutor: José Miguel Buenaposada Biencinto

Autor: Saúl Navajas Quijano



Agradecimientos

Este trabajo no habría sido posible sin el apoyo recibido por parte de mis seres queridos, que hace cuatro años me animaron a emprender mis estudios en la Robótica Software.

Gracias a mi familia, especialmente a mis padres, Manolo y Yolanda; a mi hermano, David; y a mi hermana, Jessica, por su apoyo incondicional a seguir adelante en mis estudios pese a todas las dificultades que pueda haber encontrado en el camino.

A mis amigos, por formar parte de esta etapa de mi vida y hacer esta aventura más agradable y bonita.

Por último, a mi tutor, José Miguel, por su confianza depositada en mi. Gracias por su tiempo, profesionalidad y amabilidad que me ha brindado durante este proceso.

Dedicado a todos ellos y, en especial, a mi abuela. *Yaya, tú que abandonaste la vida en la recta final de esta etapa que he vivido durante cuatro años, gracias por cuidarme y apoyarme hasta el último instante. Sé que estarás orgullosa de mí.*

Madrid, a 2 de octubre de 2024

Saúl Navajas

Resumen

En España, las distracciones al volante representan un tercio de los accidentes mortales de tráfico[1]. El principal motivo de este es el empleo del teléfono móvil, que desvía la atención del conductor hacia otro punto de interés. Detectar este tipo de imprudencias resulta clave para reducir el número de víctimas en las carreteras. En este contexto, la estimación de la dirección de la mirada puede jugar un papel fundamental como sistema para identificar la conducción distraída o desatenta.

Este Trabajo de Fin de Grado presenta el desarrollo de un modelo de *Deep Learning* para la estimación de la dirección de la mirada junto con otras características faciales a partir de imágenes capturadas por una cámara calibrada. Se han explorado diferentes arquitecturas de redes neuronales convolucionales, incluyendo *EfficientNet* y *ResNet*, con el objetivo de predecir de forma simultánea la dirección de la mirada, el tamaño de las pupilas y la orientación de la cabeza. El modelo ha sido optimizado para trabajar en un entorno con capacidad hardware limitada, y se han empleado técnicas avanzadas de ajuste de hiper-parámetros para mejorar su rendimiento.

En el desarrollo del proyecto, se ha llevado a cabo un análisis de dos de las principales bases de datos empleadas en esta tarea: ETH-XGaze y EVE, siendo esta última la empleada para el entrenamiento y evaluación del modelo. Los resultados obtenidos demuestran un error angular promedio de 7,5 grados en la estimación de la dirección de la mirada. Aunque algunos de los mejores modelos en la literatura logran un error cercano a los 2 grados, los resultados son satisfactorios debido a que el sistema desarrollado no emplea una red adicional de refinamiento de esta estimación y no necesita una capacidad de cómputo excesiva.

[1] `https://www.dgt.es/menusecundario/dgt-en-cifras/`

Acrónimos

TFG *Trabajo de Fin de Grado*

IA *Inteligencia Artificial*

ML *Machine Learning*

DL *Deep Learning*

ADAS *Sistemas avanzados de ayuda a la conducción*

CNN *Convolutional Neuronal Network*

FPS *Fotogramas Por Segundo*

FCN *Fully Convolutional Network*

POG *Point of Gaze*

3DMM *3D Morphable Model*

GAT *Graph Attention Network*

GCN *Graph Convolutional Network*

HDF *Hierarchical Data-Format File*

FC *Fully Connected*

ROI *Region Of Interest*

Índice general

Índice de figuras

Listado de códigos

Índice de cuadros

Capítulo 1
Introducción

De vez en cuando, una nueva tecnología, un antiguo problema y una gran idea se convierten en una innovación

Dean Kamen

La estimación de la dirección de la mirada tiene un impacto significativo en múltiples áreas, como la seguridad vial, la neurociencia o la interacción hombre-máquina o humano-robot. En el contexto de la conducción autónoma, detectar si el conductor está prestando atención a la carretera puede prevenir accidentes mortales y detectar de forma temprana la fatiga, mientras que en entornos de realidad aumentada, permite una interacción más natural con sistemas virtuales. Además, en la investigación médica, el análisis de la mirada ayuda a identificar trastornos neurológicos y psicológicos, especialmente aquellos relacionados a la atención, así como habilitar aplicaciones críticas como es la mejora de la accesibilidad para personas con discapacidades motrices. Todo estas potenciales aplicaciones y su impacto en la calidad de vida hacen que la resolución del problema que involucra estimar de forma precisa la mirada humana resulte tan importante en la actualidad.

En este primer capítulo, se abordaran todas las áreas que orbitan alrededor de este problema que se pretende abordar para poner en contexto la naturaleza que lo rodea, las investigaciones y análisis realizados así como los campos que resultan imprescindibles conocer para abordar el presente TFG.

1.1. Aprendizaje Automático

El *Machine Learning* es la rama de la IA enfocada al desarrollo de modelos que aprenden a generar salidas a partir de datos de entrenamiento. Atendiendo a sus características, los algoritmos de Aprendizaje Automático se pueden clasificar en tres tipos distintos:

- Aprendizaje Supervisado: al algoritmo de aprendizaje se le proporciona un conjunto de datos de entrada con sus correspondientes salidas correctas, de modo que el programa pueda "aprender" la relación entre la salida y el conjunto de entradas. Permite crear tareas de regresión, como la estimación del precio de la vivienda en una ciudad, o de clasificación, por ejemplo para diferenciar correos electrónicos que puedan ser catalogados como "spam".

- Aprendizaje No Supervisado: Se proporciona únicamente al algoritmo un conjunto de datos de entrada, con el objetivo de que el propio algoritmo o el método empleado detecte posibles patrones y estructuras de interés del conjunto de datos proporcionado. La principal aplicación de este tipo de aprendizaje es el *clustering* (segmentación), que puede servir por ejemplo a comercios electrónicos o sitios web de noticias para organizar en categorías sus artículos o productos en función de su contenido. En menor medida, también es utilizado para reducir la dimensión de los datos.

- Aprendizaje Por Refuerzo (*Reinforcement Learning*): A diferencia de los anteriores, el algoritmo aprende a desarrollar la tarea encomendada a base de un esquema de recompensas y penalizaciones ante las decisiones que toma el algoritmo en cada una de las iteraciones. Tienen grandes aplicaciones en los campos de la Robótica y la Conducción Autónoma (navegación autónoma con evitación de obstáculos), así como en los videojuegos (agentes que aprenden a pasar niveles).

1.2. Deep Learning y Redes Convolucionales

Si bien se han explicado las aplicaciones en las tres categorías, hay una común a todas que no se ha mencionado y que cobra un gran protagonismo en la actualidad y en lo que respecta a este trabajo: las **Redes Neuronales**.

Las redes neuronales artificiales son un modelo computacional inspirado en el funcionamiento del cerebro humano. Está compuesta por un conjunto de neuronas, que forman a su vez capas de neuronas interconectadas entre sí. Una red puede estar formada por N capas, donde la primera es la capa de entrada y la última la capa de salida. Entre ambas, puede haber M capas ocultas. En la figura 1.1 se muestra un ejemplo de una red con una única capa oculta.

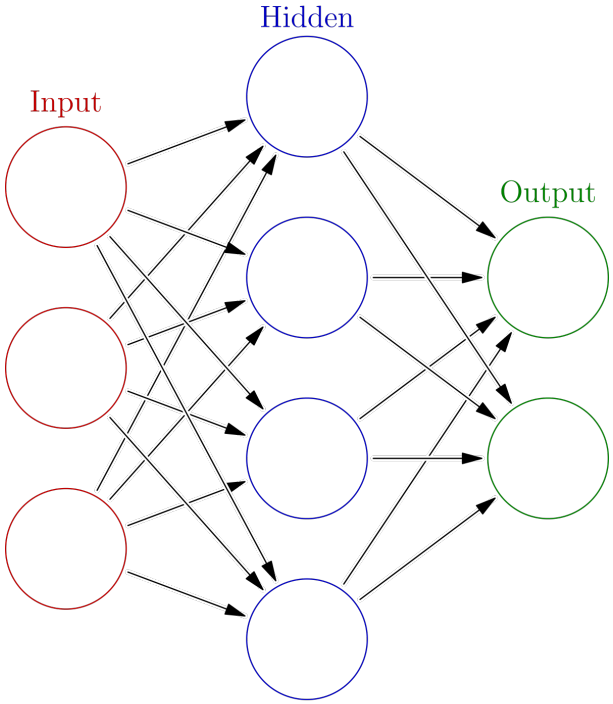


Figura 1.1: Estructura de una red neuronal.

Fuente: Wikipedia

La neurona (también conocida como perceptrón), es la unidad básica de procesamiento de una red. Simula la forma en la que una neurona biológica procesa y transmite señales en el sistema nervioso. Un perceptrón consta de los siguientes elementos principales:

- Entradas: Son los objetos de entrada que recibe la red, que pueden provenir de otras neuronas. Por ejemplo, si la entrada es una imagen, los objetos de entrada podrían ser los píxeles. Un perceptrón puede tener una o múltiples entradas.
- Pesos: Cada entrada tiene asociada un peso, que determina su importancia relativa de esa entrada en la neurona. Los pesos se ajustan durante el proceso iterativo de entrenamiento de una red.
- Umbral o sesgo: Es un valor que llevan asociadas algunas neuronas. Este valor se suma al producto de las entradas por sus pesos, y pueden ayudar a mejorar la sensibilidad del perceptrón sobre ciertos valores de entrada.
- Función de activación o decisión: Determina, una vez sumadas las entradas ponderadas, si la neurona "dispara" o no. Es decir, si produce un valor de salida

activado o no activado basada en dicha suma.

El perceptrón aprende en base a un algoritmo recursivo: tiene unos valores de pesos y sesgos inicializados aleatoriamente. Aplica el valor de salida del sumatorio de cada entrada por su peso más el sesgo, y en base a ese valor se aplica la función de decisión. A raíz del error obtenido en la salida (diferencia entre el valor deseado y el producido), se actualizan los pesos y su término independiente, para volver a realizar una nueva iteración. El proceso finaliza una vez el número de errores es menor a un umbral determinado o cuando se alcanza un número de iteraciones en concreto.

En las redes neuronales multi-capa y de aprendizaje supervisado, se aplica el método de retropropagación: los errores en la salida de la red neuronal se propagan hacia atrás, por lo que se calcula la contribución global del error en cada capa, y, por consiguiente, permite ajustar los pesos de las conexiones entre las distintas neuronas de la red. Estas redes aumentan su complejidad debido al número de capas, pero permiten crear redes mucho más potentes para una gran variedad de aplicaciones. El *Deep Learning* es la rama, dentro del *Machine Learning*, que estudian estos modelos de múltiples capas en cascada, que han permitido crear una gran variedad de aplicaciones en diferentes sectores:

- Salud y medicina: Las redes neuronales profundas se utilizan para analizar imágenes médicas, como radiografías o tomografías computarizadas, permitiendo la detección temprana y diagnóstico de enfermedades.
- Agricultura: Permiten la monitorización inteligente y la creación de mapas de salud de cultivos, para la detección de posibles plagas, enfermedades de las plantas o predecir los puntos de sequía de una zona con anterioridad.
- Asistencia virtual: Interpretación y procesamiento del lenguaje natural junto con el reconocimiento de voz. *Chatbots* especializados en mantener conversaciones como Siri o Alexa.
- Seguridad: Sistemas de seguridad y vigilancia con reconocimiento facial o detección de patrones de fraudes o transacciones ilegales.
- Conducción Autónoma: Los vehículos autónomos realizan una fusión sensorial de sus datos y los procesan utilizando redes profundas para la detección de peatones, señales o carriles, entre otros muchos objetos. Aplicaciones en el uso privado (Tesla), sector servicios (Waymo) o el sector de la logística (Aurora), entre otros.

Se puede observar como la gran mayoría de aplicaciones involucran el tratamiento de imágenes del mundo real con el fin de producir información numérica que pueda ser tratada por un ordenador. Es así como la disciplina de la Visión Computacional trata de imitar el comportamiento de nuestros ojos y cerebro para comprender el mundo que nos rodea. En la Visión Computacional clásica, el enfoque principal ha sido la extracción manual de características y la utilización de algoritmos específicos para tareas como segmentación y reconocimiento de patrones. Sin embargo, con los avances en las áreas del *Machine Learning* y *Deep Learning*, la Visión Artificial moderna hace empleo de redes neuronales capaces de interpretar y procesar imágenes de forma muy eficaz, como las redes convolucionales (CNN), aprendiendo características de forma automática, permitiendo un enfoque más generalista y robusto.

Estas redes convolucionales son un tipo de red neuronal diseñadas específicamente para procesar datos con estructuras en cuadrícula, como son las imágenes. Su arquitectura se compone principalmente de capas convolucionales, que aplican filtros a la imagen de entrada para extraer características locales, como bordes y texturas. Cada filtro o *kernel* se desliza sobre la imagen y realiza una operación de convolución, lo que permite a la red identificar patrones en diferentes regiones. Las capas de activación, como ReLU, se utilizan para introducir no linealidades en el modelo, mientras que las capas de agrupamiento (pooling) reducen la dimensión y la complejidad computacional al mantener las características más relevantes. Al final, se utilizan capas totalmente conectadas para transformar las características extraídas en predicciones de clase. Aunque fueron introducidas en los años 80, su popularidad aumentó significativamente en 1998 con LeNet-5[7] y se consolidó en 2012 con AlexNet[1], marcando un hito en el campo de la visión por computadora.

Una de las CNN más importantes son las redes residuales (más conocidas por su abreviación, *ResNets*)[5], entre las que cabe destacar ResNet50, con un total de 50 capas con pesos entrenables; o ResNet18, una variante más "ligera" y menos profunda en cuanto a número de capas refiere. En este tipo de redes, las capas de pesos aprenden funciones residuales con referencia a las entradas de las capas. De esta forma, el bloque consta de una ruta residual y otra ruta con conexión "atajo" que la soslaya. La rama residual del bloque se compone de dos capas de pesos sinápticos intercalados por una función rectificadora. El resultado es la suma de ambas, tal y como se observa en la figura 1.2:

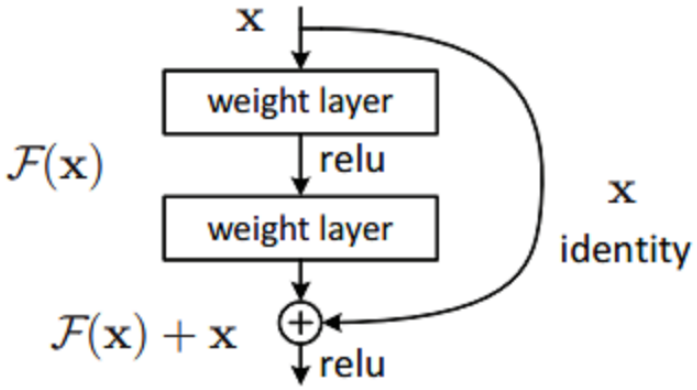


Figura 1.2: Bloque residual de una ResNet

Fuente: WikimediaCommons

Matemáticamente puede comprenderse de la siguiente manera. Sea x la entrada del bloque, y la salida producida a partir de esa entrada como *H(x)*. El residuo, que llamaremos *F(x)*, sería la diferencia entre ambas:

$$F(x) = Salida - Entrada = H(x) - x \tag{1.1}$$

Reordenando la expresión, podemos expresar el bloque residual (esto es, la salida), como la suma de ambas ramas:

$$H(x) = F(x) + x \tag{1.2}$$

No solo la introducción de este tipo de redes han mejorado los resultados en este área de investigación. Los errores de estimación en *datasets* y modelos públicos en el ámbito de la dirección de la mirada han mejorado rápidamente con el paso del tiempo gracias a dominios de adaptación que involucran redes de Bayes, CNNs multi-región o enfoques como *coarse-to-fine*. Este último hace referencia a una estrategia que se basa en realizar un proceso de refinamiento progresivo, comenzando desde una representación inicial más general o "gruesa" (*coarse*), y luego ir refinando gradualmente los detalles hasta obtener una representación más detallada o "fina" (*fine*). Es una técnica muy utilizada en visión artificial y procesamiento de imágenes, dado que en ciertos problemas complejos la información de alta resolución, como son las imágenes, no es necesaria en las primeras etapas del procesamiento de los datos, dónde es preferible un análisis de imágenes de menor resolución que permitan extraer

características más generales, además de permitir una mayor frecuencia de iteración durante la etapa de entrenamiento de una red.

Esto ha llevado a un cambio significativo en la forma en la que se abordan problemas como la detección y extracción de características en imágenes del ser humano, permitiendo crear aplicaciones que beneficien al conjunto de la sociedad, especialmente en materia de salud y seguridad. En la Unión Europea, los vehículos homologados desde julio de 2022, así como los vendidos a partir de 2024, deben contar obligatoriamente con sistemas avanzados de ayuda a la conducción (ADAS) para detectar la fatiga y sueño[4] en los conductores, responsable del 11 % de los accidentes mortales en vías interurbanas (ver figura 1.3).

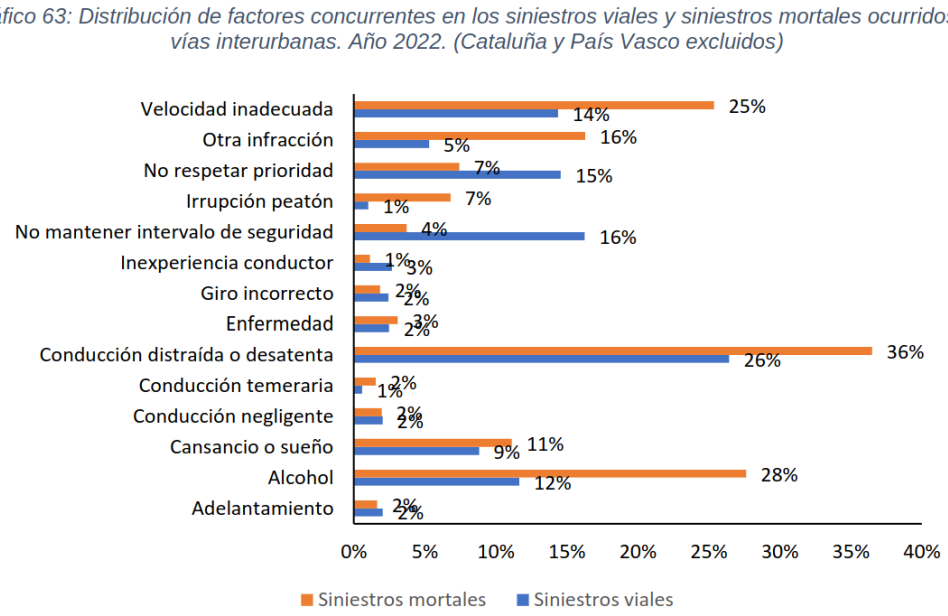


Figura 1.3: Distribución de factores en siniestros viales y mortales en vías interurbanas.

Fuente: Dirección General de Tráfico (DGT), 2022

Las nuevas herramientas de DL desarrolladas y las normativas pioneras en materia de seguridad vial aceleran las investigaciones para crear nuevos ADAS capaces de detectar otros factores de riesgo como la distracción al volante.

Este TFG se centra en desarrollar un sistema de estimación de la dirección de la mirada mediante el análisis de rostros humanos, poniendo el foco en su posible incorporación en un sistema avanzado de ayuda a la conducción para investigar su

potencial en la detección de la conducción desatenta o distraída.

1.3. Sistemas de estimación de la dirección de la mirada en imágenes

La mirada humana puede verse como un sistema en bucle cerrado retro-alimentado, donde la aparición de un objeto de interés en el campo visual provoca un movimiento de los ojos hacia él para poder observarlo. En caso de que ese objeto o el observador sufra un desplazamiento, la dirección es corregida para ponerlo de nuevo en el foco de atención (figura 1.4).

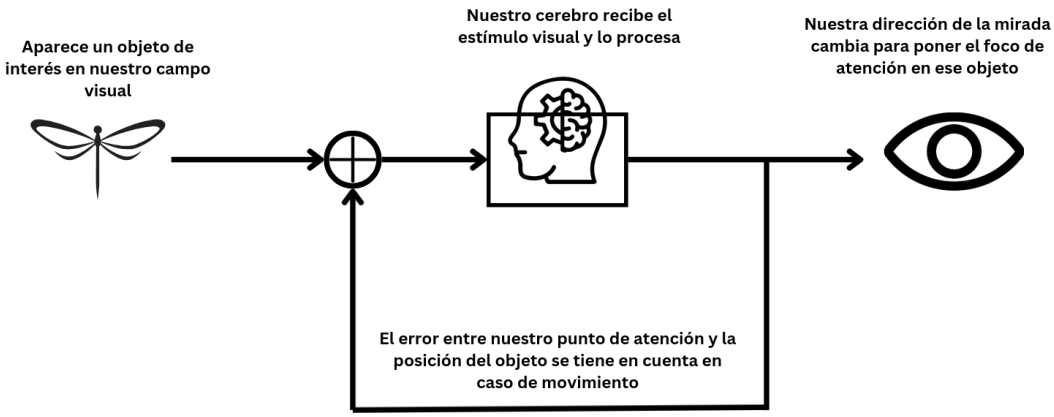


Figura 1.4: Sistema de bucle cerrado retro-alimentado de la dirección de la mirada humana

Los estudios sobre los movimientos oculares y la dirección de la mirada se remontan a hace más de 200 años. El francés Louis Emile Javal descubrió en 1879 que la lectura no implica un suave barrido de los ojos a lo largo del texto, como se suponía, sino una serie de paradas cortas llamadas fijaciones junto con "sacadas" rápidas (pequeños movimientos parecidos a tirones durante las fijaciones)[22]. Esto planteó importantes cuestiones sobre qué palabras se detenían los ojos y por cuánto tiempo. Con el paso del tiempo, las investigaciones han analizado el impacto de las señales fisiológicas y nerviosas en la estimación de la dirección ocular, como son la dilatación de las pupilas y el parpadeo de los ojos. Los cambios de tamaño en la pupila pueden indicar carga cognitiva y excitación emocional[2]. Además, los mencionados movimientos sacádicos introducen ruido en los datos de la mirada pero también sugieren un enfoque de atención[20], al permitir realizar una construcción mental de la escena rápidamente a partir de varios puntos. Factores externos, como la luz ambiental también puede influir

en la estimación de la dirección ocular debido a su impacto en el tamaño de las pupilas y la calidad de las reflexiones usadas en el seguimiento. Estos descubrimientos y estudios han permitido el desarrollo de métodos y técnicas para la estimación de la dirección de la mirada, que pueden dividirse en: (1) geométricos, (2) basados en la apariencia en imágenes de los ojos y (3) los híbridos que mezclan geometría y apariencia.

Podríamos destacar aquellos que toman un enfoque meramente basado en modelos geométricos [14]. Estos basan su funcionamiento en extraer características como el centro de la pupila de la imagen, "mapeando" estas características 2D en un modelo genérico facial en 3D para determinar la línea de visión y calcular la intersección con la pantalla para encontrar el punto de mira. Otros modelos geométricos consisten en dirigir una luz infrarroja hacia el ojo y capturar la imagen con una cámara de seguimiento ocular[3]. Además de identificar el centro de la pupila, también se calcula el reflejo de la luz infrarroja en la córnea, lo que permite calcular el vector entre estos puntos y determinar el punto de mira en la pantalla utilizando parámetros como la distancia entre la cámara y el ojo o la propia geometría ocular (ver figura 1.5).

