Doble Grado en
Matemáticas y Estadística

TRABAJO FIN DE GRADO

Tecnologías biométricas aplicadas a la ciberseguridad

Autora: Rocío Ruiz Montaño

Tutor: Luis Valencia Cabrera

Sevilla, Junio 2022

Índice general

Prólogo

El trabajo de fin de grado realizado tiene el título de “Tecnologías biométricas aplicadas a la ciberseguridad”. Este estudio, aparte de explicar los distintos sistemas biométricos que existen y que se han ido analizando en profundidad a través de diversas lecturas bibliográficas, se centra en el reconocimiento facial a través de redes neuronales convolucionales.

En primer lugar, me gustaría agradecer a mis padres y a mi hermana su continuo apoyo durante toda la carrera y la confianza depositada en mí.

También agradecer a todos mis amigos y compañeros que he conocido durante esta etapa y que no hubiera sido lo mismo sin ellos.

Finalmente, me gustaría agradecer a mi tutor, Luis, quien me ha aconsejado y dado orientaciones para completar este proyecto confiando en mí.

Resumen

Rodeados de una sociedad en constante evolución y con un gran afán de desarrollo tecnológico, la Inteligencia Artificial (IA) se posiciona como una de las áreas de mayor impacto en la actualidad. Una de las aplicaciones más útiles de IA es la verificación de la identidad a través de perfiles biométricos y conductuales. En este proyecto veremos algunos de los sistemas biométricos que existen, los tipos y la estructura del procedimiento biométrico.

En este contexto, se encuentra el reconocimiento facial, una herramienta que cada vez se utiliza más en cualquier ámbito o situación; desde el desbloqueo de dispositivos móviles, a través de cientos de aplicaciones, hasta su uso como medida de seguridad implementada por algunos gobiernos.

Este proyecto se basa en el análisis, tanto teórico como experimental, de un sistema de reconocimiento facial a partir de una base de datos. El código se implementa en el lenguaje de programación R junto a la librería de machine learning y visión por computador OpenCV y, la API de redes neuronales de alto nivel Keras.

El proceso de identificación de una cara se divide en dos subprocesos consecutivos. En cuanto al primer subproceso, la detección facial, el método utilizado se centra en el algoritmo Viola-Jones basado en Haar Cascades. Para el segundo subproceso, el reconocimiento facial, el método utilizado para ello han sido las redes neuronales convolucionales, de las cuales se ha implementado su código y se ha realizado un estudio detallado. Cabe mencionar que existen también otros métodos interesantes como son *Fisherfaces*, *Eigenfaces* y LBPH (*Local Binary Pattern Histogram*).

De esta forma, se ha realizado una comparación entre las diferentes redes que hemos diseñado para determinar su comportamiento, estudiando sus tasas de éxito, tratando de analizar la influencia de diversos factores y tratando de buscar una mejora progresiva hasta dar con la mejor solución que hemos podido encontrar, apoyándonos además en técnicas adicionales como *transfer learning* o *data augmentation*.

Abstract

Surrounded by a society in a continuous evolution and with a great eagerness for technological development, Artificial Intelligence (AI) is positioned as one of the areas of greatest impact nowadays. One of the most useful applications of AI is verifying the identity through biometric and behavioural profiles. In this project we will study some of the existing biometric systems, their types and the structure of the biometric procedure.

In this context, we find facial recognition, a tool that is increasingly used in any field or situation; from the unlocking of mobile devices, through hundreds of applications, to its use as a security measure implemented by some governments.

This project is based on the analysis, theoretical and experimental, of a facial recognition system based on a database. The code is implemented in the R programming language with the machine learning and computer vision library OpenCV and the high-level neural network API Keras.

The process of identifying a face is divided into two consecutive sub-processes. For the first sub-process, face detection, the method used is based on the Viola-Jones algorithm using Haar Cascades. For the second sub-process, face recognition, the method used has been convolutional neural networks, of which its code has been implemented and detailed. It is worth mentioning that there are also other interesting methods such as *Fisherfaces*, *Eigenfaces* and LBPH (*Local Binary Pattern Histogram*).

In this way, a comparison has been made between the different networks that we have designed to determine their behaviour, studying their success rates, trying to analyse the influence of various factors and seeking a progressive improvement until we have found the best solution we have been able to find, also relying on additional techniques such as *transfer learning* or *data augmentation*.

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo vamos a ver por qué ha surgido este estudio, cuáles son los objetivos que nos hemos propuesto y la estructura que hemos seguido en este trabajo.

1.1. Motivación del proyecto

Actualmente el uso de biometrías ha causado gran interés y está siendo de vital importancia ya que las usamos en nuestro día a día, como es el caso de reconocer delincuentes buscados a su paso por aeropuertos y distintas aduanas, desbloquear un móvil con la cara o huella dactilar, reconocimiento de firmas manuscritas, reconocimiento de voz para activar aparatos electrónicos y un largo etcétera.

Siempre me he preguntado cómo todo esto era posible y he querido investigar en este proyecto sobre ello. Quería aprender los distintos sistemas biométricos que existen y poner en práctica alguno de ellos. Debido a la confidencialidad que deben tener los datos biométricos, es bastante complejo encontrar un conjunto de dichos datos para poder realizar algún modelo de clasificación. Es por ello que me centré en el reconocimiento facial, ya que las imágenes de personas famosas pueden ser utilizadas con fines científicos de manera legal.

El reconocimiento facial también es de interés para empresas. Es conocido que Apple ha integrado en el iPhone X un sistema de reconocimiento facial para desbloquear el dispositivo o identificar al usuario; Google tiene un sistema capaz de reconocer caras en fotos almacenadas y Facebook ha implementado un sistema que reconoce a personas en fotos en las que no están etiquetadas.

Todo esto ha suscitado interés para la realización de este estudio centrándose, como se ha comentado, en el reconocimiento facial.

1.2. Objetivos

En este proyecto se realizará un estudio de algunas de las tecnologías utilizadas para desarrollar uno de los sistemas biométricos más avanzados actualmente en desarrollo, el sistema de reconocimiento facial.

El presente documento tiene como objetivos:

– Conocer los distintos tipos de biometrías, así como sus características.

– Entender la estructura del reconocimiento de patrones.

– Estudiar el procedimiento de reconocimiento facial, junto con sus problemas y dificultades asociadas.