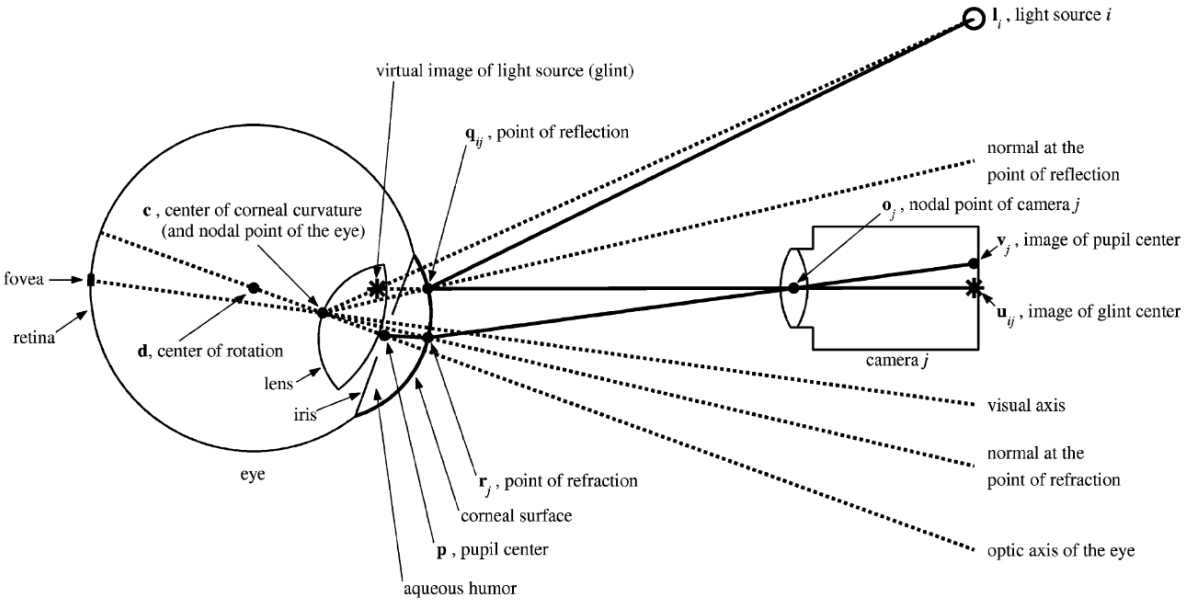


Figura 1.5: Diagrama del rastreo ocular junto con el punto de reflexión en la córnea

Fuente: [3]

Otro enfoque generalizado es la apariencia, en los que podemos encontrar los modelos de aprendizaje automático y *Deep Learning*. En los primeros, se extraen

características de la imagen, como la posición de la cabeza y la ubicación de la pupila, para luego entrenar un modelo utilizando Máquinas de Soporte Vectorial (SVM)[16] o redes neuronales junto con datos etiquetados que asocien dichas características con el punto al que se mira. Los modelos de *Deep Learning* basados en Redes Neuronales Convolucionales (CNNs) no necesitan la extracción de características previas y el modelo aprende a predecir directamente el punto al que se mira a partir de la imagen, necesitando un gran número de imágenes de entrenamiento con la dirección de la mirada etiquetada.

Por último, podríamos destacar los que aúnan ambos enfoques para construir un sistema híbrido. En estos modelos, se combinan la información geométrica y la de apariencia para una mayor precisión y robustez en la estimación. MPIIGaze[19] emplea este enfoque fusionando los puntos de datos de la posición de la pupila (geométrico) y la detección de otros puntos de interés sobre el rostro junto con la imagen completa del ojo (apariencia) normalizada que se emplea como entrada en una CNN (ver figura 1.6).

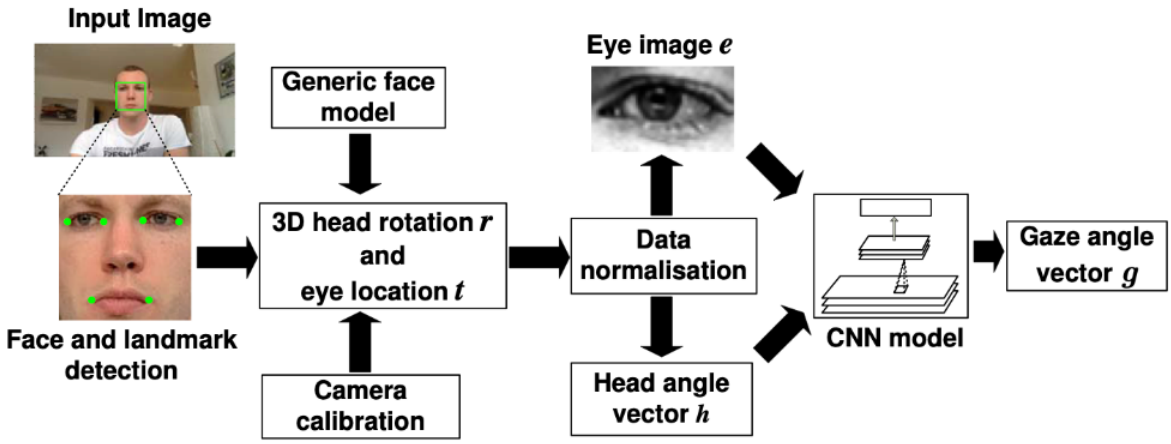


Figura 1.6: Diagrama del modelo MPIIGaze

Fuente: MPII Gaze (IEE Paper)

Todas estas características subrayadas, en suma con los tipos de redes neuronales existentes más empleadas y las investigaciones que se han realizado a lo largo de la historia, destacadas en la presente sección, constituyen los pilares fundamentales sobre los cuales un desarrollador debe construir y basar sus contribuciones en este campo. Ya

introducidos estos elementos, ya es posible avanzar a las secciones específicas de este TFG.

Capítulo 2
Objetivos

En el capítulo primero se ha descrito e introducido la naturaleza, motivación y contexto del presente Trabajo de Fin de Grado. En este nuevo capítulo, se presentarán los objetivos específicos contemplados en el proyecto, así como los requisitos que se han de cumplir para conseguirlos y la metodología empleada para alcanzarlos.

2.1. Descripción del problema

Se pretende abordar la estimación de la dirección de la mirada a través de un modelo de aprendizaje profundo. El sistema tomará como entrada un conjunto de imágenes recortadas del rostro facial, todas ellas con un tamaño fijo. A partir de estas imágenes y con ayuda de una red convolucional pre-definida, el sistema producirá en su salida un conjunto de estimaciones: la dirección de la mirada en forma de vector en coordenadas esféricas y el tamaño de las pupilas de los ojos en milímetros. Adicionalmente, se añade a la salida del sistema la estimación de orientación de la cabeza debido a su relación con la dirección de la mirada, en el mismo formato que esta última.

2.2. Objetivos

En este TFG se plantea como **principal objetivo** desarrollar mediante técnicas de Aprendizaje Profundo un sistema de estimación de la dirección de la mirada humana.

Con el fin de abordar el objetivo propuesto correctamente, se establecen una serie de **objetivos secundarios**:

1. Creación y desarrollo del sistema de D.L de estimación de la dirección de la mirada humana en conjunción con un modelo desarrollado con anterioridad por el tutor para la estimación de la orientación de la cabeza. Esto tiene como fin potenciar y ampliar los campos de aplicación del modelo base.

2. Investigación sobre el impacto del tipo de red empleado como base del modelo neuronal en términos de eficacia y eficiencia. En concreto, se plantea comparar dos redes convolucionales de uso popular: *ResNet* y *EfficientNet*.

3. Elaboración de un informe de resultados detallado del sistema, en el que se muestren la evolución del aprendizaje del modelo de forma gráfica, la evaluación de métricas de rendimiento y análisis de los resultados obtenidos de los entrenamientos de las diferentes configuraciones del sistema de estimación de la mirada.

4. Realización de pruebas técnicas del modelo generado y entrenado a partir de vídeos propios grabados a partir de la cámara de un portátil y variación de las características del conjunto de datos.

2.3. Requisitos

Se establecen los siguientes requisitos mínimos para la realización del trabajo:

- El sistema de estimación de la dirección de la mirada debe emplear una base de datos preparada para el mismo fin y que disponga de una variedad en el sexo, etnia y ausencia o presencia de gafas en los participantes que la compongan, de forma que se pueda realizar un estudio de aplicación general y realista.

- El desarrollo del sistema ha de ser realizado en una máquina que disponga de al menos una unidad de procesamiento gráfico y una capacidad de memoria suficiente para guardar una base de datos extensa.

- Se emplea el framework público pcr-upm/images_framework [1] para la gestión de imágenes, anotaciones, visualización y submódulos que permiten seguir las pautas de diseño y funciones empleadas en el sistema base de orientación de la cabeza.

2.4. Metodología

El Trabajo de Fin de Grado se inició a principios de abril de 2024 y se finalizó a principios del mes de octubre de ese mismo año. Durante los seis meses de realización del proyecto, se siguieron las siguientes pautas:

- Reuniones de seguimiento cada dos semanas para analizar lo conseguido hasta la fecha y establecer las siguientes pautas y objetivos a alcanzar. De esta manera, se

[1] `https://github.com/pcr-upm/images_framework`

ha podido seguir el camino planteado inicialmente sin desviarnos excesivamente de él, solucionando y resolviendo dudas de manera efectiva.

- Documentación periódica desde el primer día de las tareas y avances que se realizan para poder tener claro en todo momento qué se ha hecho y cómo. Esta documentación era posteriormente volcada en el presente informe de manera que se puede explicar los diferentes apartados y sub-apartados de manera más detallada y sin olvidar detalles importantes.

- Se empleó la plataforma de *GitHub*[2] para crear un repositorio donde tener guardado el código del TFG. Además, se creó una carpeta de acceso público para los estudiantes y docentes de la URJC en *OneDrive*[3] con documentación adicional referente al proyecto.

- Adicionalmente, se utilizó *Microsoft Teams* y el correo de alumno/profesor de la universidad para poder comunicar avances y plantear dudas de forma cómoda y sin necesidad de esperar a las reuniones quincenales.

- Como parte de la gestión del tiempo y tareas, empleé la metodología Scrum para adaptar las tareas a *sprints* de dos semanas, lo que me permitió gestionar de manera efectiva el progreso de mi proyecto. Adicionalmente, utilicé *Trello* como herramienta principal para organizarme (ver figura 2.1); esta plataforma me permitió crear tableros visuales donde dividí las tareas en listas y tarjetas, facilitando el seguimiento del estado de cada actividad. Gracias a esta metodología y a la organización en Trello, pude mejorar la gestión de mi tiempo y asegurarme de cumplir con los plazos establecidos tras cada reunión con mi tutor.

[2] `https://github.com/pcr-upm/saul24_gaze`

[3] `https://urjc-my.sharepoint.com/:f:/g/personal/s_navajas_2020_alumnos_urjc_es/El0EpzPBnKNAlCV6YOsSK1ABhoXwbtjMLeFcYhXy2_-y6g?e=MJJE5m`

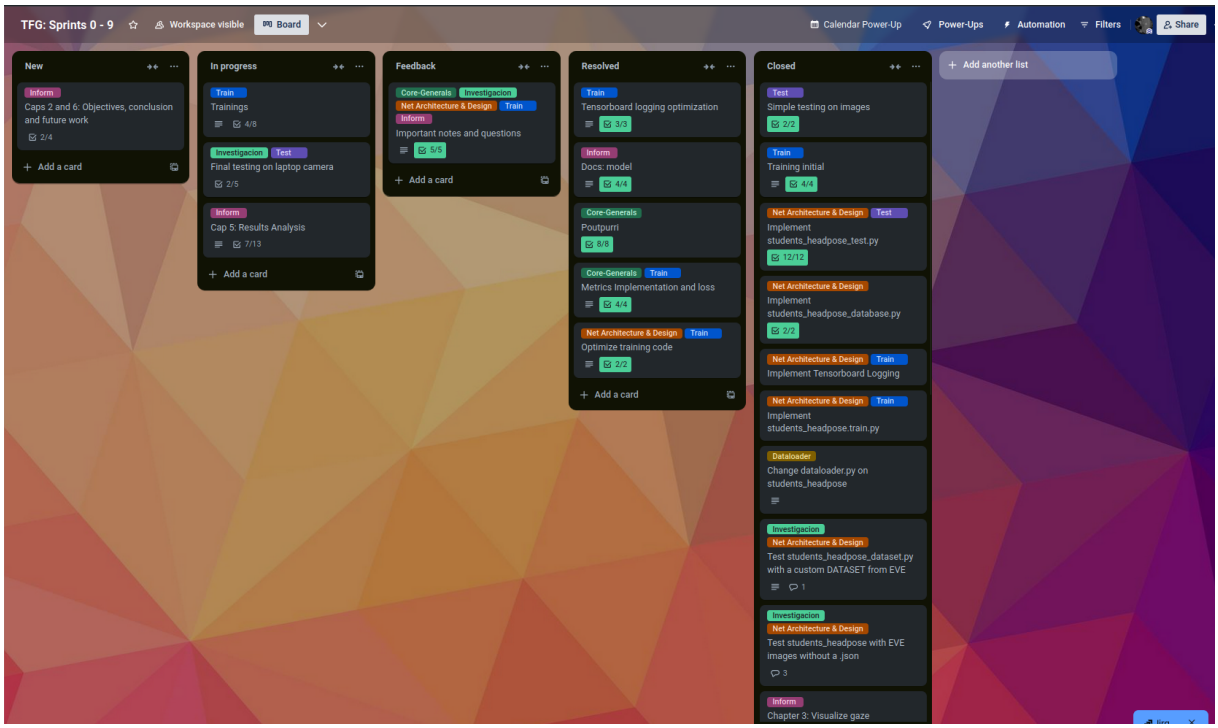


Figura 2.1: Tareas organizadas en Trello (Sprints 0 al 9)

2.5. Plan de trabajo

El plan de trabajo a seguir durante los 6 meses de duración del TFG fue el siguiente, enumerados de forma temporal:

1. Búsqueda de información y estudio del contexto histórico: esta etapa involucró investigaciones sobre el comportamiento humano de la mirada al percibir un estímulo, los movimientos oculares sacádicos presentes al percibir una nueva escena y las investigaciones realizadas en el campo de la dirección de la mirada humana y su estimación.

2. Análisis y comparación de bases de datos en este campo, con el fin de obtener las características comunes que se emplean y técnicas novedosas que puedan ayudar a la estimación.

3. Implementación del dataloader en PyTorch del modelo en base al conjunto de datos (*dataset*) escogido, para poder cargar los datos de la manera más óptima, rápida y eficiente.

4. Estudio de la adopción del dataloader y el resto de código sobre el sistema base estimación de la orientación de la cabeza junto con el framework de students_-headpose proporcionado por el tutor.

5. Preparación del código de entrenamiento, evaluación y pruebas del nuevo sistema completo, en el que se construyen una variedad de configuraciones diferentes para analizar el impacto de ciertos hiper-parámetros y componentes en el modelo neuronal.

6. Entrenamiento, test y análisis de resultados, de modo que permitan sacar las conclusiones adecuadas sobre el sistema desarrollado en el proyecto.

La realización del presente informe se realizó de forma paralela a las etapas que se han expuesto, tal y como se explicó anteriormente.

Capítulo 3
Plataformas de desarrollo y herramientas utilizadas

En este capítulo se detallan las metodologías y herramientas empleadas para la elaboración de este trabajo. Además del propio lenguaje de programación y las correspondientes librerías, se ha hecho empleo de bases de datos y redes neuronales específicas debido a la naturaleza del proyecto.

3.1. Lenguajes de Programación

3.1.1. Python

Python[1] es un lenguaje de programación de alto nivel interpretado, de tipado dinámico y multiplataforma. Posee una sintaxis clara y concisa, así como un extenso ecosistema de librerías y frameworks disponibles que lo convierten en el lenguaje por referencia en el ámbito del *Machine Learning* y las Redes Neuronales.

La elección de Python como lenguaje de programación para el desarrollo de este trabajo se debe principalmente a que el modelo base del que se parte, una red convolucional para estimación de la orientación de la cabeza, está escrito en **PyTorch**, un *framework* en dicho lenguaje de programación.

3.2. Entorno de programación

3.2.1. Visual Studio Code

También conocido como VS Code[2], es un editor de código fuente gratuito desarrollado por Microsoft y compatible con multitud de lenguajes de programación

[1] `https://www.python.org/`
[2] `https://code.visualstudio.com/`

y disponible para Windows, MacOS, Linux y Web. Según la encuesta elaborada por Stack Overflow Survey 2023 [3] , es el principal IDE empleado por más del 73 % de los programadores.

Gracias a su variedad de extensiones, permite integrar el control de versiones con git, acceder a máquinas remotas mediante *Secure Shell* (SSH) o visualizar directamente vídeos o imágenes, lo cual ha facilitado y flexibilizado el desarrollo de este trabajo.

3.2.2. Entorno virtual de programación con Conda

A la hora de emprender un nuevo proyecto software de este ámbito, es importante poder gestionar con total libertad y eficiencia los distintos paquetes y dependencias necesarias para desarrollarlo. Además, es importante que estos no interfieran de los requerimientos empleados para otro proyecto, versiones diferentes, etc. Para ello, se ha empleado un entorno virtual de programación con Conda, un gestor de paquetes y sistema de gestión de entornos de código abierto.

De esta forma, podemos crear, exportar o actualizar entornos aislados, cada uno con versiones diferentes de Python y otros paquetes mediante un simple comando que activa un entorno u otro. En concreto, se ha empleado en una máquina en remoto compartida por varios usuarios para poder desarrollar correctamente TFG sin interferir con los demás proyectos en curso.

3.2.3. Máquina Confucio de la Universidad Rey Juan Carlos

El entrenamiento de CNNs supone procesar miles de imágenes repetidamente. Para ello, es necesario contar con un equipo hardware suficientemente potente para poder desarrollar y programar el modelo sin tener limitaciones de procesamiento ni memoria. Se ha empleado la máquina **Confucio** que dispone la Escuela Técnica Superior de Ingeniería Informática de la URJC en su campus de Móstoles.

La máquina en cuestión cuenta con las siguientes características:

- **Almacenamiento**: dos discos de 10TB cada uno.
- **Procesamiento**: AMD Ryzen 5 3600 6-Core Processor con una velocidad máxima de reloj de hasta 4200MHz, 12 núcleos.

[3] `https://survey.stackoverflow.co/2023/`

- **Aceleración gráfica**: dos gráficas NVIDIA GeForce RTX 2060 Super de 8GB cada una (ver figura 3.1)

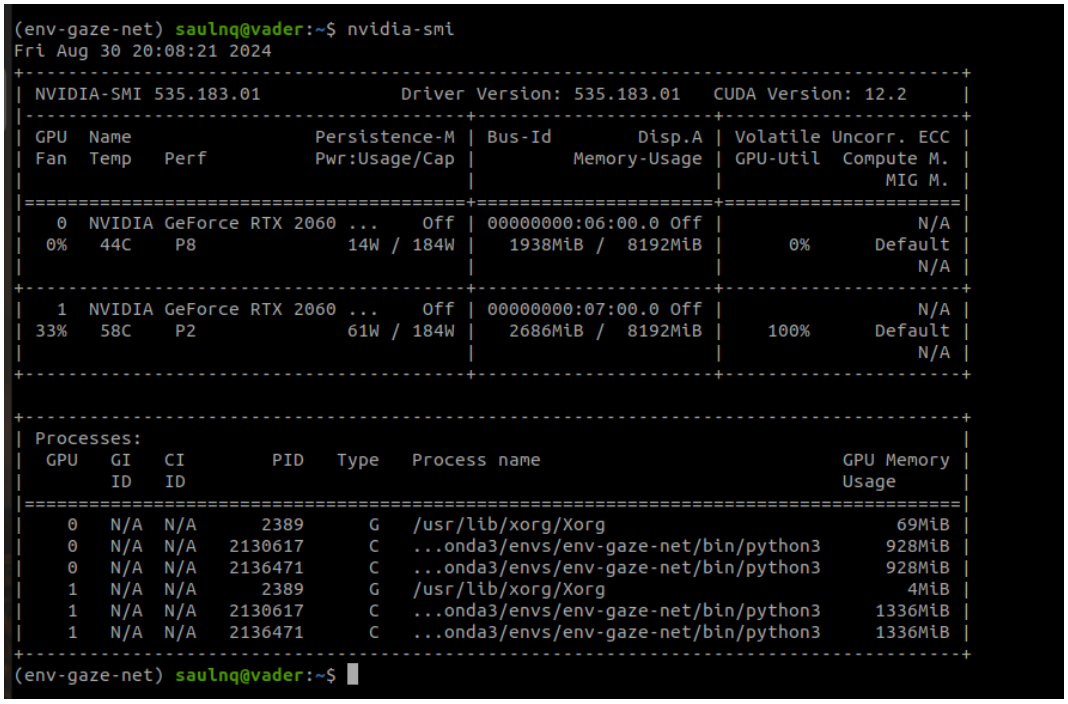


Figura 3.1: Información de las GPUs de la máquina Confucio

3.3. Librerías

3.3.1. OpenCV

[4] Es una biblioteca de código abierto diseñada para el procesamiento de imágenes y visión computacional. Ofrece una amplia gama de herramientas, funciones y métodos para realizar tareas que involucran la detección de objetos, reconocimiento de patrones, seguimiento de los ojos o la calibración de cámaras, entre otros. Fue desarrollada inicialmente por *Intel Corporation*. En el presente TFG, hemos utilizado la funcionalidad de calibración de cámaras que ofrece OpenCV (ver figura 3.2)

[4] `https://opencv.org/`

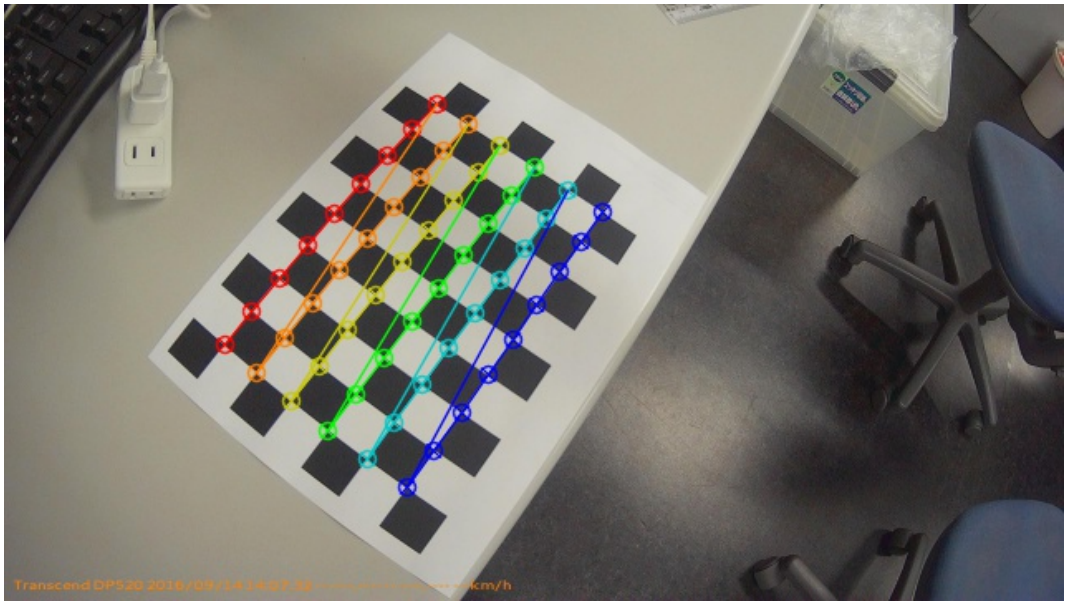


Figura 3.2: Ejemplo de calibración de cámara con OpenCV usando un tablero de ajedrez

3.3.2. PyTorch

Es un framework de *Aprendizaje Automático* y de código abierto para Python. Desarrollada principalmente por *Facebook's AI Research Lab (FAIR)*[5], destacada por su capacidad para realizar cómputos en tensores de manera dinámica, facilitando la construcción de modelos complejos y la depuración.

Aporta la ventaja de que puede ejecutarse en una unidad de procesamiento gráfico (GPU), lo cual es ideal para procesos que requieren de cálculos masivos como el entrenamiento o *fine-tunning* de redes neuronales con bases de datos excesivamente grandes, como las que se abordan en el presente TFG.

3.3.3. PyTorch Lightning

Lightning[6] es una biblioteca de Python *free-source* que proporciona una interfaz de alto nivel para Pytorch. Tiene un alto rendimiento y permite realizar modelos de Aprendizaje automáticos más escalables y fáciles de interpretar, al organizar el código de forma simple y permite manejar las utilidades de Pytorch de manera más flexible.

[5] `https://pytorch.org/`
[6] `https://lightning.ai/docs/pytorch/stable/`

Los motivos que llevan a emplear esta librería en el desarrollo del presente trabajo son los siguientes:

1. Simplificación del código: Lightning se encarga de muchas de las tareas comunes en el entrenamiento y pruebas de modelos de DL, como es el ciclo de entrenamiento o la gestión de dispositivos (CPU, GPU). El resultado es que con Lightning se escribe menos código y se reutiliza el que ya está probado en biblioteca.

2. Escalabilidad y flexibilidad: la estructura de la librería en cuestión permite amplificar la modularización del modelo y la escalabilidad, al simplificar su posterior modificación o ampliación.

3. Eficiencia: la posibilidad de entrenar de manera simultanea el modelo en múltiples GPUs y TPUs (unidades de procesamiento tensorial) de manera simplificada y el uso de hooks y callbacks para personalizar el entrenamiento y desarrollar más rápido código para las etapas de entrenamiento o evaluación.

Lightning está construido por encima de *Lightning Fabric*, el framework ligero que maximiza el control sobre el entrenamiento de PyTorch (ver figura 3.3).

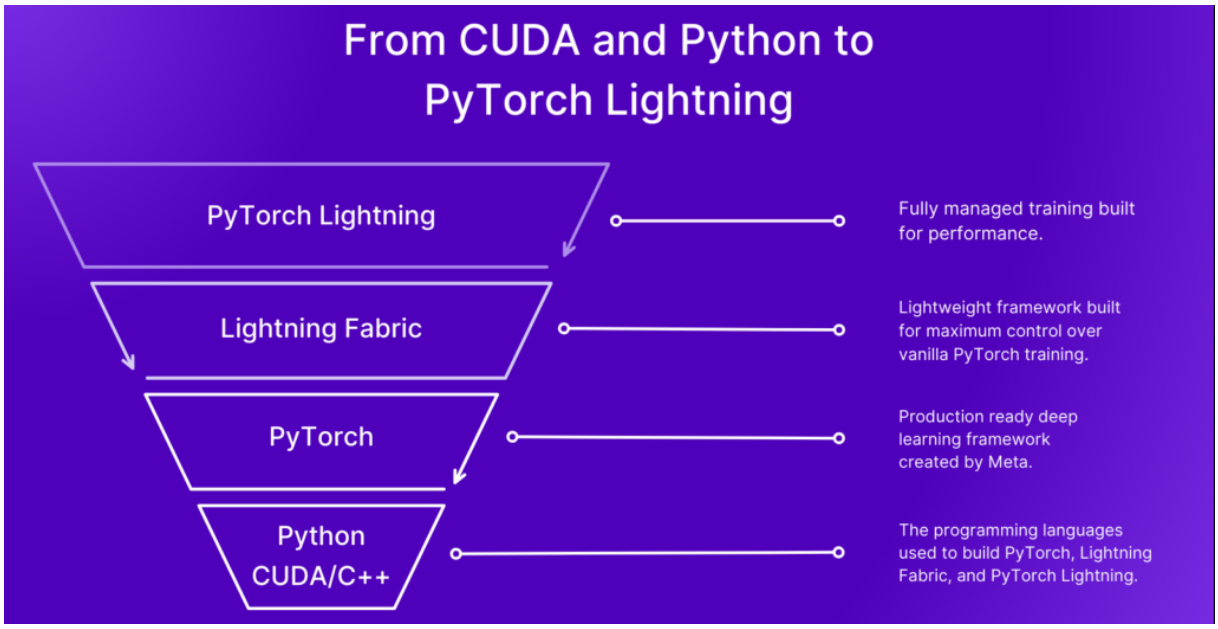


Figura 3.3: Organización en capas sobre las que se construye Lightning

Fuente: Lightning AI

3.4. Bases de datos para estimación de la dirección de la mirada

En los últimos años, diferentes Universidades y grupos de investigación han realizado sus propias bases de datos para crear soluciones para el problema de la estimación de la mirada humana. Las diferentes aproximaciones existentes reflejan no solo una gran diferencia en los modelos neuronales propuestos, si no también en las etapas previas de recolección de datos y pre-procesado de los mismos. Mientras que algunos basan sus hipótesis en el empleo de múltiples cámaras de alta resolución para abarcar mayor rango de posiciones y hacer modelos más robustos, otros *datasets* tienen como filosofía que las imágenes de un único ojo[13] son suficientes para abordar el problema de la dirección de la mirada, o incluso que el entrenamiento del modelo en imágenes de una sola persona en concreto con cámaras de baja resolución[11] pueden conseguir resultados realmente impresionantes.

3.4.1. End-to-end Video-Based Eye-tracking (EVE)

EVE [12] es un *dataset* creado por un equipo de investigadores de la Escuela Politécnica Federal de Zurich (ETH) para la estimación de la dirección de la mirada. Sus datos han sido recolectados a partir de una muestra de 54 participantes y 4 puntos de vista distintos, formando así una base de datos de más de 12 millones de *frames*, 105 horas de vídeo y 1327 estímulos visuales únicos a partir de imágenes, vídeos y texto. Además, la diversidad de género, edad, etnia y ausencia o presencia de gafas en los participantes aporta variedad al conjunto de datos.

En su desarrollo, se toma ventaja de la interacción y relación existente entre el movimiento de los ojos y el qué se está mirando. Así se mejora significativamente la precisión de la estimación de la dirección de la mirada para etiquetar las imágenes. Para ello, se introduce el concepto de *Point-of-Gaze (PoG)*, el punto exacto al que se está mirando una persona definido como medida en la pantalla plana en unidades métricas o píxeles.

Durante la fase de recolección de muestras, se presentan a cada uno de los participantes una serie de estímulos visuales a través de una pantalla de alta resolución (1920x1080). En concreto, se presentan tres clases diferentes de estímulos: imágenes, vídeo y páginas de wikipedia. Cada uno de estos estímulos será reproducido por un tiempo determinado, y el participante sera libre de fijarse en los puntos más llamativos

de la escena y, en el caso de las páginas de wikipedia, navegar libremente por la entrada y saltar a otras páginas de wikipedia. La anotación fiable o *ground truth* de cada muestra se genera a través de un *Eye Tracker* comercial, en concreto del Tobii Eye Tracker.

Con el objetivo de aportar mayor robustez al modelo, se emplean 4 cámaras en el set de grabación. 3 de ellas se sitúan encima de la pantalla y trabajan a 30 fotogramas por segundo (FPS), mientras que una cuarta cámara con obturador global *global shutter* se coloca por debajo de la pantalla de reproducción y es capaz de capturar imágenes a una frecuencia de 60 *fps*. Además, los reflejos producidos por las gafas o la luz infrarroja del *Eye Tracker* son eliminados de las cámaras empleando un filtro óptico de paso de banda.

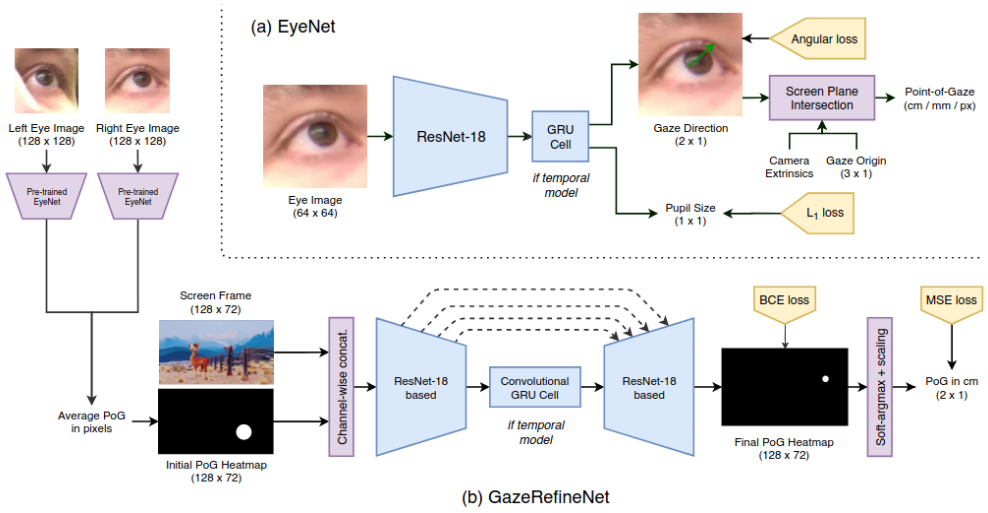


Figura 3.4: Arquitectura de los modelos EyeNet (figura a) y GazeRefineNet (figura b) de EVE

EVE configura su modelo **EyeNet** a partir de una ResNet-18, como se puede observar en la figura 3.4. Tomando como entrada las imágenes recortadas de cada uno de los ojos, produce en su salida la estimación del tamaño de la pupila y el vector de estimación de la mirada, el cual se combina con las coordenadas del centro del ojo y los parámetros de la cámara para obtener el *Point of Gaze* en la pantalla de visualización. Además del EyeNet, también se presenta el modelo **GazeRefineNet**, que toma directamente las estimaciones del PoG producidas por el **EyeNet** y las compara con el PoG verdadero tomando la información de la pantalla para refinar la

estimación y mejorar las predicciones del punto de mira del usuario en pantalla en un 28 %.

3.4.2. ETH-XGaze

Creado también por la Escuela Politécnica Federal de Zurich, ETH-XGaze [17] está compuesto por más de un millón de imágenes de alta resolución tomadas de 110 participantes de diversidad racial y cultural mediante el empleo de 18 cámaras digitales SLR y condiciones de luminosidad excelentes. XGaze tiene en consideración un mayor número de píxeles en la región peri-ocular comparado con el resto de *datasets* existentes.

En el *set* de recolección de datos, 5 pares de cámaras se emplean para capturar la información geométrica, mientras que otras 8 cámaras se emplean para captar la textura de la imagen. A ambos lados de la pantalla de visualización, diferentes cajas de luz regulan la iluminación en la escena, creando diferentes casos de luminosidad. En frente de ellos se aplican filtros de polarización para evitar los reflejos en la escena. A diferencia de EVE *dataset*, los participantes en este modelo deben concentrase en un círculo que aparece en una posición aleatoria en la pantalla de visualización para guardar las muestras. Dicho círculo se va haciendo cada vez más pequeño, hasta que se convierte en un punto. En ese preciso momento, el participante debe pulsar con el ratón en el para guardar la toma. De esta forma, el método asegura que el *ground-truth* empleado es completamente verídico, dado que el participante está obligado a mirar al punto para no fallar al pulsar en él.

El preprocesado de los datos es análogo al caso de EVE, pues involucra un proceso de recorte del área facial fuera de la imagen original como entrada para la red empleada en el entrenamiento. Por cada entrada, se aplica una detección de *landmarks* del rostro usando métodos del estado del arte. A continuación, se ajusta un modelo deformable 3D (3DMM) de la cara a los *landmarks* detectados, lo que permite estimar la orientación de la cabeza en el espacio tridimensional. Esto último, junto con la información de calibración y parámetros de la cámara, se utiliza para normalizar los datos. Durante este proceso de normalización, se define el centro de la cara como el centro de una región formada por las 4 esquinas de los ojos y 2 de la nariz.

A diferencia de EVE y la gran mayoría de bases de datos para estimación de la mirada, ETH-XGaze no incluye un conjunto de validación. La ausencia de este tipo de conjunto se compensa con diferentes métodos de evaluación del modelo que permiten

comprobar los resultados y eficiencia del *dataset* de forma robusta:

- Cross-Data Evaluation: En este tipo de evaluación, el modelo se entrena en ETH-XGaze y luego se evalúa en otra base de datos y viceversa. Debido al empleo de cámaras con mayor resolución, el entrenamiento de la red en otra base de datos y su posterior evaluación en ETH-XGaze arroja grandes errores.

- Within-Dataset Evaluation: Consiste en entrenar el modelo en el conjunto de entrenamiento y se prueba en el conjunto de Test.

- Person Specific Test: De forma análoga a los modelos basados en personas específicas, esta evaluación consiste entrenar el modelo en el conjunto de *Train* y, a continuación, se realiza un *fine-tunning* en el conjunto especial de este dataset llamado *Person-Specific Test* junto con 200 muestras de calibraciones de personas específicas, mejorando significativamente los resultados en la predicción sobre estas mismas personas. No obstante, este tipo de evaluación debe tener un balance entre conseguir buenos resultados y emplear la menor cantidad de muestras de calibración posible.

Capítulo 4
Diseño y desarrollo del modelo

El diseño de una red neuronal es un proceso multi-facético que implica varias fases críticas para garantizar la efectividad del modelo final. Este capítulo se divide en las diferentes fases de diseño que están involucradas en este trabajo, desde la preparación y organización de los datos de entrada para ser utilizados de manera eficiente en el entrenamiento, pasando por la construcción y arquitectura del modelo, finalizando con la búsqueda de los hiper-parámetros y parámetros ideales que permiten obtener resultados ilustrativos para una correcta interpretación y análisis de los mismos.

4.1. Implementación de la carga de datos

Durante esta primera etapa de diseño, se preparan y organizan los datos de entrada para ser utilizados de manera eficiente en el entrenamiento. Para ello se debe crear un componente que se encargue de todo ello. A este componente se le conoce en PyTorch como *Dataloader* (Cargador de datos).

Sus principales tareas consistirán en realizar una lectura del directorio que contiene la base de datos o *dataset*, preprocesarlos (esto incluye operaciones como normalización y aumento de datos) y organizarlos para optimizar el rendimiento y eficiencia del entrenamiento del modelo. Además, gestiona la aleatorización y mezcla de los datos para evitar el sobre-ajuste y mejorar la generalización del modelo.

Tras realizar un estudio detenido de las diferentes bases de datos públicas de modelos orientados a resolver el problema de la dirección de la mirada, se ha decidido tomar como referencia la base de datos de *EVE dataset*, debido principalmente a los siguientes factores:

- Los vídeos de entrada de las diferentes cámaras empleadas para captar los estímulos visuales están parcialmente pre-procesados. Se les ha eliminado la

distorsión de la cámara mediante un proceso de calibración con OpenCV, ahorrando operaciones adicionales durante la carga de datos.

- La diversidad racial y étnica en los participantes que componen los datos, aporta robustez al modelo entrenado. Sin embargo, la ausencia o presencia de gafas o lentes de contacto hará que el sistema no funcione bien en su presencia.
- El empleo de múltiples cámaras y datos de la pantalla de visualización permite tener una mayor cobertura en la rotación de la cabeza y mayores rangos para la detección de los rayos de dirección de la mirada en el espacio tridimensional.
- La presencia en los datos de entrada de información relacionada a la pantalla de visualización permitirá las posteriores etapas de entrenamiento, testeo y evaluación.

4.1.1. Estructura de la base de datos

Como se explicó en el apartado 3.4.1, esta base de datos se compone de 1327 estímulos visuales únicos (lotes de entrada de la red), creados a partir de diferentes muestras de 54 participantes que observan un estímulo visual que puede ser de uno de las siguientes 3 categorías: (a) imagen, (b) vídeo o (c) páginas de wikipedia. 39 participantes han sido seleccionados para conformar el set de entrenamiento, 10 para el set de test y los 5 restantes para el set de evaluación.

Por cada uno de estos participantes, se han tomado entre 80 y 180 muestras (grabaciones cortas, de entre 2 y 10 segundos, en la que se graba desde las 4 cámaras del set de grabación cómo el participante observa un estímulo visual en una pantalla). De media, el 65 % de estas muestras corresponden con estímulos visuales de imágenes, el 30 % de vídeos y el 5 % a páginas de wikipedia.

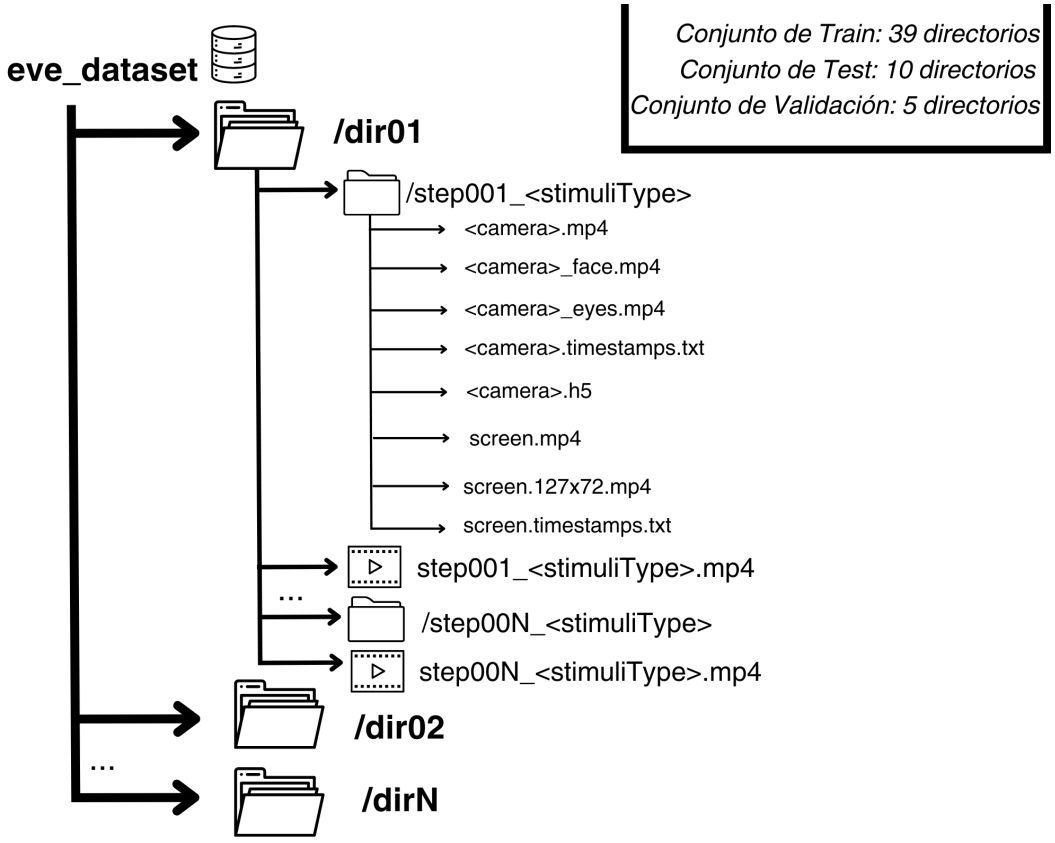


Figura 4.1: Composición del conjunto de train en la base de datos

En la figura 4.1 se muestra la estructura de los sets de entrenamiento, validación y test en el dataset de EVE. Todos siguen la misma jerarquía, siendo la única diferencia notable que los sets de entrenamiento y validación tienen vídeos que muestra el punto de la dirección de la mirada verdadero en pantalla (es decir, son muestras etiquetadas).

Por otro lado, vemos que en cada carpeta de estímulos se tienen los siguientes datos:

- Vídeos de las cámaras: archivos .mp4 de vídeo de la grabación de las cuatro cámaras (izquierda, derecha, central y *basler*). Por cada una de ellas, se observa que hay 3 vídeos por cada una de ellas: vídeo completo, de la zona facial y de la zona ocular.

- Vídeos de la pantalla: un archivo .mp4 a resolución completa y otro recortado a 127x72 píxeles.

- Fichero plano con marcas de tiempo de las cámaras: en las que aparece, por cada línea de fichero, la marca de tiempo en la que se tomo el frame correspondiente.

- Fichero con marcas de tiempo de la pantalla: Igual que el anterior, pero para la pantalla
- Fichero con marcas de tiempo del ratón: por cada línea, 3 datos separados por tabulaciones: marca de tiempo y posición del ratón (coordenadas x e y en píxeles) en cada uno de los frames.
- Archivo de datos en formato jerárquico .h5: Contiene, por cada cámara, valores intermedios del pre-procesamiento y las etiquetas del *ground-truth* asociadas con los archivos mencionados anteriormente.

4.1.2. Procesamiento de vídeos

Debido a la gran cantidad de vídeos que contiene cada estímulo visual a procesar, era necesario crear un componente capaz de manejarlos. Para ello, se creó una clase **VideoProccesor** que gestionará una cámara concreta de cada uno de los estímulos a procesar. Por cada uno de ellos guardará la duración del mismo, el número total de fotogramas y los fotogramas por segundo (FPS, *frames per second*). También permitirá extraer uno a uno los frames de cada vídeo para poder realizar un análisis y procesado más profundo de los datos, modificar su apariencia o dibujar sobre ellos. Esto se consigue empleando la clase **VideoCapture** de la librería de OpenCV.