– Ser capaces de poner en práctica el proceso completo de un sistema de reconocimiento facial, diseñando un clasificador para el reconocimiento de personas.

– Conocer y entender el funcionamiento de las redes neuronales convolucionales y cómo implementarlas en R para nuestro estudio. Se pretende empezar con modelos más sencillos y poco a poco construir modelos más complejos hasta dar con una buena solución.

– Evaluar el funcionamiento de los modelos que se realicen para la predicción de rostros que no hayan sido entrenados en los modelos.

– Realizar una interpretación de cómo podemos implementar el mejor modelo que diseñemos en sistemas de seguridad, como es el caso de reconocer personas que sean posibles delincuentes en aeropuertos.

1.3. Estructura del trabajo

El presente estudio se ha organizado en un total de seis capítulos:

1. Introducción: recoge la motivación, los objetivos y estructura de este trabajo.
2. Estado del arte: capítulo en el que se recogen los distintos sistemas biométricos que existen y cuál es la estructura del procedimiento biométrico.
3. Estado de la tecnología del reconocimiento facial: en este capítulo podemos encontrar, como su nombre indica, la tecnología del reconocimiento facial, pasando por cada una de las etapas que la componen. Asimismo se explican distintas técnicas usadas en cada paso. Los dos puntos más destacables son el algoritmo de Viola-Jones para la detección facial y las redes neuronales convolucionales, que aparte de extraer características son capaces de construir un modelo que clasifique las caras.
4. Diseño de un clasificador para reconocimiento de personas: en este capítulo abordaremos este problema en un conjunto de imágenes. En primer lugar, explicaremos cómo se realiza la detección facial mediante la librería OpenCV. Posteriormente prepararemos los datos para poder diseñar redes con Keras. Se realizarán distintos modelos de redes neuronales (incluyendo redes neuronales convolucionales) y se verá la eficacia de cada uno ellos a la hora de clasificar nuevas imágenes (conjunto *test*).
5. Herramientas software: se explican las herramientas en las cuales se ha apoyado el desarrollo de este trabajo.
6. Conclusiones: se presentarán los conceptos más importantes que han resultado este estudio y las posibles líneas de trabajo futuro.

Capítulo 2

Estado del arte

2.1. Introducción a la biometría

La biometría es la aplicación de técnicas estadísticas y matemáticas sobre las características físicas de los individuos para identificarlos y autentificarlos. Ello requiere emplear tecnologías informáticas para implementar los sistemas de identificación y autentificación.

En función de las características usadas para la identificación, se pueden establecer dos grandes tipos, dependiendo de si se fijan los aspectos físicos o se fijan en los aspectos vinculados a la conducta. Estos tipos son la *Biometría estática,* para referirnos al estudio de características físicas y, la *Biometría dinámica,* para el conjunto de características de conducta [30].

Dentro de la Biometría estática, algunas de las características que podemos encontrar son:

- **Huella dactilar**: es el método de identificación más antiguo. Se captura la huella de un usuario y se contrasta contra una huella almacenada previamente. Las huellas tienen pequeños surcos en la piel en forma de líneas que forman patrones únicos. Estos patrones pueden ser continuos o discontinuos y son la base del reconocimiento dactilar.
- **Características del ojo (retina e iris)**: el iris posee un patrón muy complejo que puede contener muchas características únicas como pueden ser surcos, anillos y coronas. El escaneo de iris es menos intrusivo que el de la retina, ya que éste es observable a varios metros de distancia. Además, los cambios en el iris debidos a la luz pueden proporcionar una verificación secundaria que garantiza que el iris es de una persona que se encuentra viva. El reconocimiento de retina crea una firma con el ojo a partir de las características vasculares de ésta, que son únicas en cada individuo y en cada ojo. Es uno de los métodos biométricos más seguros, pero como hemos dicho anteriormente es un método más intrusivo.
- **Geometría de la mano**: utiliza la forma geométrica de la mano para identificar a un usuario. Es un método cuestionable, ya que estas características no describen de manera suficientemente unívoca a un individuo.

- **Características de la cara/ reconocimiento facial**: método por el cual una cara humana es capturada para, posteriormente, ser utilizada con el fin de identificar a un individuo. Estos métodos miden y analizan la estructura y la forma de la cara, sus proporciones (distancia entre los ojos, nariz y boca; el contorno de las cuencas de los ojos; los lados de la boca; la localización de la nariz y la boca; etc.). En la verificación de identidad, el usuario debe situarse frente a una cámara. Para prevenir un posible ataque mediante el uso de una imagen en lugar del individuo real, estos sistemas, en ocasiones, esperan el parpadeo de los ojos, solicitan que el usuario sonría, haga algún gesto o utilizan otra técnica con la que se comprueba la temperatura de la cara: la termografía facial.

Por su parte, en la Biometría dinámica podemos encontrar:

- **Escritura manuscrita**: la firma manuscrita es un rasgo biométrico que cuenta con una alta aceptación social por parte de los usuarios, habiendo sido utilizada durante siglos como método de autenticación de documentos legales y de transacciones y contratos civiles. Existe gran dificultad para modelizar la firma, por lo que se necesita un gran volumen de datos y existe gran escasez de los mismos (debido a la alta variabilidad intra-usuario). Además, es difícil detectar firmas falsas, dado que el grado de bondad de un impostor no se puede predecir (baja variabilidad inter-usuario). El reconocimiento de firma se puede realizar de manera estática (cuando la firma ya se ha realizado) o de manera dinámica (en el momento de realización de la firma).
- **Voz**: los sistemas de reconocimiento de voz utilizan, en la mayoría de los casos, un micrófono para almacenar la voz enunciando un texto conocido. Esta grabación se utiliza para extraer patrones únicos y formar una plantilla de voz, que puede ser de dos tipos: estocástica (requieren coincidencia probabilística) y plantilla de modelo (utilizan patrones deterministas). Estos sistemas tienen un grado de aceptación alto dado que no son intrusivos y su precisión es bastante elevada.
- **Tecleo**: es una técnica desarrollada para mejorar la identificación mediante texto y está basada en la extracción de características durante la pulsación de teclas (por ejemplo: tiempo de pulsación de tecla o intervalo de tiempo entre dos pulsaciones).
- **Gesto y movimiento corporal**: aquí podemos encontrar al reconocimiento del paso, que es una técnica de reconocimiento biométrico que utiliza una secuencia de imágenes a partir de un vídeo de una persona caminando para identificar sus movimientos. En principio, no es un método que permita realizar demasiada distinción entre individuos y, además, el paso es un comportamiento que puede verse modificado en el tiempo debido a los cambios físicos que se puede producir en el cuerpo, por ejemplo. Pese a no ser una técnica intrusiva, su uso no está demasiado extendido debido a su poca eficacia.