Estas dos últimas utilidades mencionadas crea la necesidad de poder recomponer los frames modificados de nuevo en un vídeo con las mismas características que el original. Para ello, se emplea la librería de **FFmpeg** para Python, una herramienta para manipulación de archivos multimedia como se muestra en el listado de código 4.1:

```
def createVideo(frames, fps: int, output: str):
"""
Create video stacking frames from the frameList.
The output video will be stored in output path.
"""
if frames == []:
    return None

height, width, _ = frames[0].shape

out = (
    ffmpeg.input('pipe:', format='rawvideo', pix_fmt='bgr24',
        s=f'{width}x{height}')
    .output(output, pix_fmt='yuv420p', vcodec='libx264', r=fps)
    .overwrite_output()
    .global_args('-loglevel', 'panic') # set panic for suppressing all
        logs
    .run_async(pipe_stdin=True)
)

for frame in frames:
    out.stdin.write(frame.astype(np.uint8).tobytes())

out.stdin.close()
out.wait()
```

Código 4.1: Función para recomponer frames

4.1.3. Carga y procesado de datos

Para la creación del Dataloader se ha empleado la clase DataLoader de PyTorch, una herramienta que facilita la carga y el manejo de datos para el entrenamiento de modelos de ML. Es especialmente útil para trabajar con grandes conjuntos de datos, ya que gestiona la lectura de los datos en lotes (batches), la mezcla (shuffling) y otras operaciones comunes en el pre-procesamiento de datos. Internamente, representa un iterador en Python sobre un dataset.

En el fragmento de código 4.2 se detalla la estructura que sigue la clase. Indicando el argumento Dataset en el constructor de la clase, se especifica un objeto dataset de donde se extraerán los datos.

```
class CustomDataloader(Dataset):
def __init__(self, *args, **kwargs):
    # Dataloader initialitation (e.g: load data from files)

def __len__(self):
    # Return dataset length

def __getitem__(self, idx):
    # Return element (data and labels) from dataset given a specific
       index
```

Código 4.2: Estructura básica de un Dataloader personalizado

Dado que vamos a tratar con 3 tipos de conjuntos diferentes (entrenamiento, test y validación), se ha decidido crear una clase hija para cada una de ellas que hereda de la clase padre (principal) del Dataloader. De este modo, se podrá atender a las pequeñas diferencias entre cada una de ellas. Además, se define un archivo **yml** de configuración en el que se especificará una configuración especifica para cada componente, siendo cargados mediante la librería yml para Python.

En el constructor de la clase del cargador de datos se tomará dos únicos argumentos:

- Configuración del Dataloader: con la ruta del dataset, las cámaras a emplear, el tipo de imagen que se querrá tomar de las cámaras (zona ocular, facial o completa), etc.
- Configuración del Conjunto de Entrenamiento/Test/Validación: nombre que tiene dicho conjunto en la base de datos, número de participantes a tomar, que clases de estímulos emplear.

Con esta información, el constructor accederá a la ruta donde está el dataset e irá procesando carpeta por carpeta del conjunto, estímulo por estímulo, los datos de entrada. Nótese que el dataset en crudo es muy voluminoso, ocupando más de 500GB de memoria de almacenamiento. Esto significa que la carga de datos inicial al ejecutar el programa puede ser tardía. Para optimizar esta fase y hacerla lo más veloz posible, se emplea la librería **concurrent features**[1], que permita crear hilos *threads* de ejecución, de modo que cada uno de ellos pueda procesar un directorio de un estímulo concreto (un lote concreto de entrada), acelerando la carga de datos a memoria en el momento de ejecutar nuestra red. En la tabla 4.1 se muestran los tiempos promedios de carga

[1] `https://docs.python.org/3/library/concurrent.futures.html`

por cada uno de los conjuntos y sus diferencias entre hacerlo de manera secuencial y paralela. El promedio ha sido realizado a partir de 10 ejecuciones del dataloader para cada uno de los conjuntos que se muestran a continuación, empleando una configuración en la que se toman todas las muestras de todos los participantes (incluyendo las cuatro cámaras disponibles y los tres tipos de estímulos) y un total de 30 frames por cada vídeo:

	Training set	**Test set**	**Validation set**
Carga secuencial	5m14s	3m27s	4m46s
Carga paralela	1m30s	10.02s	45.06s

Cuadro 4.1: Tiempos promedios de carga a memoria de los datos de entrada

Si bien se puede apreciar una mejora significativa, siguen siendo tiempos relativamente altos, sobre todo para el set de entrenamiento y el de validación. Como hemos mencionado anteriormente, la carga inicial de datos conlleva conocer ciertas especificaciones del archivo de configuración. La posibilidad de poder guardar los datos cargados con una configuración concreta en una ejecución anterior para usarla posteriormente en otra que ejecute con la misma configuración permitiría ahorrar aún más tiempo en este proceso. Esto es lo que comúnmente conocemos como **caché**. Para implementarla, se ha empleado la librería de **pickle**.

A la hora de empezar a ejecutar el programa, el dataloader tomará el archivo de configuración referente a él y analizará que ratio de fotogramas por vídeo se está empleando y que conjunto de datos se quiere cargar (entrenamiento, test o validación). Si existe en el directorio de caché (también especificado en el archivo de configuración, por defecto **.cache**) un archivo *.pkl* que se haya generado con esa configuración, el dataloader cargará directamente ese archivo. En caso contrario, construirá la caché para esa configuración, que será formada por una lista de lotes de entrada que representan a cada directorio de estímulo para ese conjunto específico. Los elementos de esta lista serán listas de diccionarios que contendrán dentro la siguiente información

- Cámara $\in$ [izquierda, derecha, centro, basler, pantalla]
- Nombre del estímulo
- ID del participante al que pertenece ese estímulo (esto es, nombre del directorio que contiene el estímulo)
- Ruta absoluta al directorio del estímulo

- Lista con los índices de los fotogramas a emplear del vídeo de la cámara que se está procesando.

De esta forma, a partir de ahora nuestros estímulos de entrada o lotes tendrán en cuenta una cámara específica y no todas las contenidas en el directorio de ese estímulo concreto. Por otro lado, guardar los índices de los fotogramas de los vídeos que posteriormente se procesaran y no los fotogramas como tal permiten mantener una caché relativamente pequeña, buscando un balance entre rapidez en la fase de carga y mantener las caché lo más pequeñas posibles. En la tabla 4.2 se muestran los tiempos promedios de carga de las caché para los 3 sets, con una configuración de 30 frames por vídeo, junto con el espacio que ocupan en memoria. Los tiempos de carga especificados han sido obtenidos a partir del promedio de 10 ejecuciones:

	Training set	**Test set**	**Validation set**
Tiempo de carga	1.4s	1.2s	1.3s
Tamaño de la caché en disco	2.5M	660K	288K

Cuadro 4.2: Tiempos de carga de los datos desde caché y espacio en disco que ocupan.

Dado que ya hemos encontrado una forma eficiente de cargar los datos, falta lidiar con su pre-procesamiento. Cuando el dataloader escoja el estímulo i-ésimo de los cargados anteriormente con nombre Dataloader[i], lo que estará haciendo es llamar a la función __getitem__ de la clase pasando el índice i. El dataloader tomará la información de ese estimulo especificada arriba y leerá el fichero *Herarchical Data Format (HDF)* de la cámara que se está empleando (exceptuando los casos en los que la cámara que se emplee sea la pantalla), guardando dichos datos en un diccionario. Esto se realiza gracias a la librería **h5py**. Una vez leído, el diccionario resultante contendrá valores intermedios relativos a la información espacial y *ground-truth* de las muestras. Este diccionario se complementará con la información que guardábamos en caché para tener la información completa de cada muestra a emplear. Será en este paso cuando obtengamos las imágenes de los vídeos utilizando los índices que habíamos guardado en caché y la clase **VideoProcessor**.

El tamaño y zona de interés de la imagen que se guardará dependerá de la especificada en configuración, de modo que podamos usar las imágenes completas para realizar tests mientras que en el entrenamiento podamos emplear las imágenes recortadas de los ojos o de la zona facial. Se realizará un pre-procesamiento de las imágenes extraídas, que constará de una transformación de la iluminación de la imagen

variando de forma aleatoria su valor, saturación y tono, pasando previamente la imagen a formato HSV (Hue-Saturation-Value) y devolviéndola de nuevo en BGR, seguido de una operación de intercambio de los canales al formato habitual de PyTorch [N,C,H,W], cambiar el tipo de los píxeles a coma flotante y normalizar los colores en un rango dado. Para aplicar el normalizado, primero dividimos las componentes de cada color entre el máximo de la imagen para conseguir valores entre 0 y 1:

$$\begin{bmatrix} B & G & R \end{bmatrix}_{[0,1]} = \begin{bmatrix} \frac{B}{B_{\text{máx}}} & \frac{G}{G_{\text{máx}}} & \frac{R}{R_{\text{máx}}} \end{bmatrix} \tag{4.1}$$

A continuación, se cambia aplica la siguiente fórmula para normalizarlo entre un rango concreto de valores A y B, para A < B:

$$\begin{bmatrix} B & G & R \end{bmatrix}_{[A,B]} = \begin{bmatrix} B_{min} & G_{min} & R_{min} \end{bmatrix} + (\begin{bmatrix} B & G & R \end{bmatrix}_{[0,1]} * (B - A)) \tag{4.2}$$

Por defecto, se han normalizado las imágenes de las cuatro cámaras principales entre 0 y 1, mientras que las imágenes de la pantalla se han normalizado entre -1 y 1.

Finalmente, el resto de datos del diccionario que estén en formato de numpy array serán transformados a tensores de PyTorch, y se devolverá dicho diccionario en la función.

Dado que estamos trabajando con lotes que pueden contener cada uno una cantidad grande de fotogramas dependiendo del estímulo concreto (duración del video de entrada), era crucial asegurar, además de una carga rápida de datos, asegurarse de tener una optimización correcta en el método de __getitem__ que se llama cada vez que queremos obtener un lote específico de nuestra clase Dataloader. Si bien Python cuenta con un mecanismo de recolección de basura (*Garbage Collector*) cíclico, este no actúa de forma inmediata cada vez que se elimina la referencia a un objeto empleando *del* ni cuando se iguala a *None*. No obstante, podemos forzar una recolección manual utilizando el módulo **gc** y llamando a *gc.collect()*.

Algo parecido ocurre con cómo gestiona la memoria PyTorch. Cuando el objeto Tensor ya no tiene referencias activas, PyTorch libera la memoria ocupada por dicho objeto. Sin embargo, la memoria no es devuelta inmediatamente al sistema operativo o a otros procesos en la GPU, sin o que guarda en caché la memoria liberada para que pueda

ser reutilizada en posteriores operaciones (es decir, se retiene internamente la memoria). Podemos marcar ciertos bloques de memoria como disponibles para ser reutilizados por PyTorch (y sin ser devuelta al sistema operativo) de forma explícita. Para ello, se emplea la llamada a *torch.cuda.empty_cache()* para liberar memoria disponible de la GPU sin interrumpir el flujo del programa.

La combinación de la librería *gc* y *torch.cuda.empty_cache()* nos permite poder gestionar de manera manual y óptima la memoria del programa. Para ello, se ha creado una función útil para dicho propósito: cuando se creen tensores dentro de funciones y se quieran eliminar una vez ya no vayan a ser empleados sin tener que esperar a salir de la función y esperar a que Python y PyTorch se encarguen de liberar la memoria, llamaremos a **SalvageMemory()** (ver listado de código 4.3).

```
def salvageMemory():
"""
Clean up all posible resources using Garbage Collection and freeing
posible cache memory.
"""
torch.cuda.empty_cache()
gc.collect()
```

Código 4.3: Función para liberar todos los posibles recursos y memoria caché

Este método será esencial en las posteriores etapas de diseño de la arquitectura del modelo para poder ampliar en la medida de lo posible la cantidad máxima de tamaño de lotes que podrá soportar la entrada del mismo durante el entrenamiento.

4.2. Visualización de datos

Este apartado aborda las operaciones matemáticas y transformaciones espaciales necesarias para poder traducir los datos contenidos en los archivos HDF a representaciones visuales que permitan, tanto al usuario final como al programador del modelo, detectar de forma eficiente y rápida los resultados del modelo una vez ha sido entrenado, o de los propios datos del ground-truth.

La estimación de la dirección de la mirada es un problema que ha de ser abordado en diferentes puntos de vista. En el problema que pretendemos abordar, podemos describir la dirección de la mirada como un vector en el espacio tridimensional, que se origina en el centro de la pupila del ojo y se dirige hacia el punto al cual se está observando.

Visualmente puede interpretarse como dos haces que se originan en los ojos y que interseccionan con un plano que es perpendicular a dos de los ejes. Este plano es la pantalla de visualización de los datos.

Cuando los haces intersecan en él, se obtienen un par de puntos en un espacio bidimensional referente al plano imagen o pantalla en unidades métricas o píxeles. Esta definición corresponde con el concepto de *Point Of Gaze* introducido en el apartado 3.4.1. Se puede observar entonces el manejo de dos sistemas de referencias distintos, y por ende la necesidad de buscar una traducción que nos permita encontrar la correspondencia de un punto en el espacio 3D de la cámara en el espacio 2D de la imagen. Para ello, es imprescindible entender la generación de la imagen (ver figura 4.2).

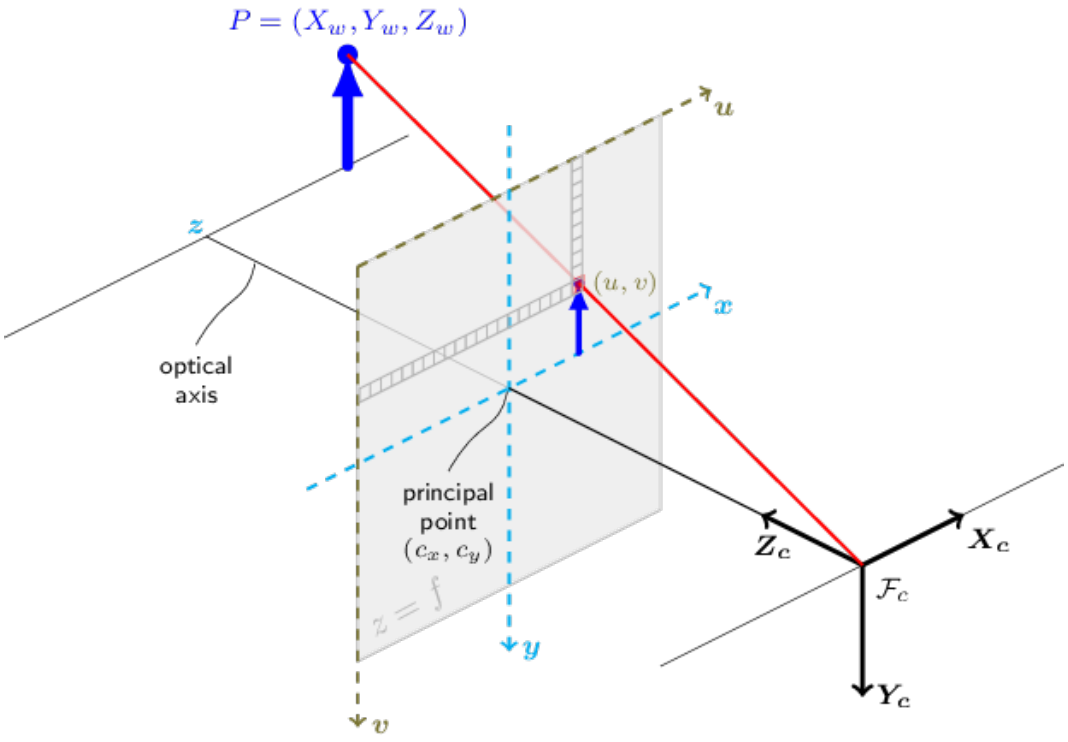


Figura 4.2: Formación de la imagen en el modelo pinhole

La traducción de un punto del espacio 3D de la cámara al espacio 2D de la imagen implica comprender el modelo pinhole. En este modelo simplificado la luz entrante a través del pequeño agujero (estenopeico) de la cámara converge en un punto concreto denominado centro óptico. La distancia desde este punto hasta el plano de la imagen se conoce como distancia focal. Estas dos características vinculadas

a la formación de la imagen se conocen como **parámetros intrínsecos**, y puede representarse matricialmente cómo:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{4.3}$$

Si bien el sistema de referencia 3D de la cámara se define con respecto a ella misma, con centro en su centro óptico, el sistema de referencia 3D global o de la escena es el marco en el cual se definen las posiciones y orientaciones de todos los objetos y la escena concreta, y se fija de manera arbitraria en el contexto de la aplicación. En este nuevo sistema de referencia, la cámara está definida por una serie de parámetros que describen su orientación y traslación. Son los que se denominan **parámetros extrínsecos**, y se representan, respectivamente, mediante una matriz de transformación y un vector de traslación.

Por simplificación, podemos representar conjuntamente la rotación y traslación de un punto en el espacio 3D empleando coordenadas homogéneas, (x', y', z', w), donde w es el factor de escala adicional utilizado. Combinando la información de los parámetros intrínsecos y extrínsecos, podemos formar una nueva matriz, la **matriz de transformación homogéneas**, para representar la transformación del vector en estas coordenadas a otro sistema de referencia:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{4.4}$$

Con toda esta información, ya es posible conseguir el propósito que planteamos en este apartado. Los archivos HDF presentes en cada uno de los estímulos empleados como entrada de la red contienen toda esta información, así como las coordenadas de los centros de los ojos en el sistema de referencia de la cámara o los factores de escala para pasar de milímetros a píxeles y viceversa en el sistema de referencia de la pantalla, entre otros.

Primeramente, se definen las funciones necesarias para realizar las transformaciones entre diferentes sistemas de referencia en base a las siguientes fórmulas. Dado que los centros de los ojos contenidos en el HDF vienen en el sistema de referencia de la cámara (3D), es necesario proyectarlos sobre la imagen 2D para poder dibujarlos sobre la predicción:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_c} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{4.5}$$

Adicionalmente, el HDF de las cámaras contiene el *Point of Gaze* en pantalla de cada uno de los ojos. Esto permite dibujar directamente en la imagen de la pantalla los puntos de mira izquierda y derecho, que generalmente serán muy cercanos, dado que el usuario suele centrar la vista en un único punto con los ojos (ver figura 4.3).

Figura 4.3: Punto de la dirección de la mirada en pantalla

Aún más relevante es el hecho de poder dibujar la dirección de la mirada en los ojos del usuario. Nótese que el HDF contiene, definido por coordenadas esféricas (r, θ, ϕ), donde r es la semirrecta radial partiendo del origen de coordenadas, θ la colatitud y ϕ el azimutal (ver figura 4.4).

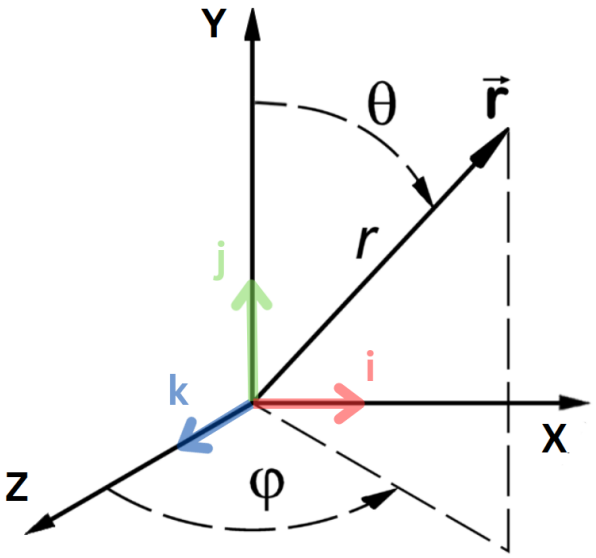


Figura 4.4: Vector representado en coordenadas esféricas

Este vector define el eje Z del sistema de referencia que él mismo representa, con origen $(0, 0, 0)$ en el centro de la pupila del ojo. Podemos obtener los ejes X e Y ortogonales a él para dibujar completamente dicho sistema de referencia con los tres ejes ortogonales, de modo que permita visualizar e interpretar de una forma más completa y fácil las predicciones y el rayo de dirección de la mirada. Así pues, se desprende fácilmente de la figura que podemos obtener la dirección del eje Y a partir del ángulo pitch (θ). Obtenemos el vector que define el plano XY perpendicular a Z, definido por $(-\sin(\phi), \cos(\phi), 0)$. A partir de este, podemos hallar el vector unitario del eje Y, $\mathbf{j}$, dividiendo el vector anterior por su módulo. Dado que $\|\mathbf{v}\| = \sqrt{(-\sin(\phi))^2 + (\cos(\phi))^2 + 0^2} = \sqrt{\sin^2(\phi) + \cos^2(\phi)} = \sqrt{1} = 1$, tenemos que:

$$\mathbf{j} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = (-\sin(\phi), \cos(\phi), 0) \tag{4.6}$$

Recordamos además que tenemos las coordenadas esféricas del eje Z (r, θ, ϕ). A partir de ellas, podemos hallar su vector unitario $\mathbf{k}$ considerando r=1. Para ello, implementamos una función que realiza la transformación de coordenadas, mostrada en el listado de código 4.4.

```
def toCartesian(pitchyaw, r = 1):
"""
Convert spherical coordinates to cartesian coordinates
where yaw=theta and pitch=phi and r is the radius.
"""
pitch, yaw = pitchyaw
x = math.cos(pitch) * math.sin(yaw)
y = math.sin(pitch)
z = math.cos(pitch) * math.cos(yaw)

return (x, y, z)
```

Código 4.4: Función para transformar coordenas esféricas en cartesianas

Finalmente, a partir de los dos vectores unitarios de los ejes Y,Z, podemos obtener el vector unitario del eje restante X a partir del producto vectorial de ambos. No obstante, debemos tener en cuenta la orientación del sistema de referencia de la cámara: El eje Z es la profundidad, y dado que la cámara siempre se va a encontrar en el plano de la pantalla de visualización (plano imagen), tenemos que está en z=0 y apunta hacia fuera. Si lo que queremos es dibujar el eje Z en el ojo del participante, apuntando hacia la imagen, debemos rotar 180 grados el sistema de referencia para que se dirija correctamente hacia la cámara.

Ahora sí, se puede aplicar el producto vectorial de Gibbs $\mathbf{j} \times \mathbf{k} = \mathbf{i}$ para obtener el vector unitario del eje X. Obtenidos los 3 vectores unitarios, ya podemos dibujarlos en la imagen. Los sumaremos al punto que define el centro del ojo en la imagen, multiplicado por una distancia arbitraria α definida por el usuario, y establecida por defecto a 50 para que los ejes sean de un tamaño considerable, suficientemente grande como para visualizarlo bien y no excesivamente grande que pueda desdibujar la imagen:

Figura 4.5: Dirección de la mirada representada en los ojos de un participante

De esta forma, queda abordado el problema de la representación de la dirección de la mirada para las futuras predicciones de la red e interpretación de resultados.

4.3. Arquitectura e implementación del modelo

Las decisiones sobre el diseño y estructura de la red constituyen la fase central del desarrollo del modelo completo. En esta etapa, se establece la arquitectura del modelo, lo cual incluye la selección del tipo de red, las características que debe tener la entrada y el número de clases que se desean tener en la salida, así como las métricas a definir que nos ayuden a interpretar los resultados. A ello se suma la búsqueda de los mejores hiper-parámetros, como son la tasa de aprendizaje, el tamaño del batch o la configuración del optimizador. En definitiva, esta etapa resulta crucial, ya que determina la capacidad del modelo para aprender y generalizar a partir de los datos.

La primera cuestión a abordar antes de empezar a diseñar la arquitectura de la red fue la posibilidad de fusionar este modelo con otro que estime otros parámetros del rostro diferentes pero que estén ligados por su naturaleza. Esto aportaría, por un lado, la posibilidad de aumentar la funcionalidad de un modelo ya existente que trabaje con

imágenes de caras para que pueda estimar la dirección de la mirada humana. Por otro lado, plantearía la hipótesis de estudiar la viabilidad de la coexistencia de la estimación conjunta de ambos dominios y su posible aumento de aplicaciones al abarcar más ámbitos. Dadas las características de las imágenes en la base datos a emplear, se planteo la posibilidad de fusionarlo con algún modelo existente de estimación de la orientación de la cabeza. Esto inicialmente tenía ciertas desventajas, y era que la base de datos si tiene variedad de ángulos de la dirección de la mirada pero no posee fotogramas muy variados en cuanto a rotaciones de la cabeza del participante refiere. Además, el croma verde empleado de fondo para las imágenes podría "engañar" al modelo durante su entrenamiento y podría tomarlo como una característica que realmente no esta relacionada. No obstante, también se contaba con dos motivos muy importantes:

1. El campo de la estimación de la orientación de la cabeza es un área de investigación asentado, con modelos públicos y que trabaja directamente con imágenes del rostro.

2. Las imágenes de EVE están tomadas desde 4 ángulos distintos, siendo estos 4 puntos de vista correspondientes a la misma orientación en un instante de tiempo dado. Esto es un punto fundamental que aumenta la robustez y la posibilidad de obtener estimaciones generalizadas. Esto en cierta manera podría difuminar el problema relacionado con que la red no tenga tantas orientaciones de la cabeza como otras bases de datos.

Debido a que el problema del fondo podría ser solucionado mediante Data-Augmentation, modificándolo para conseguir variedades distintas de entornos, se decidió fusionar ambas tareas y partir de un modelo de estimación de la orientación de la cabeza.

El Laboratorio de Percepción Computacional y Robótica (PCR) de la Universidad Politécnica de Madrid puso a mi disposición los siguientes dos repositorios funcionales:

- pcr_upm/image_frameworks: es un framework para imágenes, anotaciones, visualización y otros submódulos para la detección automática de caras o landmarks generales. El repositorio contiene además funciones básicas y clases generales encargadas de cargar datos.

- pcr_upm/students_headpose: es un framework que implementa en PyTorch Lightning el entrenamiento y la evaluación de un estimador de la orientación

de la cabeza con respecto a la cámara, implementado como submódulo dentro del propio image_frameworks. Contiene diferentes tipos de redes y clases para transformaciones de datos.

De esta manera, el objetivo principal era modificar students_headpose para añadir la estimación de la dirección de la mirada, empleando las herramientas que ofrecía images_framework para ofrecer una nueva variante del repositorio que siga, en la medida de lo posible, la misma estructura e implementación de funciones. El repositorio de students_headpose contenía un script para entrenar el modelo de orientación en la cabeza, otro para test empleando imágenes, vídeos o la cámara del portátil y un último script para procesar la base de datos empleando las etiquetas del dataset. El framework soporta una gran variedad de bases de datos, entre las que se pueden destacar Annotated Facial Landmarks in the Wild (AFLW)[9], 300wLP[21] o DAD-3DHeads [10].

A pesar de soportar diferentes bases de datos, todas contenían sus anotaciones en ficheros CSV y cargaban las imágenes de entrada directamente desde la ruta donde se encuentran contenidas en el dataset. En nuestra base de datos (EVE), se tienen que tomar el número de fotogramas que se quiere de cada uno de los vídeos del dataset y las anotaciones están compactadas en los archivos en estructura jerárquica HDF. Dado que guardar los frames en disco temporalmente y las anotaciones en archivos de texto sería ineficiente al aumentar el tiempo de carga y el consumo de espacio en disco, se incrusto directamente el dataloader diseñado en el apartado 4.1.3 y se realizaron las modificaciones pertinentes para poder tomar directamente los datos tal y como vienen en el dataset.

Con solo este par de cambios ya era posible modificar el script de procesamiento de nuestro Dataset, empleando las técnicas de dibujado y visualización explicadas en la sección 4.2. Dado que la información de los HDF del dataset ya contiene la orientación de la cabeza en el mismo formato que la dirección de la mirada y el centro de la cabeza en el espacio 3D, se siguieron los mismos pasos para transformar las coordenadas 3D del centro de la cabeza a 2D en la imagen y dibujar la dirección de la cabeza del mismo modo que hicimos con la mirada. Además, se empleo el concepto de **distancia interocular**, distancia euclídea entre los centros de los ojos en unidades métricas en pantalla, para establecer empíricamente un factor de escala en ancho y largo para todas las imágenes del dataset y calcular la *bounding box* de la cabeza del usuario en las imágenes en pantalla completa. Se establecieron dichos factores de escala a 1.2 y

1.5 para ancho y largo, respectivamente, y se empleo los píxeles como unidad métrica.

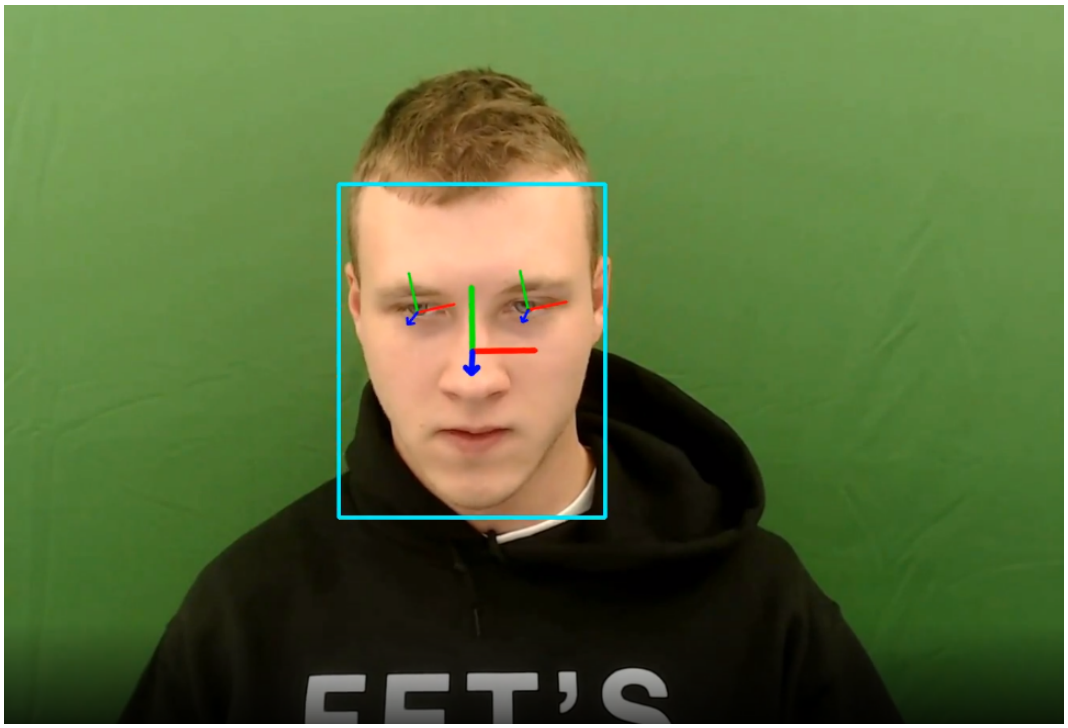


Figura 4.6: Representación de la dirección de la mirada, orientación de la cabeza y bounding box del rostro facial

4.3.1. Implementación del modelo ResNet

Students Headpose Había sido diseñado para poder operar empleando una red neuronal residual con distintos números de capas: ResNet18, ResNet34, ResNet50, ResNet101 y ResNet152. Como se ha descrito en secciones anteriores, es el modelo más recurrente empleando en aplicaciones de estimación de la dirección de la mirada y orientación de la cabeza, por lo que parecía racional emplear una red residual como base para construir nuestro modelo. La red en concreto se construye directamente empleando la librería de **torchvision**, cuyo constructor toma unos pesos como parámetros para usar unos pesos pre-entrenados u otros. En nuestro caso, emplearemos los pesos por defecto que corresponden a los de ImageNet1K. Esta es una versión específica de ImageNet, un conjunto de datos de más de 14 millones de imágenes que se ha empleado para entrenar y evaluar la arquitectura de ResNet. En concreto, la versión empleada '1K' refiere a que organiza las imágenes en 1000 categorías (o tipos de objetos) diferentes.

ResNet50 espera como entrada una imagen de tamaño 224x224x3, un tamaño

estándar que emplean otros modelos como VGG o Mobilenet. Esto varía ligeramente con las imágenes recortadas del área facial de nuestro conjunto de datos, que tienen un tamaño de 256x256x3. Esto plantea dos enfoques totalmente diferentes a la hora de abordar el diseño de nuestro modelo, pues podríamos tomar por un lado la imagen 256x256 y hacer un recorte centrado para transformarla al tamaño estándar de la ResNet, o emplear directamente la imagen sin recortar, modificando con la entrada del modelo para que soporte esta nueva dimensión.

Enfoque	**Encoder personalizado (256x256)**	**Recorte a 224x224**
Tamaño de imagen inicial	256x256	256x256 (recorte a 224x224)
Proceso inicial	Convolución y MaxPool personalizados	Cropping y uso del pipeline original
Pérdida de información	No se pierde información de la imagen	Pérdida de los bordes (16 píxeles por lado)
Uso del modelo pre-entrenado	Modificación en las primeras capas	Uso directo sin modificar las capas originales
Escalabilidad de características	Características más detalladas	Características a partir de imagen recortada
Impacto en el desempeño	Mejor conservación de detalles, pero ajuste de pesos requerido	Uso directo del modelo pre-entrenado, pérdida de detalles en bordes

Cuadro 4.3: Comparación entre emplear un encoder personalizado o realizar un recorte a 224x224 píxeles en ResNet

Teniendo en cuenta las características expuestas en la tabla 4.3, se decidió seguir el enfoque del encoder para evitar pérdida de información y características que puedan ser útiles a partir de los bordes de la cara, que serían inevitablemente afectados en caso de seguir el enfoque contrario.

```
# Define encoder
self.encoder = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2,
        padding=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
)

# Replace the first conv and maxpool layer with the encoder
self.model.conv1 = nn.Identity() # passthrough layer
self.model.maxpool = nn.Identity() # passthrough layer
```

Código 4.5: Definición del encoder en Python

El encoder definido transforma una imagen de entrada de 256 píxeles de ancho, 256 píxeles de largo con 3 canales (256x256x3) en una representación más compacta. Comienza con una capa de convolución 2D (nn.Conv2d) que tiene 64 canales de salida y se aplica a una ventana de 7x7 píxeles de la imagen (kernel=7x7) lo que permite capturar características locales y detalles importantes. El stride de 2 indica que el kernel se mueve 2 píxeles en cada paso, lo que reduce las dimensiones espaciales de la salida, haciendo que la red sea más eficiente al disminuir el tamaño de las características extraídas. El padding de 3 añade 3 píxeles de bordes alrededor de la imagen, lo que ayuda a preservar la información de los bordes durante la convolución. Le sigue la función de activación ReLU (nn.ReLU()), que introduce no linealidad y mejora la capacidad de aprendizaje de patrones complejos. Finalmente, se aplica una capa de max pooling 2D (nn.MaxPool2d) con un kernel de 3x3, un stride de 2 y padding de 1, que vuelve a reducir el tamaño de las características a la mitad pero aumenta la profundidad (64x64x48), enfatizando las más significativas. Estas configuraciones permiten que mi encoder se adapte perfectamente a las dimensiones de entrada de la red, reemplazando las capas iniciales de la ResNet con capas de identidad para un procesamiento eficiente y específico de las imágenes.

En resumen, la capa de convolución reduce la entrada de [256x256x3] a [127x127x64], la capa de ReLU mantiene las dimensiones pero introduce no linealidad y por último la capa de max pooling reduce de [127x127x64] a [64x64x64], pues sabemos que

$$\text{Salida}_{\text{conv}} = \left(\frac{\text{Tamaño de entrada} + 2 \times \text{Padding} - \text{Tamaño del kernel}}{\text{Stride}} \right) + 1$$

Una vez aplicados los cambios, la ResNet recibirá del encoder la imagen o entrada de [64x64x64] y acabará produciendo un vector de características de [1x1x2048].

Por último, faltaría modificar la última componente de nuestro modelo encargada de producir las salidas o clases que deseamos. Como hemos observado, la ResNet tiene en su final una capa totalmente conectada (*Fully Connected Layer* o *FC Layer*) que contiene todas las características. Esta recibe una entrada de 1x1x2048 y produce una salida de 1x1x1000, que son las 1000 clases del tipo especifico de ResNet que hemos empleado (ImageNet1K). Para adaptar la salida de la red a las clases que nos interesa, debemos tener en cuenta qué clases nos interesa tener en la salida. Es evidente que la dirección de la mirada tanto del ojo izquierdo como el derecho estarán presentes junto con la orientación de la cabeza de la red original empleada en pcr_upm/students_-headpose. No obstante, esta última venía en forma de ángulos de Euler, a diferencia de lo trabajado hasta ahora con la dirección de la mirada en nuestro proyecto.

Trabajar con ángulos de Euler acarrea una serie de inconvenientes muy importantes:

1. Disposiciones: las rotaciones de los ángulos de Euler varían en función del orden de los ejes sobre los que se rota, dando lugar a resultados diferentes en función de dicho orden. Además, las rotaciones pueden ser extrínsecas o intrínsecas en función de si se aplica en un sistema de coordenadas que es fijo o móvil. Dadas todas las combinaciones posibles, existen 24 combinaciones diferentes de representación con ángulos de Euler.

2. Gimbal Lock: cuando dos de los tres ejes de rotación se alinean, resulta en una pérdida de un grado de libertad en el movimiento.

3. Interacción no intuitiva: la relación entre las rotaciones en ángulos de Euler no resulta muy intuitiva, lo que dificulta la comprensión y control.

Por todo ello, se modifica la salida de la orientación de la cabeza para que se represente en un vector en coordenadas esféricas al igual que la dirección de la mirada. En conjunto con la dirección de la mirada y orientación de la cabeza, se sugiere sacar también el tamaño de las pupilas de los ojos, debido a la información adicional que pueden arrojar, tal y como describimos en la sección 1.3. Además, podría mejorar la precisión de predicción al incluir características adicionales y también partir de base para futuras ampliaciones en este modelo (detectar información adicional relacionadas

con la salud o atención en base al tamaño de las pupilas).

En total, contamos con 8 parámetros: 2 por cada vector de dirección de la mirada y orientación de la cabeza (θ, ϕ), y otro por cada pupila (tamaño en milímetros). Introducimos a modo de *decoder* 5 capas completamente conectadas, una por cada característica deseada (ver listado de código 4.6).

```
self.fc_input_size = self.model.fc.in_features

# Create decoder as fully connected layers producing the desired outputs
self.FcLeftEye = nn.Linear(in_features=self.fc_input_size,
    out_features=2)
self.FcLeftPupil = nn.Linear(in_features=self.fc_input_size,
    out_features=1)
self.FcRightEye = nn.Linear(in_features=self.fc_input_size,
    out_features=2)
self.FcRightPupil = nn.Linear(in_features=self.fc_input_size,
    out_features=1)
self.FcHeadpose = nn.Linear(in_features=self.fc_input_size,
    out_features=2)

self.model.fc = nn.Identity() # passthrough layer
```

Código 4.6: Definición del decoder en Python

De esta forma, nuestro modelo queda totalmente modificado para que pueda tomar una imagen del tamaño de la zona facial de los participantes de nuestro dataset, pasar por una Resnet y producir las clases deseadas. En la figura 4.7 se representa gráficamente el modelo propuesto.

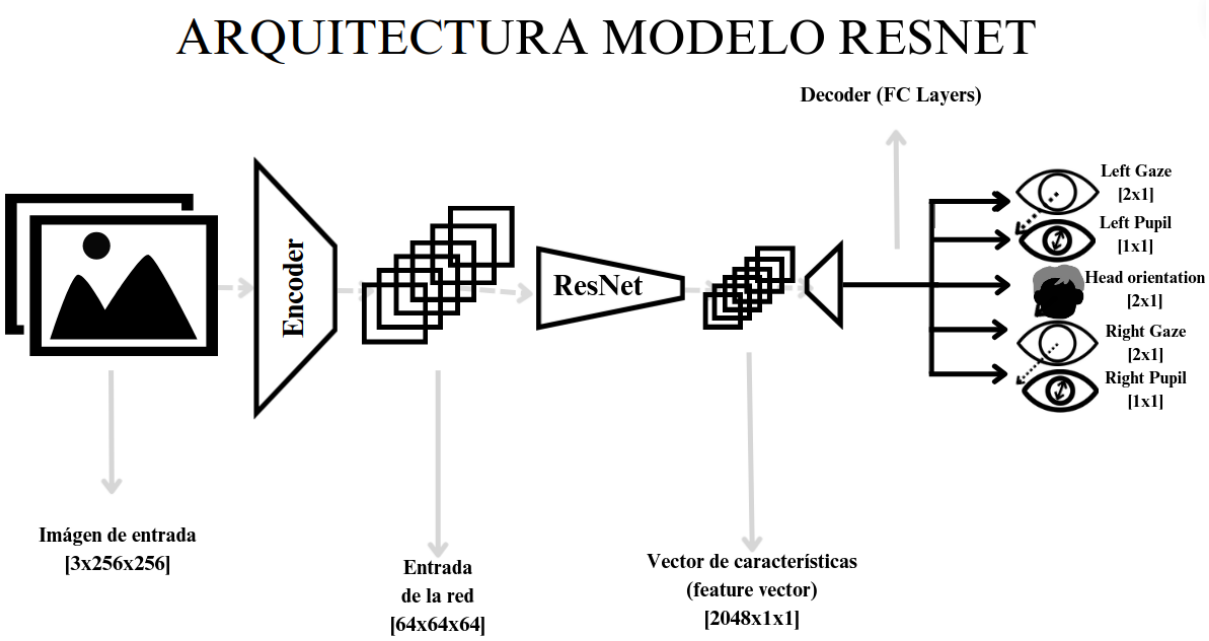


Figura 4.7: Arquitectura del modelo propuesto con ResNet

La red en cuestión ha sido implementada empleando el framework de PyTorch Lightning, que permite crear un modelo mucho más escalable y deshacerse en gran parte del bucle de entrenamiento o las actualizaciones del optimizador, entre otros. Para ello, la ResNet se representa como una clase que hereda del módulo LightningModule, y su aspecto es el mostrado en el fragmento de código 4.7.

```
class CustomNet(pl.LightningModule):
  def __init__(self):
      super(MyLightningModel, self).__init__()
      # Init model components and specific configuration
      self.lossFunction = nn.L1Loss() # MAE
      pass

  def configure_optimizers(self):
      # define optimizers, learning rate initial value, etc...
      pass

  def forward(self, x):
      # Define forward step
      x = self.model(x)
      return x

  def _step(self, batch):
      # compute outputs, compare with targets and get loss value
      target = batch['target']
      output = self.model(batch['image'])

      loss = self.lossFunction(output, target)
      return loss

  def training_step(self, batch, batch_idx):
      # Define training step (e.g log training loss)
      loss = self._step(batch)

      self.log('train_loss', loss, batch_size=self.batchSize,
          on_step=True, on_epoch=True, sync_dist=True)
      return loss

  def validation_step(self, batch, batch_idx):
      # Define validation step (e.g: log hyperparams)
      loss = self._step(batch)

      self.logger.log_hyperparams(self.hparams, loss)
      return loss

  def test_step(self, batch, batch_idx):
      # Define test step
      pass

  def on_validation_epoch_end(self):
      # Execute after validation epoch (e.g: log learning rate current
          value)
      lr =
          self.trainer.lr_scheduler_configs[0].scheduler.get_last_lr()[0]
      self.log('learning_rate', lr, batch_size=self.batchSize,
          on_step=False, on_epoch=True, sync_dist=True)
```

Código 4.7: Ejemplo de red definida en PyTorch Lightning

Si bien esta arquitectura es suficiente para entrenar el modelo y analizar resultados variando hiper-parámetros y otras características de la red, se propone modificar otro de los modelos contenidos en pcr_upm/students_headpose para aprovechar los puntos fuertes de otro tipo de red neuronal.

4.3.2. Implementación del modelo EfficientNet

Si bien se ha explicado y mencionado algunas de los tipos de CNN más famosos y empleados hoy en día, hasta ahora no habíamos hablado de EfficientNet.

EfficientNet [15] es una arquitectura de red neuronal convolucional diseñada para ser más eficiente y precisa en la tarea de clasificación de imágenes. Fue presentada por un equipo de investigadores de Google AI en 2019. Los modelos más grandes y profundos presentados antes que EfficientNet, como ResNet, DenseNet o Inception, logran buenos resultados en tareas de clasificación de imágenes, pero son cada vez más complejos y difíciles de entrenar. Aunque aumentar el tamaño de la red generalmente mejora el rendimiento, no hay un enfoque sistemático sobre cómo escalar la arquitectura de manera eficiente en términos de profundidad, ancho (número de canales) y resolución de entrada. La propuesta del equipo de Google AI aborda el problema de la escalabilidad de las redes neuronales de manera sistemática mediante el uso de un método llamado *Compound Scaling*. Este método escala estas tres dimensiones de manera conjunta, utilizando un pequeño conjunto de hiper-parámetros que mantienen el modelo eficiente y mejoran el rendimiento sin requerir una gran cantidad de recursos computacionales adicionales. Además, utiliza el bloque MBConv, que es un tipo de convolución profunda con atajos de conexión invertidos. Este bloque es muy eficiente en cuanto a memoria y permite mantener la precisión reduciendo el número de parámetros.

En general, EfficientNet logra una mejor relación entre rendimiento y eficiencia computacional que ResNet, gracias al enfoque de escalado compuesto, y también suele implicar un modelo más ligero y menos profundo que algunas de las variantes de ResNet.

Students_headpose implementa la EfficientNetB4 de la librería torchvision, que toma como pesos los correspondientes a ImageNet1K, como en el caso anterior de la ResNet. No obstante, y a diferencia de esta y otras redes convolucionales, toma como entrada una imagen de dimensiones de una resolución notablemente mayor (380x380x3), y significativamente más grande que nuestra imagen de la zona facial. Si

ahora quisiéramos tomar el enfoque contrario al tomado en el caso anterior de ResNet (escalar la imagen de entrada 256x256x3 de nuestro dataset a la de EfficientNetB4), la imagen no resultaría en pérdida de información al perder los bordes de la cara, puesto que ahora se tendría que aumentar su tamaño. No obstante, esto igualmente implicaría una pérdida importante de información debido a la distorsión y pérdida de detalles que ocasionaría realizar un reescalado tan significativo.