Estas últimas, son por naturaleza fáciles de suplantar, ya que no se tratan de características intrínsecas del cuerpo humano. Es por ello que cuando se habla de biometría se utilizan las características físicas estáticas, aquellas únicas, inamovibles y propias de cada ser vivo.

Para que las características físicas y conductuales sean elementos de identificación deben cumplir:

- Universalidad: cada persona presenta la característica.
- Singularidad: dos personas cualesquiera tienen que ser diferenciadas una de otra con la característica.
- Estabilidad: la característica tiene que ser lo suficientemente estable a lo largo del tiempo y condiciones ambientales.
- Cuantificable: puede ser medida cuantitativamente.
- Aceptabilidad: la característica debe tener un nivel de aceptación para que sea considerada parte de un sistema de identificación biométrico.
- Rendimiento: debe haber un elevado nivel de exactitud para que la característica sea aceptada.
- Usurpación: para no caer en fraudes.

El objetivo final de las características enunciadas es que se pueda conseguir la identificación y verificación de una persona. Sin embargo, no existe ningún rasgo biométrico que cumpla todos los requisitos.

2.1.1. Historia de la Biometría

Vamos a hacer una breve visión de cómo empezó y se ha ido desarrollando la biometría a lo largo de los años.

Se data al siglo VIII como el primer momento en el que existe referencia acerca del uso de una característica biométrica con fines identificativos, fecha en la que se encuentran en China huellas dactilares en sitios como esculturas o documentos. Siglos más tarde, en el año 1000, Quintiliano utilizó las huellas dejadas por las palmas de unas manos ensangrentadas para resolver un crimen. Posteriormente, en 1686, Marceló Malpigio hizo el primer estudio sistemático de las huellas dactilares.

En 1856, sir William Herschel fue el primero en implantar la huella del pulgar como método de identificación en documentos para personas analfabetas. Años más tarde, en 1880, Henry Faulds publicó un artículo en la revista Nature en el que planteaba que las huellas dactilares encontradas en la escena de un crimen podían identificar al culpable. De hecho, con esta técnica Faulds consiguió inculpar a un inocente y encontrar al culpable de un robo.

En 1941, Murray Hill comenzó el estudio de la identificación por voz. En 1986 sir Alec Jeffreys usó por primera vez el ADN para identificar a un asesino.

El uso de la biometría como una tecnología moderna y de aplicación a fines comerciales tiene su arranque en los años 70, con la implantación de los primeros sistemas de reconocimiento automático de huellas dactilares. A partir de mediados y finales de los noventa el interés ha ido creciendo y, además, han ido aumentando los presupuestos de financiación para la investigación y desarrollo vinculados a la Biometría.

2.1.2. Estructura del procedimiento biométrico

Existen diferentes modelos acerca de la estructura del procedimiento biométrico. Nos basamos en la estructura propuesta por James L. Wayman, director del U.S. National Biometric Test Center de la San Jose State University [33]. Este esquema propuesto (figura 2.1) sigue la siguiente estructura con cinco subsistemas:

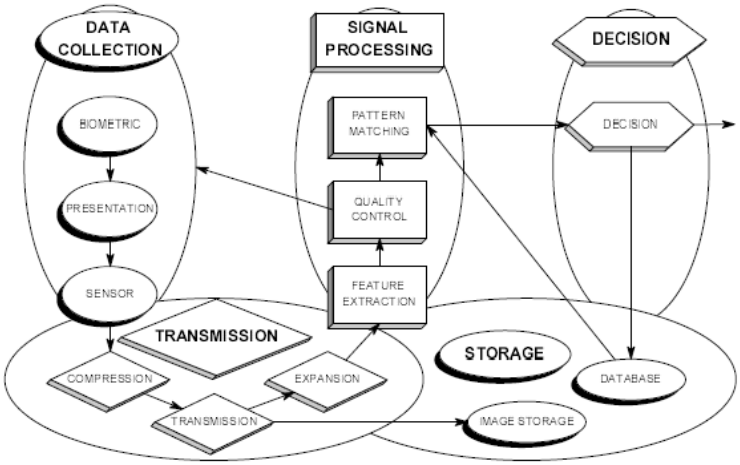

Figura 2.1: Esquema general de un sistema biométrico propuesto por Wayman. Fuente: Wayman, James L (2001) [33].

1. Recopilación de datos: los sistemas biométricos comienzan con la medición de una característica conductual o fisiológica. La clave de todos los sistemas es la suposición de que la información biométrica medida (de alguna característica) es distintiva entre individuos y repetible a lo largo del tiempo para el mismo individuo. La característica del usuario debe presentarse a un sensor. Hay que indicar la medida biométrica, la forma en que se presenta dicha medida y las características técnicas del sensor. Es muy importante estandarizar los datos para garantizar la correcta recogida de datos.

2. Transmisión de datos: algunos sistemas biométricos recopilan datos en un lugar pero los almacenan y/o procesan en otro. Dichos sistemas requieren transmisión de datos. Si existen una gran cantidad de datos involucrados, puede ser necesaria la compresión antes de la transmisión. En tales casos, los datos comprimidos deben expandirse antes de su uso posterior. Estos procesos de compresión y expansión generalmente causan pérdida de calidad/información. Según la técnica biométrica usada, se utilizan métodos de compresión con mínimo impacto en el subsistema de procesamiento de señales. Además, la transmisión de información puede presentar problemas de incorporación de ruido.

3. Procesamiento de señales: la información que proviene de las señales transportadas por el subsistema anterior es transformada mediante diferentes algoritmos. Estos algoritmos son los encargados de extraer características biométricas presentes en la señal original. Dichas características se consideran las más relevantes a efectos de comparación entre distintas muestras.