Ante esta situación, se plantea emplear el modelo EfficientNetB2, que toma de entrada una imagen 260x260x3, y permitiría escalar la imagen sin temor a sufrir el efecto de distorsión que si conllevaría escalar la imagen a 380 píxeles. Las diferencias entre ResNet y ambas variantes de EfficientNet se pueden observar en 4.4:

Característica	**ResNet-50**	**EfficientNet-B2**	**EfficientNet-B4**
Tamaño de entrada	224x224	260x260	380x380
Profundidad (capas)	50	66*	70*
Número de parámetros	25.6 millones	9.2 millones	19 millones
Precisión en ImageNet1K (Top-1)	76.0 %	82.7 %	84.2 %
Precisión en ImageNet1K (Top-5)	93.3 %	95.5 %	96.6 %

Cuadro 4.4: Comparación entre ResNet-50, EfficientNet-B2 y EfficientNet-B4 en términos de entrada, profundidad, parámetros y efectividad en ImageNet1K. *****: *EfficientNet utiliza bloques compuestos que pueden variar en la cantidad de capas dependiendo de la implementación y del nivel de escalado*

Si bien la diferencia de parámetros es bastante notable entre las tres, especialmente la brecha entre EfficientNetB2 y las otras dos, podemos ver que la precisión obtenida en ImageNetK1 entre las dos versiones de EfficientNet no es muy considerable, y teniendo en cuenta el factor de distorsión que podría provocar usar la variante B4, parece razonable escoger EfficientNetB2 para esta nueva implementación de nuestro modelo.

Se implementa un escalado de la imagen de entrada que redimensiona un tensor

de imagen x a un tamaño de 260x260 píxeles utilizando interpolación bilineal. Este método calcula el valor de cada nuevo píxel basándose en un promedio ponderado de los píxeles vecinos, lo que ayuda a suavizar la imagen y a preservar los detalles. Al establecer align_corners=False, se asegura que la alineación de los bordes de los píxeles sea coherente y se obtengan resultados más naturales para trabajar con la red (ver listado de código 4.8)

```
def forward(self, x):
  # Resize input:
  x = nn.functional.interpolate(x, size=(260, 260), mode='bilinear',
      align_corners=False)

  # EfficientNet:
  x = self.model(x)

  # Decoder:
  leftEye = self.FcLeftEye(x)
  leftPupil = self.FcLeftPupil(x)
  rightEye = self.FcRightEye(x)
  rightPupil = self.FcRightPupil(x)
  headpose = self.FcHeadpose(x)

  return leftEye, leftPupil, rightEye, rightPupil, headpose
```

Código 4.8: Implementación del forward en el modelo de EfficientNet

Finalmente, se implementa el "decoder" de la red de forma análoga al modelo de ResNet, mediante el empleo de 5 capas totalmente conectadas que obtengan las 8 clases diferentes de salida. En la figura 5.1 se muestra la arquitectura final de la red.

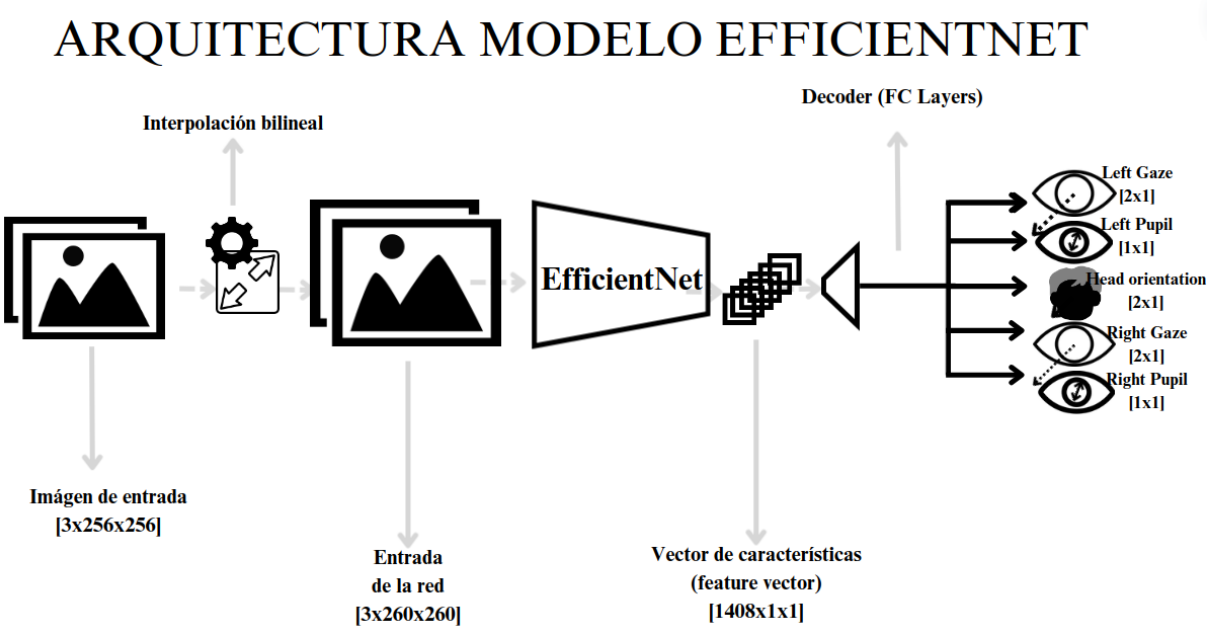


Figura 4.8: Arquitectura del modelo propuesto con EfficientNet

Capítulo 5
Experimentación y análisis de resultados

El presente capítulo tiene como propósito determinar de forma experimental la efectividad y utilidad del modelo neuronal desarrollado. Para ello, es necesario realizar una experimentación que se divide en tres fases: una etapa de entrenamiento de la red, en la cual se realiza una investigación sobre los parámetros que ajusten los pesos de la red para producir una salida o predicción que pueda generalizar correctamente todos los casos de entrada; una etapa de evaluación, en la que se emplea una parte del conjunto de datos diferente al usado durante el entrenamiento para evaluar precisamente si la generalización es correcta empleando unas métricas concretas; y la etapa de test, en la cual se utiliza el conjunto de pesos del modelo entrenado sobre nuevas imágenes para determinar su utilidad en situaciones reales.

5.1. Entrenamiento

Antes de la etapa de entrenamiento, es necesario establecer una serie de variable que afectan directamente al comportamiento, capacidad de aprendizaje, generalización y estructura de la red durante la etapa de *training*. Este conjunto de variables es conocido como *hiper-parámetros*. Dado que el modelo puede emplear como base dos arquitecturas diferentes (ResNet y EfficientNet), es necesario hallar por separado los valores ideales de los hiper-parámetros para cada una de ellas.

En concreto, es fundamental determinar correctamente los valores para los siguientes parámetros:

- Tamaño de lotes (*Batch Size*): número de muestras que la red procesa antes de actualizar los pesos. Un batch size pequeño permite que las actualizaciones sean más frecuentes, pero puede hacer que el aprendizaje sea más ruidoso, mientras

que un tamaño grande genera actualizaciones más estables, aunque más lentas y costosas en términos de memoria.

- Tasa de aprendizaje (*learning rate*): controla la magnitud de los ajustes de los pesos durante la retropropagación. Si el learning rate es demasiado alto, el modelo puede saltarse la solución óptima, mientras que una tasa baja puede hacer que el entrenamiento sea lento o quede atascado en mínimos locales.

- Optimizador: es el algoritmo que ajusta los pesos de la red para minimizar la función de pérdida (medida cuantitativa que indica qué tan bien o mal está funcionando la red, comparando las predicciones del modelo con los valores reales). Algunos optimizadores comunes son SGD (*Stochastic Gradient Descent*) o Adam *Adaptative Moment Estimation*. Afectan a la velocidad de convergencia y la estabilidad del entrenamiento.

- Número de episodios (*epochs*): cantidad de veces que el modelo procesa el conjunto completo de datos de entrenamiento. Un número bajo puede resultar en un entrenamiento incompleto, mientras que un número alto puede llevar a sobre-ajuste (*overfitting*) si no se maneja con cuidado.

No obstante, otros hiper-parámetros como el num_workers (número de subprocesos en paralelo para cargar los datos en la memoria durante el entrenamiento), prefetch_factor (número de batches a precargar en memoria antes de ser utilizados) o el patience (controla cuántos epochs más el modelo puede seguir entrenándose sin mejorar las métricas de validación antes de detener el entrenamiento) tienen impacto directo en la rapidez con la que se entrena el modelo, el uso de memoria y la capacidad de parar cuando el modelo no es capaz de seguir mejorando su aprendizaje.

Debido a la cantidad de configuraciones diferentes posibles, se establece para mayor comodidad una configuración en formato .yml que permita especificar cómo se va a entrenar el modelo (ver figura 5.1).

```yaml
Model:
  dataset_name: eve
  accelerator: 'gpu'
  devices: [0] # -1 for all available
  strategy: 'auto'
  precision: "32-true" # 16-mixed | 32-true | ...
  use_profiler: False
  input_size: [256, 256] # (w,h) in px
  use_checkpoint: False # Set to True to continue training with latest available model .ckpt
  lr_finder: True
  Hyperparameters:
    seed: 2324
    benchmark: False # Set to false to ensure reproducibility with same seed
    deterministic: True # Set to true to ensure reproducibility with same seed
    epochs: 12
    backbone: resnet
    version: 50
    batch_size: 4
    num_classes: 8 # 2 left eye vector + 2 right eye vector + 2 head vector + 2 eye pupils, DO NOT CHANGE
    num_workers: 8
    prefetch_factor: 16
    transfer: False
    tune_fc: False
    lr: 1.0e-07
    patience: 20
    optimizer: 'adam'
```

Figura 5.1: Configuración del entrenamiento e hiper-parámetros

Nótese que se especifican opciones adicionales para continuar el entrenamiento desde un punto de partida específico (use_checkpoint), emplear pesos pre-entrenados si se habilita el *transfer learning* o habilitar solo la actualización de pesos en la capa totalmente conectada de la salida del modelo (tune_fc).

5.1.1. Configuración común para el entrenamiento

Los diferentes entrenamientos realizados con los dos modelos y sus posibles configuraciones fueron realizados a partir de una configuración básica común para todos ellos en lo que respecta a términos de poder computacional, uso de memoria, aceleración gráfica y observación de efectividad. Todos las pruebas de entrenamiento que posteriormente se explicarán fueron entrenadas usando la primera GPU de la máquina Confucio detallada en la sección 3.2.3, con la estrategia automáticamente 'auto' a seguir con esta aceleración. En todas las configuraciones realizadas, se tomaron todos los estímulos contenidos en el subconjunto de Training de todas las cámaras y de todos los participantes del dataset de EVE (esto es, se empleó la totalidad de datos del subconjunto de entrenamiento). Además, se estableció un *frame rate* de 10 imágenes por vídeo, de modo que podamos mantener el tamaño de los lotes fijo independientemente de la longitud del vídeo del estímulo, evitando que ciertos estímulos contribuyan más al entrenamiento y permitiendo una gestión de memoria más sencilla.

El número de epochs de entrenamiento fue arbitrariamente escogido para cada una de las configuraciones dado que dependiendo del tamaño de lote y el valor de otros hiper-parámetros se puede tardar más o menos en llegar a obtener buenos resultados. No obstante, se estableció de forma común un *patience* de 4 epochs en todas las configuraciones. Además, debido a que la tasa de aprendizaje está ligada al tamaño del batch, para los diferentes entrenamientos se busco automáticamente el *learning rate* inicial en función del *batchSize* usando la herramienta *lr finder* de PyTorch. Esta herramienta itera una cantidad determinada de veces al inicio de entrenamiento empleando diferentes tasas para hallar la mejor, mostrando una sugerencia de *lr* inicial. La cantidad de iteraciones para el lr finder se definió como:

$$\text{lr\_finder\_steps} = \lceil \text{len(train)} + \text{len(evaluation)} \rceil * \text{batch\_size} \tag{5.1}$$

Learning rate finder también permite visualizar una gráfica de resultados de la búsqueda realizada, que además de indicar un valor sugerido permite escoger otro en función de la gráfica de evolución. En la figura 5.2 se muestra los resultados de la búsqueda para el modelo EfficientNet con un batch=8, con un valor sugerido en torno a $lr = 1{,}0 \cdot 10^{-9}$. No obstante, vemos que hasta $lr = 1{,}0 \cdot 10^{-5}$ no empieza a decrecer la función de pérdida, por lo que resulta más efectivo establecer una tasa inicial en la parte intermedia de la curva descendente del mínimo absoluto (en torno a $lr = 1{,}0 \cdot 10^{-4}$), de forma que mientras en el learning rate disminuye la función de pérdida también lo haga hasta llegar a su mínimo con el paso de las iteraciones.

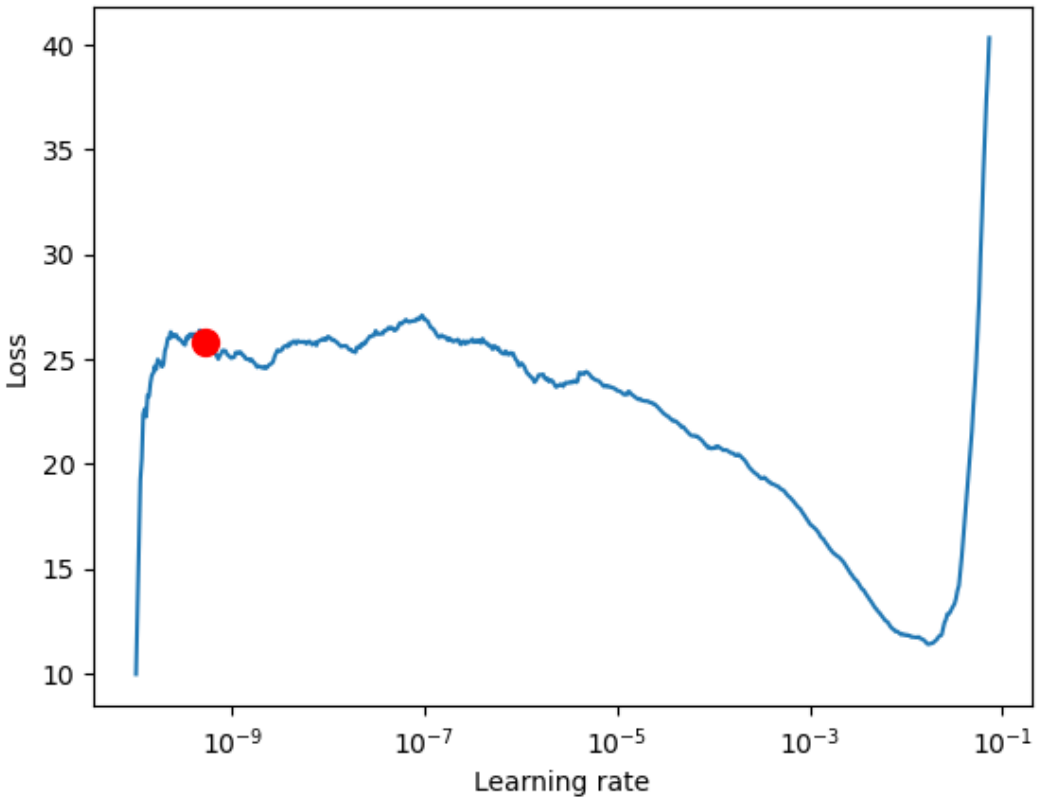


Figura 5.2: Resultado de la búsqueda del learning rate finder para una configuración específica. En rojo, la tasa en concreto sugerida.

Finalmente, para poder comparar correctamente los resultados del entrenamiento entre las distintas configuraciones del modelo se define una función de pérdida común para todas ellas. Debido a las clases de salida que se tienen en el modelo, se definen la función de pérdida total $\mathcal{L}_total$ como:

$$\mathcal{L}_{total} = \alpha \cdot \mathcal{L}_{gaze} + \beta \cdot \mathcal{L}_{pupils} + \gamma \cdot \mathcal{L}_{head} \tag{5.2}$$

Donde $\mathcal{L}_{gaze}$, $\mathcal{L}_{pupils}$ y $\mathcal{L}_{head}$ son las pérdidas de la dirección de la mirada, tamaño de las pupilas en milímetros y orientación de la cabeza, respectivamente. La pérdida de la orientación de la cabeza y dirección de la mirada se define como la diferencia en grados entre la dirección del vector estimado g' y el real g en coordenadas esféricas, en base a la fórmula:

$$\mathcal{L}_{gaze} = \mathcal{L}_{head} = \frac{180}{\pi} \cdot \arccos\left(\frac{g \cdot g'}{||g|| \cdot ||g'||}\right) \tag{5.3}$$

Por otro lado, la pérdida del tamaño de las pupilas es calculado empleando el error

absoluto medio o MAE (*Mean Absolute Error*) entre el tamaño real en milímetros n y el estimado n':

$$\mathcal{L}_{pupils} = |n - n'| \tag{5.4}$$

Los coeficientes α, β y γ ajustan el peso de cada función de pérdida individual al cómputo global. Empíricamente, se establecen en $\frac{1}{3}$ cada uno. La principal razón de esto es que los errores tienen magnitudes parecidas.

5.1.2. Configuración y entrenamiento con ResNet50

Se realizaron diferentes configuraciones para poder entrenar el modelo empleando la arquitectura de ResNet50 variando ciertos hiper-parámetros. En concreto, se entrenó el modelo con ResNet50 con 2 tamaños de lotes diferentes, y estos dos a su vez con dos optimizadores distintos: SGD (*Stochastic Gradient Descent*) y Adam (*Adaptative Moment Stimation*), de modo que se tiene las siguientes configuraciones:

(a) Batch de 8 empleando precisión mixta '16-mixed'. Esto significa que Se utiliza *float* precisión 16 para la mayoría de las operaciones excepto para operaciones críticas, como la acumulación de gradientes, en las que se emplea *float* 32. El learning rate para este tamaño de lote se estableció en $lr = 1{,}0 \cdot 10^{-6}$ para ambos casos (SGD y Adam)

(b) Batch de 4 empleando precisión *float* 32 para todas las operaciones. En este caso, el learning rate elegido fue de $lr = 1{,}0 \cdot 10^{-8}$ para el caso de SGD, mientras que se mantuvo en $lr = 1{,}0 \cdot 10^{-6}$ para el caso de Adam, ya que es un optimizador adaptativo.

Para poder determinar el efecto que tienen las diferentes configuraciones del entrenamiento, en las figuras 5.3 y 5.4 se muestran los ejemplos de las gráficas de los entrenamientos de dos configuraciones "contrapuesta", SGD8 y Adam4.

Primeramente, en la figura 5.3 se observan los resultados para el entrenamiento con precisión mixta, empleando el optimizador del descenso por gradiente estocástico. Vemos que, por un lado, que la función de pérdida total para ambos casos (validación y train) tiene forma de función exponencial decreciente, que indica a priori un correcto entrenamiento. Al final del mismo, el valor de la función se sitúa en rangos comprendidos entre el 10 y el 12 % para ambos casos, obteniendo el valor de pérdida de entrenamiento más bajo en torno al episodio 10 (step 18k) y en torno al episodio

12 (step 22k) en el caso del error de validación. Se observa además que la evolución de las componentes de la función de pérdida en su conjunto también decrecen a lo largo del entrenamiento de pequeñas diferencias, pero todas ellas de forma muy similar.

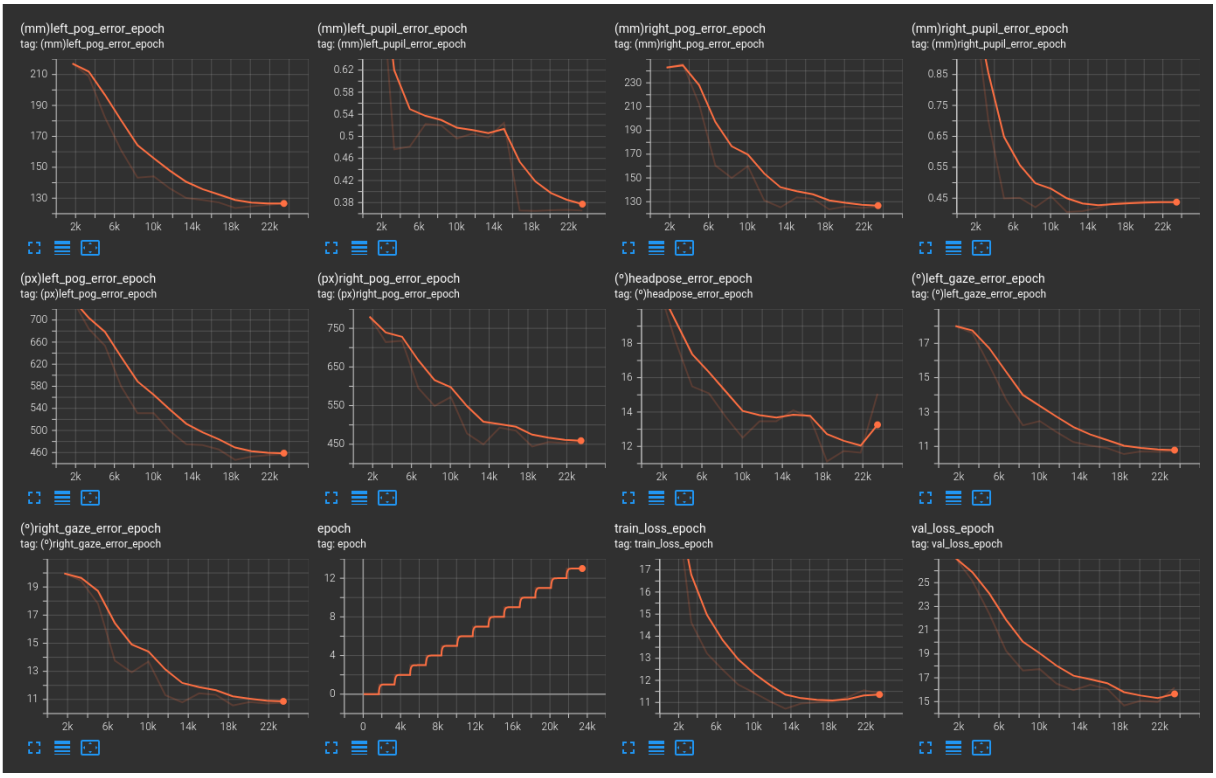


Figura 5.3: Gráficas de evolución de la función de pérdida y métricas para ResNet50 con optimizador SGD y batch de 8

En la figura 5.4 se muestra el resultado arrojado por el optimizador Adam para el mismo tamaño de lote. Vemos que los resultados en este caso son ligeramente mejores a los conseguidos por el optimizador SGD. Esto podría ser en medida debido a que, a diferencia de SGD que actualiza los pesos únicamente en base al gradiente, Adam utiliza tanto el momento exponencial de los gradientes pasados (*momentum*) como el promedio exponencial de los cuadrados de los gradientes. Esto significa que ajusta la tasa de aprendizaje de cada parámetro basándose en la magnitud que tiene cada uno.

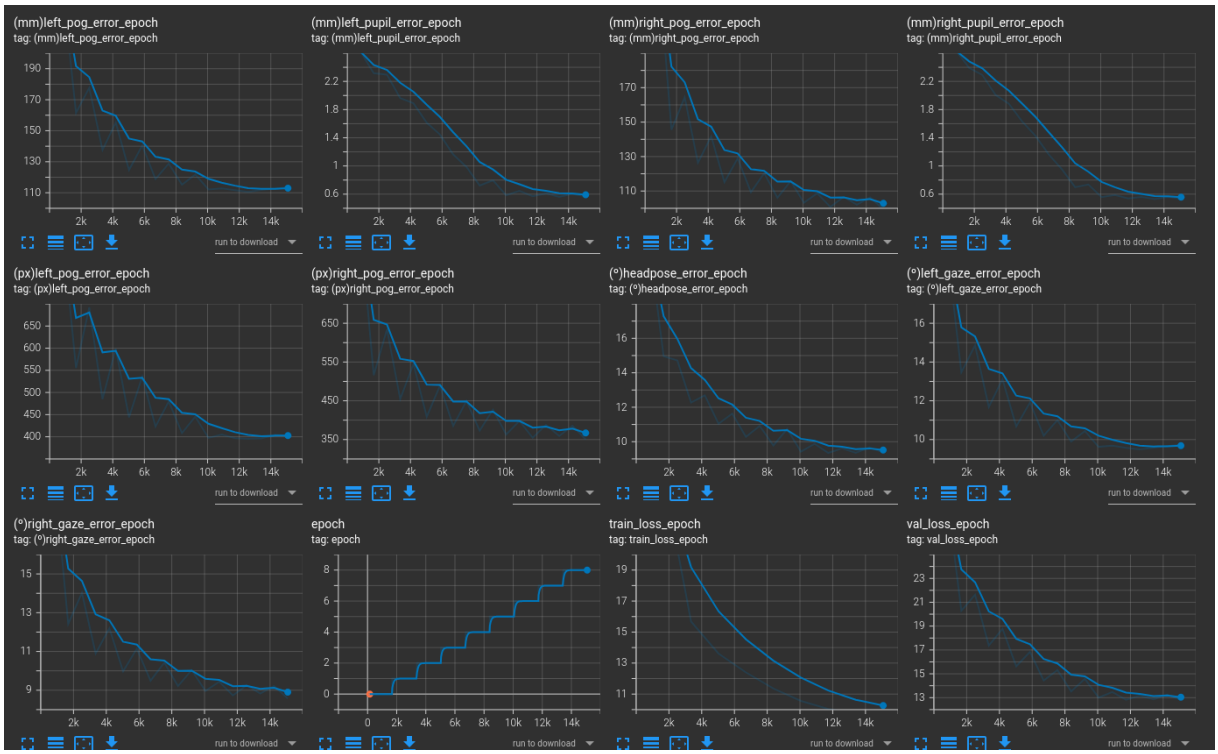


Figura 5.4: Gráficas de evolución de la función de pérdida y métricas para ResNet50 con optimizador Adam y batch de 8

Faltaría entonces conocer si utilizar un tamaño de lote menor pero con mayor precisión podría arrojar mejores resultados. En la figura 5.5 se observan los resultados con precisión *float32* para la configuración con optimizador Adam y batch de 4. Podemos observar que al emplear un tamaño más pequeño de lote (batch=4), la pérdida de entrenamiento y de evaluación empieza notablemente más alta que en el caso anterior, si bien conseguir bajar de manera eficiente hasta llegar a su mejor resultado en torno al episodio 12. Si nos fijamos en la evolución de las diferentes métricas, podemos observar como todas descienden conjuntamente de forma muy similar. Durante este entrenamiento, es el propio callback de *EarlyStopping* el que detiene el training dado que el modelo ya no consigue mejores resultados tras esperar con el patience=4 configurado previamente.

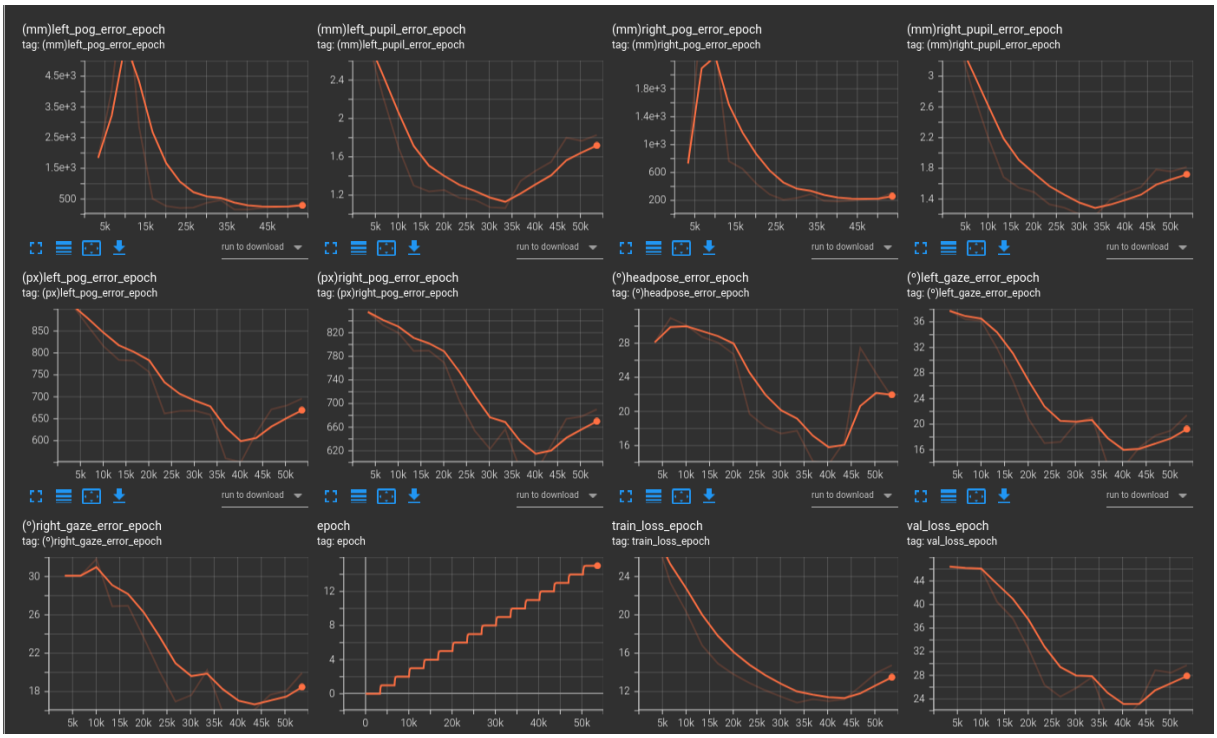


Figura 5.5: Gráficas de evolución de la función de pérdida y métricas para ResNet50 con optimizador Adam y batch de 4

A juzgar por las notorias diferencias entre utilizar un optimizador u otro, y entre usar un tamaño de lote más grande o más pequeño, podríamos decir que a pesar de usar mayor precisión en la configuración con batch=4, duplicar el tamaño de lote y emplear precisión mixta permite obtener mejores resultados en la función de pérdida del entrenamiento tanto en el error global de las métricas. Además, estos resultados son obtenidos en menor cantidad de episodios. Además, el uso de un optimizador "adaptativo" como es el caso de Adam a permitido obtener unos resultados ligeramente mejores a los conseguidos por el optimizador SGD.

5.1.3. Configuración y entrenamiento con EfficientNet-b2

Las configuraciones para el caso del modelo con EfficientNet-b2 se establecieron con los mismos criterios de tamaño de lote, precisión y learning rate que en el caso de ResNet50. No obstante, la menor profundidad y tamaño de la red permiten emplear tamaños de lotes más grandes que teóricamente podrían mejorar la capacidad de generalización en las estimaciones del modelo:

(a) Batch de 16 empleando precisión mixta y *learning rate inicial de* $1{,}1 \cdot 10^{-5}$ para Adam y $lr = 5{,}5 \cdot 10^{-4}$ para SGD.

(b) Batch de 8 empleando precisión *float* 32 y tasa de aprendizaje inicial de $lr = 5{,}25 \cdot 10^{-5}$ para el caso de Adam y $2{,}75 \cdot 10^{-4}$ para SGD.

De nuevo, ambas configuraciones fueron entrenadas empleando los optimizadores *Stochastic Gradient Descent* y Adam con la misma tasa de decaimiento de pesos.

En la figura 5.6 se muestran los resultados para la configuración con $batch = 16$ empleando el optimizador Adam. En las gráficas se pueden observar un decrecimiento de la función de pérdida de entrenamiento y del error de las métricas de evaluación casi idéntico, que se reduce a lo largo de los episodios en un 50 %. Si observamos la evolución de las diferentes métricas, vemos que todas ellas van disminuyendo su error tímidamente, a excepción de los errores en el tamaño de las pupilas, que si experimentan una bajada mucho más acentuada.

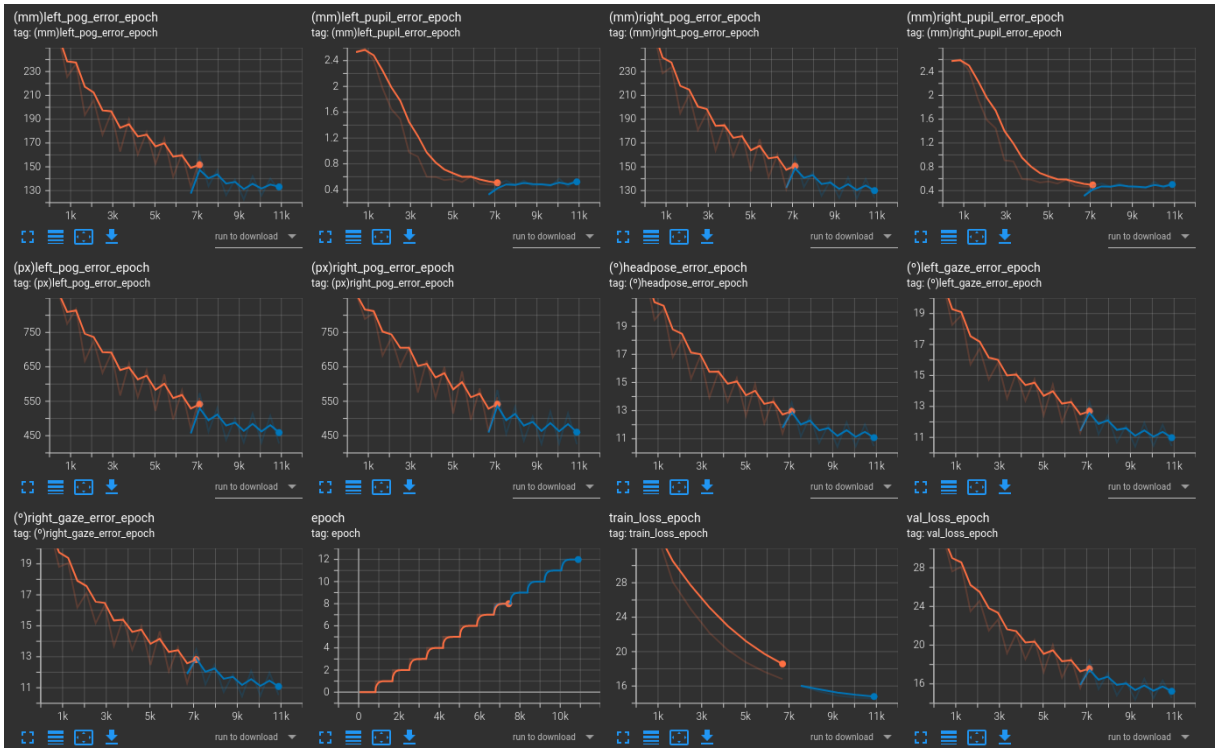


Figura 5.6: Gráficas de evolución de la función de pérdida y métricas para EfficientNet con optimizador Adam y batch de 16. *Entrenamiento pausado y continuado desde el checkpoint en epoch=8. En naranja, epochs del 1 al 7. En azul, del 8 al final*

Veamos ahora los resultados obtenidos con SGD y mismo tamaño de lote, mostrados en la figura 5.7. En este caso observamos una caída más acentuada en todas las métricas y función de pérdida, llegando a obtener el error más pequeño en torno al episodio 14.

No obstante, el valor de pérdida más pequeño en los datos de validación ($\mathcal{L} = 21$) es superior al de la configuración anterior ($\mathcal{L} = 15$).

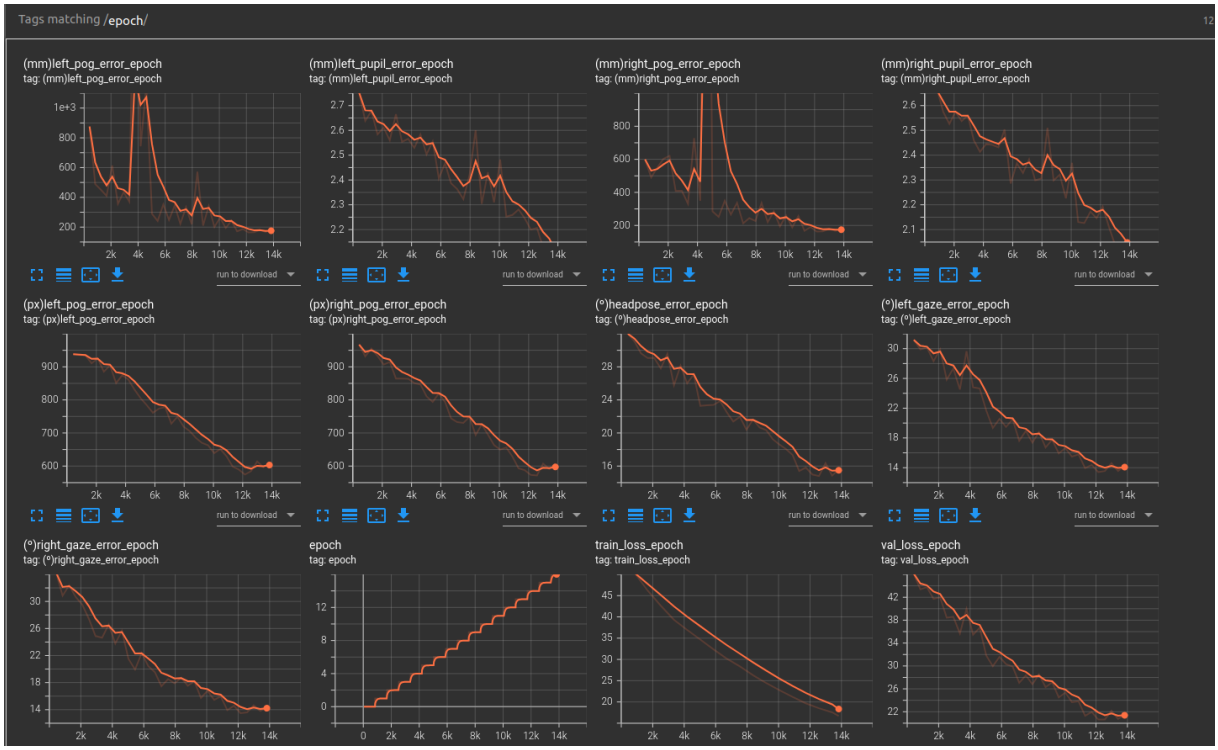


Figura 5.7: Gráficas de evolución de la función de pérdida y métricas para EfficientNet con optimizador SGD y batch de 16

Tratamos a continuación de averiguar si aumentar la precisión disminuyendo el tamaño de lote mejora los resultados del modelo. En la figura 5.8 se observan los resultados para la configuración con optimización Adam y batch 8. Observamos de nuevo un decrecimiento muy rápido en las métricas de validación y en la función de pérdida. No obstante, vemos que para el episodio 3 estas ya no consiguen mejorar, a pesar que la función de pérdida del entrenamiento si continúa en su tendencia descendente.

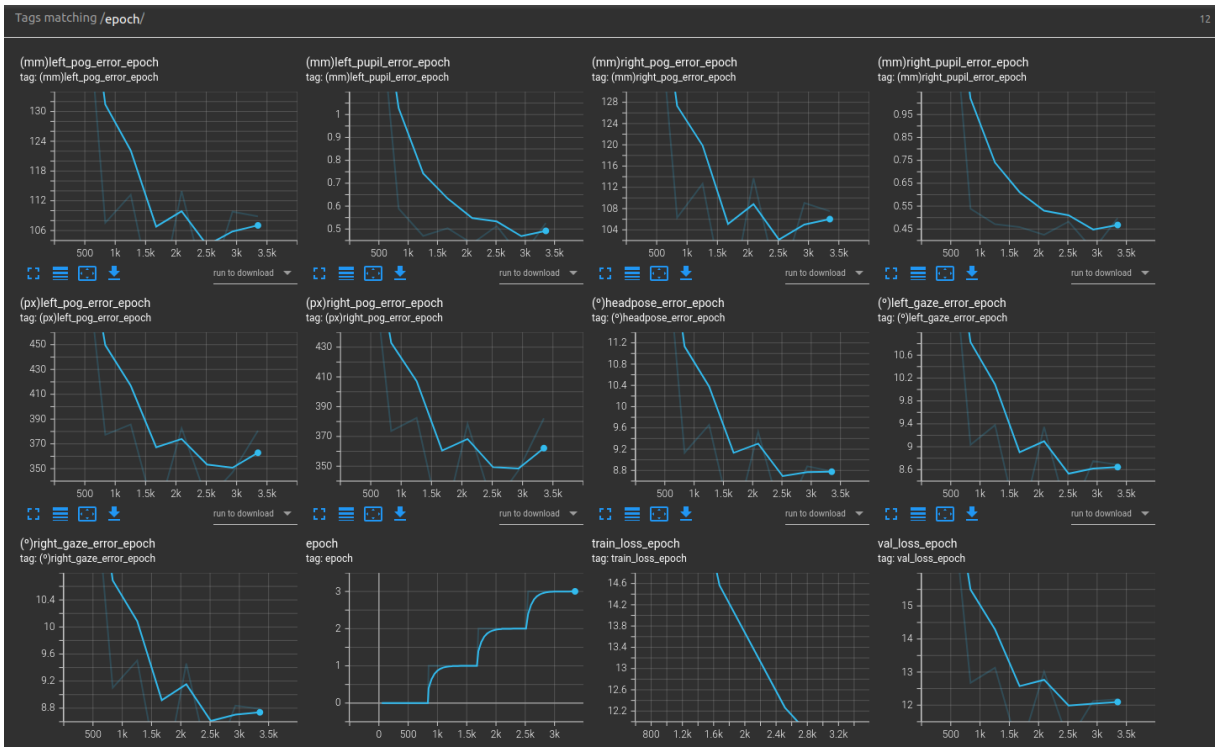


Figura 5.8: Gráficas de evolución de la función de pérdida y métricas para EfficientNet con optimizador Adam y batch de 8

Los resultados obtenidos para las diferentes configuraciones de esta red sugieren que, a diferencia del caso observado en ResNet, emplear precisión *float32* y menor tamaño de lote puede conseguir resultados superiores a configuraciones con precisión mixta y con doble tamaño de batch. Esta diferencia observada con respecto a lo visto anteriormente en ResNet podría deberse, principalmente, a que estamos empleando un mayor batch size tanto en las configuraciones con precisión *32-true* como en aquellas con precisión *16-mixed*.

5.2. Validación

Con el fin de evaluar los resultados obtenidos durante el entrenamiento y capacidad de generalización de la red, se establecen las siguientes métricas de evaluación:

- **Error medio angular**: promedio de las distancias angulares entre las predicciones y los valores reales, expresado en grados. Se emplea como métrica para la dirección de la mirada y orientación de la cabeza.
- **Error medio absoluto**: medida promedio de las diferencias absolutas entre las predicciones de un modelo y los valores reales. Se emplea como métrica para el tamaño de las pupilas en milímetros.

- **Error euclídeo**: distancia recta entre dos puntos en un espacio multidimensional. Esta métrica es aplicada al Point of Gaze en pantalla y se utiliza los píxeles como unidad métrica.

El punto de mira en pantalla para cada ojo (PoG) se calcula empleando la fórmula para hallar la intersección de una recta que viene definido por un punto (origen del ojo) y un vector de dirección (vector de dirección de la mirada), con un plano que sabemos que se encuentra en Z=0 (plano XY):

$$P_{\text{intersection}} = \begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = O_{gaze} + t \cdot \vec{g} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \frac{-(Ax_0 + By_0 + Cz_0 + D)}{Ag_x + Bg_y + Cg_z} \begin{pmatrix} g_x \\ g_y \\ g_z \end{pmatrix} \quad (5.5)$$

Donde t es el parámetro que indica la posición de un punto a lo largo de la recta. Si $t = 0$, el punto coincide con el origen O_{gaze}. Los coeficientes A, B y C definen la normal del plano $\vec{n} = [A, B, C]^T$. El punto $P_{intersection}$ indica el punto de intersección en la pantalla en milímetros, y debe ser multiplicado por los coeficientes correspondientes para transformarlo a las coordenadas en pantalla en píxeles:

$$P_{\text{screen}} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \cdot \begin{pmatrix} p'_x \\ p'_y \end{pmatrix} \quad (5.6)$$

El subconjunto de Validación del dataset es empleado para obtener una media de los resultados de estas métricas. Los resultados obtenidos para las configuraciones de ResNet se muestran en la tabla 5.1.

ResNet config	*Left Gaze (°)*	*Left Pupil (mm)*	*Left PoG (mm)*	*Left PoG (px)*	*Head (°)*	*Right Gaze (°)*	*Right Pupil (mm)*	*Right PoG (mm)*	*Right PoG (px)*
$Adam_{b=8}$	**9.57**	0.57	**111.71**	**396.51**	**9.35**	**8.73**	0.54	**100.78**	**354.47**
$SGD_{b=8}$	10.77	**0.37**	126.48	457.48	15.07	10.73	**0.44**	125.86	456.38
$Adam_{b=4}$	13.28	1.46	157.45	551.40	13.79	15.26	1.48	184.15	583.92
$SGD_{b=4}$	16.71	0.99	211.55	646.69	19.11	20.61	1.09	342.06	705.23

Cuadro 5.1: Resultados de validación para las configuraciones de ResNet

Por otro lado, los resultados obtenidos con las configuraciones de EfficientNet son los siguientes:

Efficient Net config	*Left Gaze (°)*	*Left Pupil (mm)*	*Left PoG (mm)*	*Left PoG (px)*	*Head (°)*	*Right Gaze (°)*	*Right Pupil (mm)*	*Right PoG (mm)*	*Right PoG (px)*
$Adam_{b=16}$	10.41	**0.48**	122.77	429.81	10.44	10.45	**0.46**	121.50	425.43
$SGD_{b=16}$	13.40	2.20	164.50	574.16	14.98	13.50	2.19	162.74	573.55
$Adam_{b=8}$	**7.50**	0.50	**88.65**	**308.46**	**7.66**	**7.54**	**0.46**	**87.61**	**305.27**
$SGD_{b=8}$	17.59	2.57	216.10	693.09	19.25	17.40	2.63	228.04	686.22

Cuadro 5.2: Resultados de validación para las configuraciones de EfficientNet

Los mejores resultados con ResNet se obtienen con la configuración que emplea optimizador Adam, batch size de 8 y precisión mixta; mientras que emplear esa misma configuración pero con precisión *float32* en EfficientNet mejora los resultados anteriores, siendo así la mejor configuración encontrada.

5.3. Test en vídeos diferentes a los de EVE

En el apartado anterior, se compararon los resultados obtenidos de las diferentes configuraciones del modelo empleando un set de validación con imágenes de nuevos participantes que no habían formado parte del conjunto de datos empleado durante el entrenamiento. Si bien este informe de resultados parece suficiente para demostrar la generalización de la predicción del modelo, se propone emplear el mejor modelo pre-entrenado para emplearlo en unos vídeos de prueba propios grabados a través de la cámara del portátil, y también en nuevas imágenes modificadas del dataset modificando el fondo de las imágenes.