4. Almacenamiento de datos: la información registrada durante la fase de inscripción o procedimiento de darse de alta debe almacenarse en forma estructurada para facilitar el procedimiento de indentificación y verificación. Debe ser flexible para permitir que incluya no solo los vectores de los datos biométricos, sino también etiquetas identificativas del usuario del que provienen.
5. Proceso de decisión: el proceso de identificación y verificación finaliza con la medida de un índice de comparación entre las plantillas almacenadas y los datos introducidos por un usuario en un momento cualquiera de acceso al sistema. Este índice permitirá tomar la decisión acerca de si la identificación/verificación es satisfactoria o no.

2.2. Reconocimiento de patrones

El reconocimiento de patrones se encarga de la descripción y clasificación de personas, objetos, animales, señales, representaciones, etcétera.

El objetivo del reconocimiento de patrones es justamente ajustar un sistema para que sea capaz de clasificar señales u objetos de entrada según una clases previamente definidas. El esquema por tanto puede ser el siguiente que se ve en la Figura 2.2:

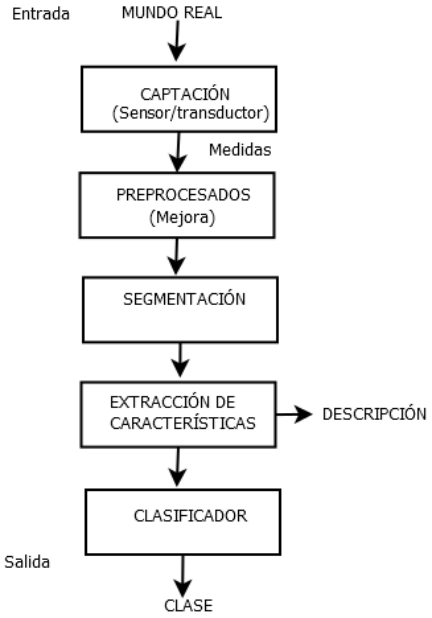

Figura 2.2: Esquema de un sistema de reconocimiento de patrones. Fuente: Elaboración propia.

Para poder clasificar correctamente los datos de entrada, será necesario un proceso de aprendizaje en el cuál se creará un modelo a partir de una secuencia de entrenamiento. Posteriormente, se pondrá a prueba con una secuencia test.

Existen fundamentalmente dos tipos de aprendizaje más ampliamente extendidos y de un uso en una gran variedad de situaciones: el aprendizaje supervisado y el no supervisado. Hay otros tipos, como el aprendizaje por refuerzo, que permite obtener excelentes resultados en determinados contextos, pero que quedan más alejados del objetivo de este trabajo.

- Aprendizaje supervisado: los algoritmos están basados en el entrenamiento de los datos desde un conjunto de datos inicial que incluye una correcta clasificación o valor para cada caso. Este entrenamiento permite que este tipo de modelos puedan categorizar nuevos datos sin conocer su clasificación correcta. Para este tipo de enfoque, hay que asumir la existencia de un supervisor que asigne el valor correcto para cada uno de los ejemplos del conjunto de datos inicial. Algunos de los algoritmos más populares de Machine Learning en esta categoría son la regresión lineal, la regresión logística, *support vector machines*, *decision trees*, *random forest* y redes neuronales.
- Aprendizaje no supervisado: los algoritmos están basados en identificar patrones ocultos en datos de entrada no etiquetados y es por ello que el algoritmo intentará clasificar la información por sí mismo. Algunos de los algoritmos más conocidos de esta categoría son *clustering* (de partición estricta, con algoritmos como como k-means, o jerárquico) o *principal component analysis* (PCA).

2.3. Extracción de características

En un reconocedor de patrones uno de los aspectos más importantes a tener en cuenta es el tipo de parametrización que se realiza sobre la señal de entrada.

En el caso de imágenes se puede utilizar como vector de características la propia imagen ya que podemos verlo como un mapa de bits.

En cada tipo de reconocimiento se pueden emplear unas técnicas u otras, pero en todas ellas la parametrización que se elija proporciona múltiples ventajas ya que se puede reducir el número de datos a procesar y por consiguiente obtener un conjunto de datos con mayor facilidad a discriminar.

2.4. Selección de características

La selección de características no es un problema trivial, ya que trabajar con casos de menor dimensión puede resultar más sencillo y tener menor coste computacional pero, por contra, puede haber una menor consistencia en los resultados.

2.4.1. Selección secuencial hacia adelante

También conocida como *Sequential Forward Selection*. Es un procedimiento para la selección de variables que consiste en el que las variables se introducen secuencialmente en el modelo. La primera variable en ser introducida en la ecuación es aquella con mayor correlación (sea positiva o negativa) con la variable a estudiar. La variable se introducirá en la ecuación si se satisface el criterio de entrada. Así, se van introduciendo en la ecuación de manera sucesiva las variables que tengan la mayor correlación parcial. El procedimiento acaba cuando no hay variables que cumplan el criterio de entrada.

2.4.2. Eliminación secuencial hacia atrás

También conocido como *Sequential Backward Elimination*, este procedimiento es al contrario que el anterior. Se parte con todas las variables en la ecuación y se van eliminando secuencialmente. La primera variable a eliminar es aquella que tenga la menor correlación parcial con la variable dependiente. Después de eliminar la primera variable, la siguiente variable a eliminar es la siguiente que tenga menor correlación con la ecuación que tenemos hasta el momento. El procedimiento termina cuando no hay variables en la ecuación que satisfagan los criterios de eliminación.

2.5. Proyección de los vectores de características

Los métodos vistos en el apartado anterior son computacionalmente más costosos que los que vamos a ver a continuación. Existen otros métodos como el análisis en componentes principales (*Principal Component Analysis, PCA*) o el análisis discriminante lineal (*Linear Discriminant Analysis, LDA*). Estos métodos pueden considerarse como una reducción de la dimensionalidad no supervisados, puesto que no tienen en cuenta a qué clase pertenece cada uno de los vectores de la secuencia de entrenamiento originales sino que se tratan todos conjuntamente.

Análisis en componentes principales (PCA)

El análisis en componentes principales o PCA es un método estadístico que sirve para reducir la dimensionalidad del problema, es decir, reducir el número de variables mientras se minimice el error cuadrático medio. Esta técnica es utilizada para describir un conjunto de datos en nuevas componentes (variables) no correlacionadas entre sí.

El algoritmo PCA busca una proyección de las variables para que los datos queden mejor representados y así conseguir que dado un conjunto de datos de variables correlacionadas entre sí, se transforme en un nuevo conjunto de valores sin correlación lineal, y es lo que se denomina componentes principales.

Análisis discriminante lineal (LDA)

Es una técnica de análisis multivariante que procura encontrar relaciones lineales entre las variables continuas que mejor discriminen en los grupos categóricos previamente definidos. En 1936, esta técnica fue introducida por Fisher en el primer tratamiento moderno de problemas separatorios. Su propósito era analizar si existen diferencias significativas entre grupos de objetos respecto a un conjunto de variables medidas sobre los mismos y, en el caso de que existan, explicar en qué sentido se dan y proporcionar procedimientos de clasificación sistemática de nuevas observaciones de origen desconocido en uno de los grupos analizados. La variable dependiente de clasificación es una variable no métrica, mientras que las variables independientes deben ser métricas.

Reducción de la dimensionalidad mediante redes neuronales

Para reducir la dimensión de los vectores de características, reducir el número de variables, se puede realizar mediante una red neuronal. Para que se produzca dicha reducción de dimensionalidad de los vectores de P a d, las capas de entrada y salida deben tener P neuronas y la capa oculta intermedia deberá tener d neuronas. Durante el entrenamiento se da a la red neuronal vectores de características y se forzará el aprendizaje a la salida de cada vector. Una vez se ha entrenado la red, se presentan los vectores a comprimir y, se obtiene la salida de la capa oculta intermedia. Si la red neuronal tiene solo una capa oculta el sistema es equivalente al análisis PCA. En cambio, al usar más capas ocultas, se obtiene una compresión no lineal, lo cuál permite una mayor flexibilidad.

Explicaremos en el próximo capítulo con más detalle las redes neuronales.

Detección de objetos por correlación

Es posible detectar si un objeto de una imagen está presente en otra imagen mediante la correlación bidimensional de dichas imágenes. De esta forma, puede llevarse a cabo de manera simultánea la segmentación del objeto de la imagen y su reconocimiento.

Capítulo 3

Estado de la tecnología del reconocimiento facial

Desde los inicios de la visión por computador ha sido estudiado el reconocimiento facial. Es una rama de la biometría que ha causado gran atractivo ya que, como vimos en el capítulo anterior, es un método poco intrusivo. Por contra, hay otras ramas más fiables como son las huellas dactilares y escaneo del iris pero que son más intrusivas.

En los últimos años, ha habido un auge con las tecnologías del reconocimiento de rostros que han permitido crear algoritmos más complejos y desarrollar sistemas de reconocimiento facial a tiempo real. A pesar de ello, el reconocimiento de rostros a través de un computador tiene muchas limitaciones y según las condiciones esta tarea es más rápida con el ser humano.

3.1. Introducción histórica

Uno de los pioneros del reconocimiento facial fue Woodrow Wilson Bledsoe [7], quien a mediados de los años 60 trabajó en un sistema para clasificar los rasgos del rostro humano a través de la tabla RAND. Este proceso era muy manual y consistía en digitalizar las caras mediante un lápiz óptico para hallar las coordenadas de los distintos rasgos faciales como son los ojos, nariz y boca. Estos datos eran registrados junto con el nombre de la persona en una base de datos. Finalmente, con una fotografía de una cara desconocida, el sistema usaría un método basado en distancias euclídeas para obtener la imagen en la base de datos que más se aproxime a la fotografía en cuestión.

Una década después llegarían Goldstein, Harmon y Lesk, que detallaron estas características faciales e iniciaron una mejora en la precisión del reconocimiento facial. En 1988 ocurre un acontecimiento importante, L. Sirobich y M. Kirby utilizan el álgebra lineal para el reconocimiento facial, en concreto, aplicaron el análisis de componentes principales (PCA) y del que posteriormente, en 1991, surgiría el método de Eigenfaces gracias a Turk y Pentland.

En 2001, surge el algoritmo de Viola-Jones para detectar objetos dentro de imágenes pero enseguida fue utilizado para la detección de rostros obteniendo un gran éxito. En los últimos años, aparecen las redes neuronales convolucionales (*Convolutional Neural Networks*), que hasta hoy suponen el mejor modo de detectar rostros.

Actualmente, el reconocimiento facial es usado en diversas áreas como seguridad, entretenimiento, comercio y marketing.

3.2. Procedimiento de reconocimiento facial

El problema del reconocimiento facial se basa en una identificación de patrones visuales. En general, podemos decir que un sistema de reconocimiento facial comprende cuatro etapas: detección, preprocesado, extracción de características y comparación para su clasificación.

En primer lugar, en la detección facial se pretende localizar el rostro en una imagen. Posteriormente, se debe realizar un preprocesado de la imagen, alineando, normalizando rostros, intentando quitar ruido, etcétera. Posteriormente se pone en marcha la extracción de características; es un paso muy significativo, ya que se encargará de extraer información que permita distinguir unas caras de otras. Finalmente, el vector de características que hemos obtenido será comparado con la base de datos para decidir de quién es ese rostro. Para esta comparación en la base de datos se construye un modelo que clasifique a la persona de manera automática.

Cada una de las etapas es muy valiosa, ya que el resultado final dependerá de la precisión con la que se han adquirido las características, que depende tanto de la detección del rostro como de la normalización de la imagen.

En las siguientes secciones se estudiará cada una de estas fases que componen el funcionamiento de un sistema de reconocimiento facial.

Básicamente el esquema del procedimiento a seguir es el que podíamos ver en la figura 2.2.

3.3. Detección facial

La detección facial es el proceso mediante el cual el sistema localiza la posición de los rostros de personas en una imagen o fotograma.

Antiguamente, en el reconocimiento facial, se tenía que llevar a cabo de forma manual localización de la cara en una imagen. Actualmente existen muchas técnicas para la detección automática.

Existen una infinidad de métodos de detección facial, pero nosotros hablaremos de tres de ellos.

Es imprescindible que un detector de caras nos dé la posición y tamaño de una caja de inclusión en la que se encuentren los rasgos faciales como son los ojos, nariz y boca. Más aún, sería una gran aportación que nos indique la fiabilidad de esa detección.

A la hora de evaluar cómo de bueno es un detector de caras deberemos fijarnos en las siguientes medidas:

- Tasa de detección: es el porcentaje de caras correctamente detectadas respecto del total de caras de la imagen.