5.3.1. Calibración de cámara y pruebas sobre vídeos propios

Para realizar esto de forma correcta, primero ha de calibrarse la cámara del portátil, para poder eliminar la distorsión de los vídeos generados. Este proceso se realiza empleando OpenCV. Primeramente, se toman diferentes imágenes de un patrón de calibración, similar a un tablero de ajedrez como el mostrado en la figura 3.2. A continuación, se emplean los métodos de cv2.calibrateCamera y cv2.getOptimalNewCameraMatrix para obtener los coeficientes de distorsión y la matriz de paramétros intrínsecos de nuestra cámara:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 774{,}68 & 0 & 283{,}87 \\ 0 & 778{,}055 & 110{,}78 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

$$\omega = \begin{pmatrix} k_1 \\ k_2 \\ p_1 \\ p_2 \\ k_3 \end{pmatrix} = \begin{pmatrix} -0{,}051 \\ 0{,}035 \\ -0{,}028 \\ -0{,}006 \\ -0{,}014 \end{pmatrix} \tag{5.8}$$

Donde k_i son los coeficientes de distorsión radial y p_i los coeficientes de distorsión tangencial.

Los vídeos de prueba grabados son modificados para eliminar la distorsión empleando cv2.undistort. Dado que los vídeos originales grabados tiene una dimensión de 1920x1080 píxeles, han de ser redimensionados al tamaño de las imágenes empleadas para calibrar la cámara (640x480) para que la eliminación de la distorsión se haga de forma correcta. De esta forma, ya podremos pasar al código de test los frames de vídeo pre-procesados. Dado que la dirección de la mirada ha de corregirse teniendo en cuenta los paramétros intrínsecos de la cámara, se siguen los pasos correspondientes para este tipo de correcciones en sistemas de estimación de la mirada explicado en diversos artículos. Un ejemplo de estos es *Revisiting Data Normalization for Appearance-Based Gaze Estimation*[18]. A partir de los parámetros intrínsecos de la cámara, los coeficientes de distorsión y una serie de *landmarks* de la cara, se halla las matrices de rotación de corrección de la mirada para cada uno de los ojos y la cabeza, tal y como vienen en el dataset de EVE.

```
    def getRotateFixGaze(faceModel, rvec, tvec):
        """
        Get the rotation correction for both eyes
        """
        tvec = tvec.reshape((3,1))

        headRot = cv.Rodrigues(rvec)[0]
        coordsLandmarks = np.dot(headRot, faceModel) + tvec

        rightEye = 0.5*(coordsLandmarks[:,0] +
            coordsLandmarks[:,1]).reshape((3,1))
        leftEye = 0.5*(coordsLandmarks[:,2] +
            coordsLandmarks[:,3]).reshape((3,1))

        face1 = 0.5*(coordsLandmarks[:, 1] + coordsLandmarks[:,
            2]).reshape((3,1))
        face2 = 0.5*(coordsLandmarks[:, 4] + coordsLandmarks[:,
            5]).reshape((3,1))
        faceCenter = 0.5*(face1 + face2)

        data = []
        for coords in (leftEye, rightEye, faceCenter):
            distance = np.linalg.norm(coords)
            headRotX = headRot[:, 0]

            forward = (coords / distance).reshape(3)

            down = np.cross(forward, headRotX)
            down /= np.linalg.norm(down)

            right = np.cross(down, forward)
            right /= np.linalg.norm(right)

            eyeRot = np.c_[right, down, forward].T
            data.append(eyeRot)

        return data
```

Código 5.1: Función para hallar las matrices de corrección de la mirada

Una vez se han hallado los vectores de dirección de la mirada corregidos, ya se puede dibujar correctamente sobre los vídeos de prueba:

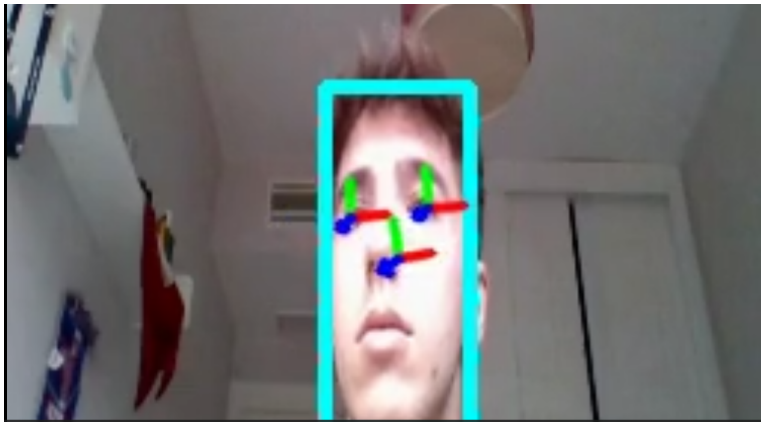


Figura 5.9: Test de la predicción del modelo en el vídeo sin distorsión grabado

Durante la fase de calibración se ha extraído la matriz de intrínsecos de la cámara y los coeficientes de distorsión, pero no se tiene información de la profundidad de la imagen dado que la cámara empleada no es de profundidad. Si quisiéramos dibujar la aproximación del punto de mira en la pantalla de visualización, debería estimarse la distancia a la que se encuentra nuestra persona en el momento de grabar el vídeo con respecto a la cámara (unos 50cm), y a partir de ahí transformar las coordenadas de los ojos en pantalla al mundo 3D para poder posteriormente calcular la intersección con la pantalla, tal y como se explicó en pasos anteriores.

5.3.2. Data Augmentation

En ocasiones, hasta los datasets más variados y extensos contienen patrones comunes que pueden ocasionar que los resultados funcionen bien en su conjunto de validación pero no en otras imágenes. En el caso del dataset empleado, EVE, se ha visto que contiene multitud de puntos de vista, participantes con diferentes aspectos físicos, ropa, sexo, o raza. No obstante, en todos ellos, hay una característica en común: el croma verde de fondo. Si bien el dataloader realiza transformaciones aleatorias en la iluminación de las imágenes para variar su aspecto ligeramente, la red puede estar empleando como característica el fondo de la imagen, común en todas las entradas, para su predicción en alguna de las clases de salida. Si bien la dirección de la mirada no parece tener ninguna relación con el fondo, las estimaciones de la orientación de la cabeza del sistema pueden estar siendo, en mayor o menor medida, predecidas en base a la apariencia del fondo.

Pongamos el ejemplo concreto de una imagen de entrada tomada por la cámara izquierda. Desde esta perspectiva, el fondo de la imagen será visible en la zona derecha

de la imagen, y tendrá una forma u otra en función de la rotación de la cabeza del participante. Para averiguar si este patrón ha afectado a la predicción de la red y la medida en la que haya podido afectar, se modifica el conjunto de validación para modificar de forma aleatoria el fondo de cada uno de los vídeos. Esto es, aplicar *Background Augmentation*. El objetivo será comparar las métricas obtenidas en el conjunto de validación original con respecto a este nuevo conjunto de validación con el fondo de la imagen modificado.

Para modicar el fondo, se estableció una clase para transformar las imágenes del conjunto de validación de forma análoga a la clase que varía la iluminación de cada fotograma de entrada de la red. Estas imágenes son convertidas a formato HSV, y posteriormente se aplica una máscara para aislar todo los píxeles de la imagen que no correspondan con el color verde. Los píxeles del fondo son reemplazados por imáges de la base de datos *Study Room Object Detection Computer Vision Project*. Adicionalmente se creó un script que permite generar directamente los vídeos con el fondo modificado para guardarlos en memoria si se desea para futuras pruebas. Por otro lado, la región de interés (ROI) de los ojos se preserva en todo momento para evitar que píxeles individuales sean modificados por error en el proceso de filtrado.

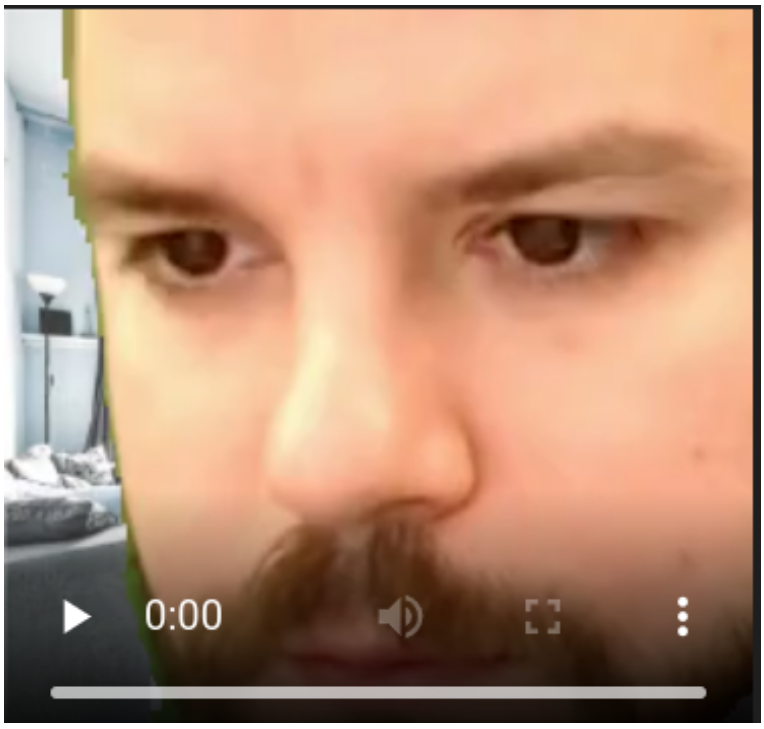


Figura 5.10: Imagen de entrada (256x256) con el fondo modificado

Una vez modificado, se prueba el modelo pre-entrenado con los nuevos vídeos de validación para hallar las métricas de rendimiento. Para ello se emplea el script de evaluación, students_headgaze_eval.py, que directamente realiza la evaluación del conjunto especificado. En la tabla 5.3 se muestra la comparativa de las métricas de validación entre el subconjunto de validación original y el modificado con fondos aleatorios para los pesos obtenidos de la configuración $Adam_8$ de ResNet50:

Validation Set	*Left Gaze (°)*	*Left Pupil (mm)*	*Left PoG (mm)*	*Left PoG (px)*	*Head (°)*	*Right Gaze (°)*	*Right Pupil (mm)*	*Right PoG (mm)*	*Right PoG (px)*
Original	9.56	0.57	111.70	396.51	9.35	8.73	0.54	100.78	396.51
Augmented	12.42	0.63	147.86	527.48	12.41	11.93	0.62	139.20	492.83

Cuadro 5.3: Comparativa métricas de validación con y sin Data Augmentation

Vemos que el error en la predicción de la mirada aumenta en 3(°) en el subconjunto aumentado, así como en la predicción de la orientación de la cabeza. Si nos fijamos en el error en el tamaño de las pupilas, vemos que es casi inexistente (0.07mm, unos 70μm). Estos resultados sugieren que, para la configuración empleada, el fondo puede haber sido ligeramente empleado en la estimación de la dirección de la mirada y orientación de la cabeza, pero no lo suficiente como para haber sido una característica determinante en la estimación de estas métricas.

No obstante, podría ser de utilidad aplicar *Background Augmentation* al conjunto de entrenamiento y de validación en futuros entrenamientos para mejorar la generalización de la red en estos términos.

Capítulo 6
Conclusiones

En este último capítulo se cerrará el desarrolló del TFG exponiendo las conclusiones obtenidas tras su realización, así como las posibles líneas de desarrollo a futuro.

6.1. Objetivos conseguidos

A continuación, se repasan la serie de objetivos establecidos en el apartado 2.2

6.1.1. Objetivo principal

El principal objetivo del presente TFG era crear un sistema de estimación de la dirección de la mirada humana empleando Deep Learning. A lo largo de las diferentes secciones que componen esta memoria, se ha ido explicando con detalle los pasos seguidos que han hecho que podamos considerar este objetivo como cumplido.

El capítulo 4 se centro principalmente en explicar como se realizó este sistema. En la sección 4.1 se desarrolló la primera componente del mismo, el Dataloader, que daría pie a a poder alimentar el futuro sistema con imágenes de más de 50 participantes de diferentes razas, sexo y presencia o ausencia de gafas que permitiese que el sistema aprendiese características generales de los humanos para poder cometer su objetivo.
En el posterior capítulo 4.3 se explicó la arquitectura e implementación del modelo en sí, empleando como base una red de estimación de la orientación de la cabeza para poder ampliar su funcionalidad y aplicación. Durante esta etapa también se comparó las diferentes formas de abordar la entrada y salida de datos en la red, así como la propia arquitectura de la misma.

Finalmente, en los apartados de la sección 5 se pudo realizar el entrenamiento del sistema desarrollado para que aprendiese a estimar la dirección de la mirada tanto con una red residual (ResNet50) como una red convolucional más optimizada en términos

de eficiencia y parámetros como era EfficientNet-b0. Cada una de estas redes fueron entrenadas con distintas configuraciones de hiper-paramétros en la sección 5.1 con el objetivo de averiguar el impacto de estos en las estimaciones obtenidas. La evaluación de este impacto y las conclusiones obtenidas sobre las diferentes configuraciones realizadas fueron descritas en la sección 5.2.

6.2. Objetivos secundarios

A continuación se proceden a enumerar los objetivos secundarios propuestos y como se han logrado:

1. **Creación y desarrollo del modelo de D.L de estimación de la dirección de la mirada humana en conjunción con modelos pre-existentes de dominios similares**: en la sección 4.3 se introduce la cuestión de abordar la implementación del sistema de estimación de la mirada empleando como base el modelo del Laboratorio de Percepción Computacional y Robótica de la UPM, pcr-upm/students_headpose, que estima la orientación de la cabeza. Este a su vez emplea el framework pcr-upm/image_framework, que contiene herramientas y clases que permiten generalizar el manejo de anotaciones, imágenes y visualización. Esto a permitido aumentar las posibles aplicaciones del repositorio anterior.

2. **Investigación sobre la impacto del tipo de red empleado como base del modelo neuronal en términos de eficacia y eficiencia**: Se ha visto que los modelos convolucionales ResNet50 y EfficientNet-b2 difieren en gran medida en términos de número de pesos, profundidad y precisión sobre conjuntos de datos como ImageNet-K1. En el caso de nuestro modelo y el dataset empleado (EVE), se ha observado diferencias notorias tanto en términos de eficacia como de eficiencia.

3. **Elaboración de un informe de resultados detallado**: Los entrenamientos correspondientes a las diferentes configuraciones del sistema se han registrado en TensorBoard, permitiendo crear una variedad de gráficas de la evolución del aprendizaje del modelo, su función de pérdida y las diferentes métricas especificadas en la sección 5.2.

4. **Realización de pruebas técnicas del modelo generado y entrenado a partir de vídeos propios**: El proceso de calibración de la cámara de un portátil

y la grabación de vídeos para probar la eficacia de alguno de los modelos pre-entrenados han aportado fiabilidad a los resultados obtenidos.

6.3. Futuras líneas de desarrollo

En virtud de los objetivos cumplidos en el desarrollo del presente proyecto, se podría considerar que el sistema creado puede ser de gran utilidad en un amplio espectro de aplicaciones. No obstante, nuevas líneas de investigación podrían nacer a partir de este TFG para mejorar el actual sistema:

- **Data Augmentation**: dados los puntos expuestos en la sección 5.3.2, entrenar el sistema con las imágenes modificadas podrían mejorar la estimación del sistema y generalizar su uso.
- **Especialización del sistema en Conducción Autónoma**: el database empleado calcula el punto de mira de los humanos en una pantalla plana de humanos en un set de grabación estándar. Emplear otras bases de datos como *Drive and Act*[8], que posee información de profundidad e imágenes en el habitáculo de un vehículo, podría especializar el sistema en el campo de la Conducción Autónoma.
- **Exploración de otras redes convolucionales**: a la vista de los resultados obtenidos empleando ResNet50 y EfficientNet-b2, otras variantes de estas redes o de otros tipos de modelos convolucionales podrían emplearse para mejorar las estimaciones del sistema.

Capítulo 7
Anexo

A. Disponibilidad del código y documentos relacionados

- (GitHub) **images_framework**[1]: Framework que contiene las herramientas utilizadas relacionadas con el manejo de imágenes, anotaciones y otros submódulos.
- (GitHub) **saul24_gaze**[2]: Contiene el código completo desarrollado en el proyecto. Esta contenido como submódulo del anterior repositorio del Laboratorio de Percepción y Robótica de la UPM. El repositorio no es público, si bien podría darse acceso al mismo bajo solicitud.
- (OneDrive) **Documentos adicionales**[3]: Dado que el anterior repositorio del código está restringido, se ha creado un OneDrive público que contiene tanto las imágenes de las gráficas de entrenamiento de los modelos del sistema, como sus logs y los modelos de pesos ".ckpt". También se pueden encontrar vídeos de prueba en los que se dibuja la dirección de la mirada, orientación de la cabeza en vídeos tanto del conjunto de datos empleados como de vídeos de prueba.

B. Aceleración de entrenamiento

Debido a la limitación en el equipo hardware para realizar un entrenamiento con más de una GPU y con un batch size mayor, se han empleado varias técnicas de aceleración además de usar los pesos pre-entrenados de ResNet50 y EfficientNetb2 para ImageNetK1:

[1] `https://github.com/pcr-upm/images_framework`
[2] `https://github.com/pcr-upm/saul24_gaze`
[3] `https://urjc-my.sharepoint.com/:f:/g/personal/s_navajas_2020_alumnos_urjc_es/El0EpzPBnKNAlCV6YOsSK1ABhoXwbtjMLeFcYhXy2_-y6g?e=MJJE5m`

- **Simulación de un mayor batch size**: se utiliza la técnica de acumulación del gradiente para aumentar el tamaño de los lotes de manera "virtual". Esta técnica consiste en que los gradientes calculados a partir de varios pasos del *forward* se acumulen antes de actualizarse, simulando un tamaño efectivo de lote mayor al especificado en los hiper-parámetros.

- **Compilación del grafo de operaciones del modelo**: Durante el entrenamiento de los modelos propuestos se ha empleado *torch.compile* para acelerar la ejecución del modelo, rediciendo la sobrecarga de Python. Esta herramienta permite crear una versión optimizada del modelo capturando el grafo computacional mediante una serie de optimizaciones internas, que incluyen reduciendo de cálculos innecesarios, combinación de operaciones y optimización del flujo de datos, entre otros.

- **Aumento del intervalo de validación**: Si empleamos lotes de tamaño pequeño, los pesos pueden llegar a actualizarse más rápido de lo normal y aumentar el riesgo de overfitting. Aumentar el número de comprobaciones con el subconjunto de validación cada epoch de entrenamiento permite corregir más rápido posibles sobre-ajustes del modelo y ajustar patrones incorrectos durante el entrenamiento.

- **Ajuste dinámico de la tasa de aprendizaje**: Se emplea el callback *ReduceLROnPlateau* para ajustar dinámicamente el learning rate cuando el error en validación deja de mejorar en el entrenamiento, mejorando la convergencia del modelo. Además, en caso de llegar a un mínimo local, permite disminuir el *lr* para permitir escapar del mismo, acelerando el entrenamiento. Esto, en conjunto con el aumento del intervalo de validación, es fundamental para poder encaminar el entrenamiento correctamente.

Bibliografía

[1] Alex Krizhevsky, I. S., and Hinton, G. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (2012).

[2] Duque, A., and Vázquez, C. Implicaciones clínicas del uso del tamaño pupilar como indicador de actividad psicológica: una breve revisión. *Clínica y Salud 24*, 2 (2013), 95–101.

[3] Guestrin, E., and Eizenman, M. General theory of remote gaze estimation using the pupil center and corneal reflections. vol. 53, pp. 1124–1133.

[4] Juniatur. La unión europea obliga a instalar un nuevo sistema para detectar la fatiga de los conductores, 2022.

[5] Kaiming He, Xiangyu Zhang, S. R., and Sun, J. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)* (2015).

[6] Kopuklu, O., Zheng, J., Xu, H., and Rigoll, G. Driver anomaly detection: A dataset and contrastive learning approach. In *IEEE/CVF Winter Conference on Applications of Computer Vision* (2021).

[7] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE* (1998).

[8] Martin, M., Roitberg, A., Haurilet, M., Horne, M., Reib, S., Voit, M., and Stiefelhagen, R. Drive and act: A multi-modal dataset for fine-grained driver behavior recognition in autonomous vehicles. In *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2019).

[9] Martin Koestinger, Paul Wohlhart, P. M. R., and Bischof, H. Annotated Facial Landmarks in the Wild: A Large-scale, Real-world Database

for Facial Landmark Localization. In *Proc. First IEEE International Workshop on Benchmarking Facial Image Analysis Technologies* (2011).

[10] Martyniuk, T., Kupyn, O., Kurlyak, Y., Krashenyi, I., Matas, J., and Sharmanska, V. Dad-3dheads: A large-scale dense, accurate and diverse dataset for 3d head alignment from a single image. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (2022).

[11] Mohd Faizan Ansari, Pawel Kasprowski, P. P. Person-specific gaze estimation from low-quality webcam images. In *IEEE Sensors 2023* (2023).

[12] Park, S., Aksan, E., Zhang, X., and Hilliges, O. Towards end-to-end video-based eye-tracking. In *European Conference on Computer Vision (ECCV)* (2020).

[13] Rishi Athavale, Lakshmi Sritan Motati, R. K. One eye is all you need: Lightweight ensembles for gaze estimation with single encoders. In *14th ACM Symposium on Eye Tracking Research and Applications* (2022).

[14] Strupczewski, A., Czuprynski, B., Naruniec, J., and Mucha, K. S. Geometric eye gaze tracking. In *VISIGRAPP* (2016).

[15] Tan, M., and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning, 2019* (2019).

[16] Wu, Y.-L., Yeh, C.-T., Hung, W.-C., and Tang, C.-Y. Gaze direction estimation using support vector machine with active appearance model.

[17] Zhang, X., Park, S., Beeler, T., Bradley, D., Tang, S., and Hilliges, O. Eth-xgaze: A large scale dataset for gaze estimation under extreme head pose and gaze variation. In *European Conference on Computer Vision* (2020).

[18] Zhang, X., Sugano, Y., and Bulling, A. Revisiting data normalization for appearance-based gaze estimation. In *Proc. International Symposium on Eye Tracking Research and Applications (ETRA)* (2018), pp. 12:1–12:9.

[19] Zhang, X., Sugano, Y., Fritz, M., and Bulling, A. Appearance-based gaze estimation in the wild. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015), pp. 4511–4520.

[20] Zhao, M., Gersch, T. M., Schnitzer, B. S., Dosher, B. A., and Kowler, E. Eye movements and attention: the role of pre-saccadic shifts of attention in perception, memory and the control of saccades. *Vision Research 74* (2012), 40–60.

[21] Zhu, X., Lei, Z., Liu, X., Shi, H., and Li, S. Z. Face alignment across large poses: A 3d solution. *CoRR abs/1511.07212* (2015).

[22] Émile Javal, L. *Essai sur la phisiologie de la lecture.* 1879, p. 81. Annales d'oculistique.