- Tasa de falsos positivos: indica el número de regiones que se han marcado como caras dónde realmente no hay caras.
- Tasa de falsos negativos: indica el número de caras que no han sido correctamente detectadas. Es decir, que han sido clasificadas negativamente.

3.3.1. Algoritmo de Viola-Jones

Este algoritmo es el primero que supuso un salto de calidad importante en la detección facial. Se describe en el artículo *"Rapid Object Detection using a Boosted Cascade of Simple Features"* [32]. Fue propuesto por Paul Viola y Michael J.Jones, cuya aportación principal fue la de una cascada de clasificadores muy sencillos ejecutándose uno detrás de otro. Cada clasificador de la cascada se entrena con el algoritmo de *boosting* AdaBoost.

Su trabajo se distingue por la contribución de tres puntos clave.

1. El primer punto clave es la introducción del concepto de imagen integral, que a grandes rasgos consiste en analizar la imagen mediante segmentos rectangulares en vez de hacerlo píxel a píxel, por lo que se reduce bastante el tiempo de procesado (esto hace que sea un algoritmo rápido). La imagen integral en el punto (x, y) contiene la suma de los valores de cada uno de los píxeles que se encuentran por encima y a la izquierda de (x, y):

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y')$$

donde $ii(x, y)$ es la imagen integral y $i(x, y)$ es la imagen original. Usando el siguiente par de recurrencias:

$$s(x, y) = s(x, y - 1) + i(x, y)$$

$$ii(x, y) = ii(x - 1, y) + s(x, y)$$

(donde $s(x, y)$ es la suma acumulativa de las filas, $s(x, -1) = 0$, y $ii(-1, y) = 0$) la imagen integral se puede calcular en una pasada sobre la imagen original.

Esta aportación se basa también en el trabajo "A General Framework for Object Detection" realizado por Papageorgiu y otros en 1998 [23]. En él se explica cómo, a partir de la secuencia de funciones Haar wavelets (cascadas de Haar), se obtienen los descriptores Haar rectangulares, de modo que si se combinan con la imagen integral, es posible procesarlos de manera mucho más rápida. Estos descriptores de Haar se utilizan principalmente en la detección de objetos.

Cada uno de estos descriptores toma un único valor, que es calculado al restar los píxeles de la imagen que están en la zona blanca y al sumar los que están en la zona oscura (figura 3.1). En otras palabras, se pretende hacer una búsqueda de los bits más oscuros respecto a otros para encontrar las zonas de mayor contraste de una cara. Inicialmente, los descriptores Haar tenían una medida fija, pero al introducir el concepto de imagen integral, se pueden calcular características similares a Harr de cualquier tamaño, con los que se puede obtener el contorno de un rostro.

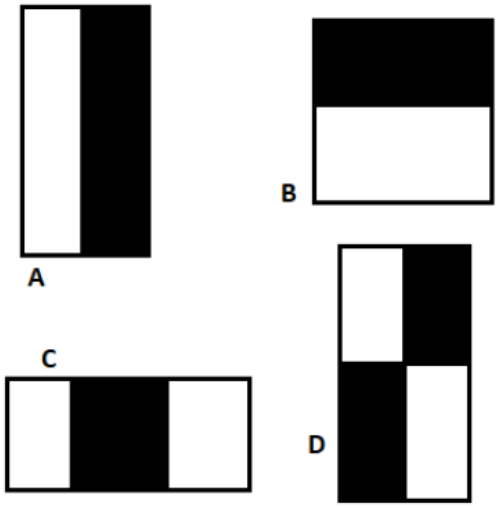

Figura 3.1: Descriptores Haar. A, B: Descriptores de dos rectángulos; su valor es la resta entre la suma de píxeles de cada rectángulo. C: Descriptor de tres rectángulos; su cálculo se realiza restando los rectángulos exteriores al rectángulo central. D: Descriptor de cuatro rectángulos; se calcula la diferencia entre dos pares de rectángulos diagonales. Fuente: Viola and Jones (2001) [32].

Las cascadas de Haar permiten desechar una gran parte de las regiones de una imagen para centrarse en las zonas en las que es más posible que haya algún rostro.

2. La segunda contribución consiste en un método para elaborar un clasificador seleccionando una cantidad pequeña de descriptores rectangulares utilizando AdaBoost [28]. Cualquier imagen puede tener tantos descriptores Haar como píxeles, o incluso muchísimos más ya que existen una gran cantidad de combinaciones. Por ejemplo, en una imagen de dimensiones 24 x 24 píxeles existen más de 180.000 combinaciones de descriptores Haar. Para evitar esto y así garantizar un procesado rápido, mediante el clasificador, compuesto de clasificadores simples (uno por cada descriptor Haar distinto), se excluyen la mayoría de los descriptores disponibles, dejando solamente los que afectan a las zonas más útiles para la identificación de un rostro.

   Como hemos explicado, para cada descriptor, se crea un clasificador simple que, mediante una función de clasificación, determina el umbral óptimo que clasifique el menor número de imágenes de forma errónea. De esta forma, cada clasificador simple, $h_j(x)$, es una función binaria (que toma el valor 1 o 0) construida a partir de un umbral θ_j y de un descriptor Haar $f_j(x)$:

$$h_j(x) = \begin{cases} 1 & \text{si } f_j > \theta_j \\ 0 & \text{otros} \end{cases}$$

   donde j representa un descriptor y x es una sección de la imagen 24×24 píxeles.

   Este algoritmo está diseñado para seleccionar el descriptor único que mejor separe las muestras negativas y positivas. De esta forma, el primer descriptor Haar rectangular será el más inclusivo, y se basa en el contraste entre los ojos y las mejillas (como se

puede apreciar en la figura 3.2). Los descriptores posteriores serán poco a poco más restrictivos y se irán ajustando a las características faciales.

Figura 3.2: De izquierda a derecha: Imagen utilizada para el test, primer descriptor seleccionado por AdaBoost, segundo descriptor seleccionado por AdaBoost. Fuente: Viola and Jones (2001) [32].

Está demostrado que un clasificador con 200 descriptores rectangulares tiene una tasa de éxito que ronda un 95 %.

3. Por último, la tercera contribución y además, la mayor de ellas, se centra en un método para combinar clasificadores AdaBoost sucesivamente más complejos en una estructura en cascada (figura 3.3) que sirve para aumentar la velocidad de detección al conseguir una focalización en las regiones con más interés, descartando las imágenes que no poseen características propias de una cara.

   En el clasificador inicial se eliminan una gran cantidad de muestras negativas con muy poco procesamiento, y a medida que se avanza se requiere cada vez más tiempo en realizar el proceso. Cuando la imagen llega al final del recorrido de la cascada, significa que se ha detectado una cara. El objetivo de este algoritmo en cascada es el de rechazar tantos negativos como sea posible y en la etapa más temprana posible, lo que hace que en las primeras etapas vaya muy rápido y en las etapas posteriores tarde más.

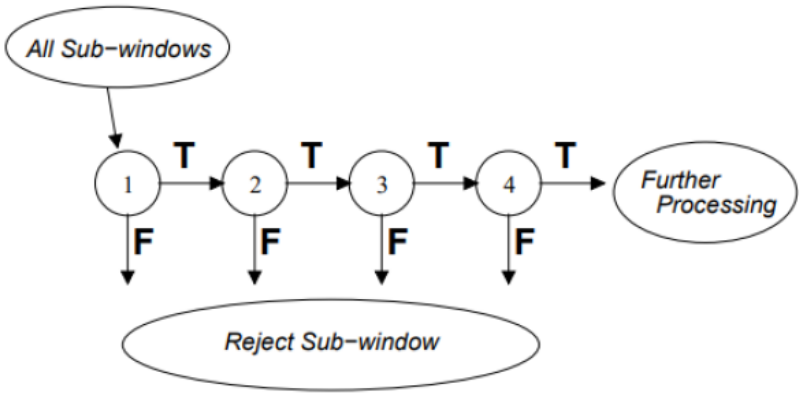

Figura 3.3: Esquema de la detección en cascada. F y T representan Falso (False) y Verdadero (True), respectivamente. Las secciones de la imagen que obtienen T (resultado positivo), pasan a la siguiente fase del procesado, mientras que las que obtienen F, son rechazadas. En cada fase interviene un clasificador simple. Fuente: Viola and Jones (2001) [32].

Dos años más tarde, en el año 2003, Viola y Jones presentaron una extensión a su trabajo inicial [17]. En ella se añadió la funcionalidad de poder detectar caras de perfil o con el rostro girado.

Finalmente, cabe destacar que este algoritmo se trata de un sistema de detección facial que es 15 veces más rápido que cualquier otro sistema desarrollado hasta ese momento y, además, es muy efectivo.

3.3.2. Redes Neuronales Convolucionales

En esta sección explicaremos, en primer lugar, las redes neuronales artificiales, incluyendo su arquitectura y distintos tipos de funciones de activación existentes. Posteriormente, describiremos las redes convolucionales y en qué consisten las capas de convolución y *pooling*.

Las redes neuronales artificiales son sistemas para procesar información, inspirados en la estructura y funcionamiento del cerebro humano. Son capaces de almacenar el conocimiento que han adquirido a través de un proceso de adaptación y, posteriormente, utilizarlo con capacidad de generalización. Es por ello que se dice que simulan características del ser humano como memorizar y asociar hechos.

Las redes neuronales no solo resultan útiles para para los problemas de clasificación, sino que también son capaces de hallar patrones.

Arquitectura de una red neuronal

Es conocido que la arquitectura de una red neuronal se puede definir como un grafo cuyos nodos o neuronas [31], se organizan por niveles o capas [6]. Las neuronas de cada capa, en toda red hacia delante (en definitiva, no recurrente ni perteneciente a otro tipo de red más especial), están conectadas por su entrada con la salida de las neuronas de la capa anterior y estas a su vez, conectan su salida a las entradas de las neuronas de la capa posterior, tal y como podemos ver en la figura 3.4.

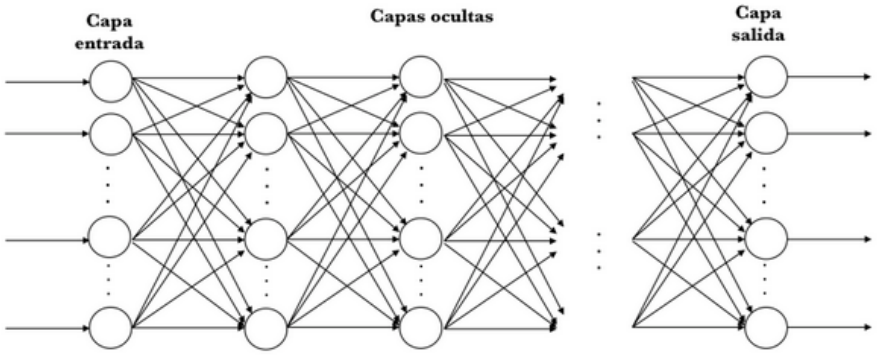

Figura 3.4: Esquema de la arquitectura de una red neuronal. Fuente: Jordi Torres (2018) [31].

La capa de entrada (conocida como *input layer*) recibe los datos de entrada y la capa de salida (*output layer*) devuelve la predicción realizada por la red. Las capas intermedias se denominan capas ocultas (*hidden layers*), y podemos añadir tantas como queramos, consiguiendo así una red de más complejidad y profundidad. No obstante, no siempre conseguimos un mejor resultado incluyendo más capas, pues si nos pasamos podemos llevar a un mayor sobreajuste.

Neurona

Es la unidad básica de procesamiento de una red neuronal que representa el conjunto de soluciones de un problema de regresión lineal. Este problema viene descrito por la suma de los valores de sus conexiones de entrada x_j, ponderadas por una serie de pesos w_j y sumadas a un parámetro de sesgo b, tal y como podemos ver en la figura 3.5.

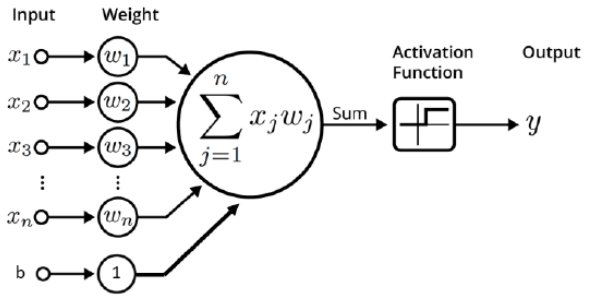

Figura 3.5: Arquitectura de una neurona [6].

Este es el modelo en el que se basa la idea más básica de una neurona: el perceptrón. La solución del perceptrón, presentada por Frank Rosenblatt en 1957, determina un hiperplano que define una región de decisión para el problema de clasificación. Sin embargo, este modelo tiene serias limitaciones porque solo puede resolver problemas lineales, es decir, solo resuelve aquellos problemas de clasificación que se puedan resolver empleando únicamente una recta (en el caso 2D), tal y como se muestra en la figura 3.6 (en general serían problemas con sus clases linealmente separables por un hiperplano).

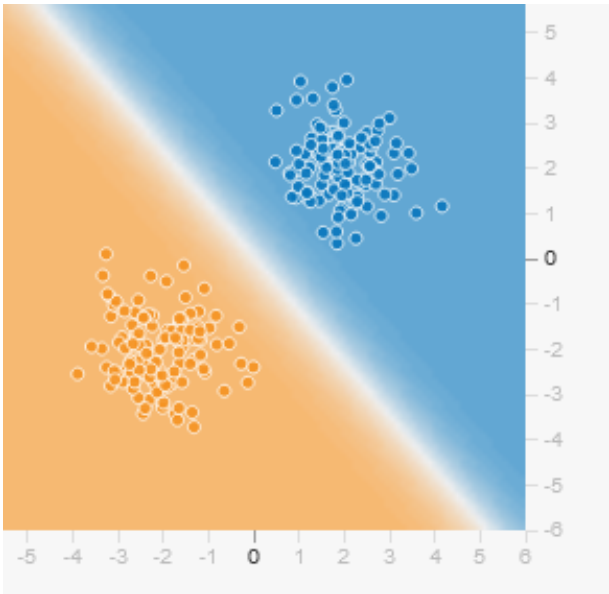

Figura 3.6: Solución a un problema de clasificación utilizando un único perceptrón. Fuente: TensorFlow Playground.

Como hemos dicho, solo es capaz de resolver problemas lineales y es por ello, que el ejemplo que se muestra en la figura 3.7 no es capaz de resolverlo. Para poder resolver dicho problema se podría pensar en utilizar vatios perceptrones para modelar la geometría de la región, sin embargo, esto sería de poca utilidad ya que emplear grupos de perceptrones en serie y paralelo acaban por colapsar dando como resultado un único perceptrón. Es por ello que entra en juego la función de activación.

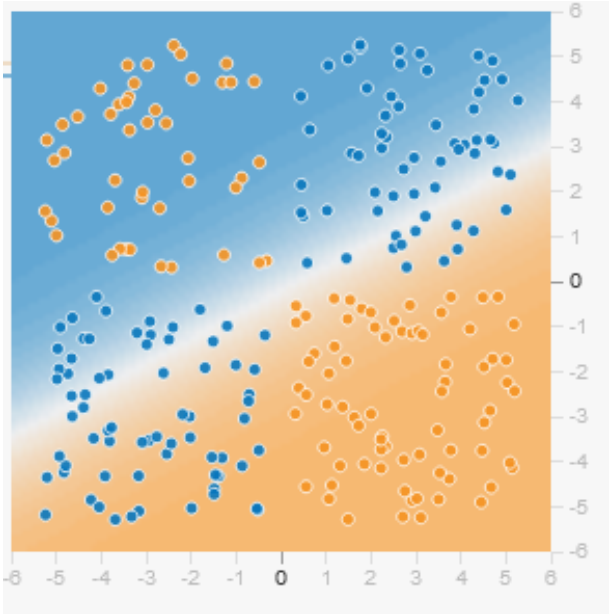

Figura 3.7: Ejemplo visual de un problema de clasificación que no conseguimos resolver utilizando un único perceptrón. Fuente: TensorFlow Playground.

Función de activación

El objetivo de la función de activación es transformar el valor a la salida de una neurona a través de una función, para que esta pase de ser lineal a no lineal y desarrollar así sistemas de mayor complejidad geométrica. Es por ello que se utilizan funciones de activación que consiguen esta serie de no linealidades a la salida de la suma ponderada de la neurona.

Algunas de estas funciones son:

– **Linear**: es la función identidad en la que, en términos prácticos, significa que la señal no cambia.

$$f(x) = x$$

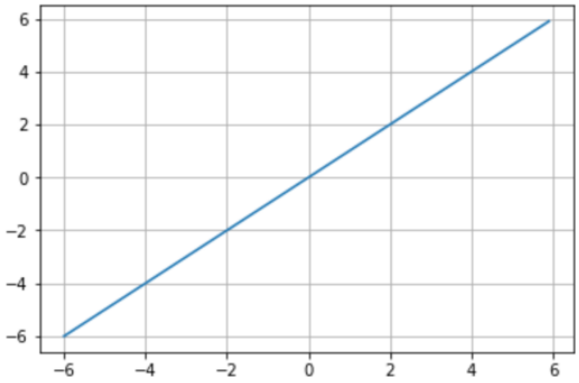

Figura 3.8: Función de activación linear. Fuente: Jordi Torres (2018) [31]

– **Sigmoid**: esta función permite reducir valores extremos o atípicos en datos válidos sin eliminarlos. Convierte variables independientes de rango casi infinito en probabilidades simples entre 0 y 1. La mayor parte de su salida estará muy cerca de los extremos de 0 o 1.

$$f(x) = \frac{1}{1 - e^{-x}}$$

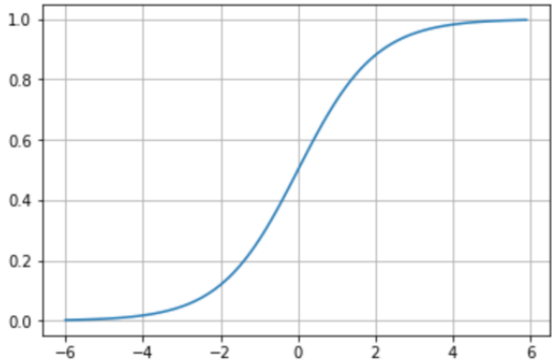

Figura 3.9: Función de activación sigmoid. Fuente: Jordi Torres (2018) [31]

– **Tanh**: representa la relación entre el seno hiperbólico y el coseno hiperbólico: $tanh(x) = sinh(x)/cosh(x)$. A diferencia de la función sigmoid, el rango normalizado de tanh está entre -1 y 1, que es la entrada que le va bien a algunas redes neuronales. La ventaja de tanh es que puede tratar más fácilmente con números negativos.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

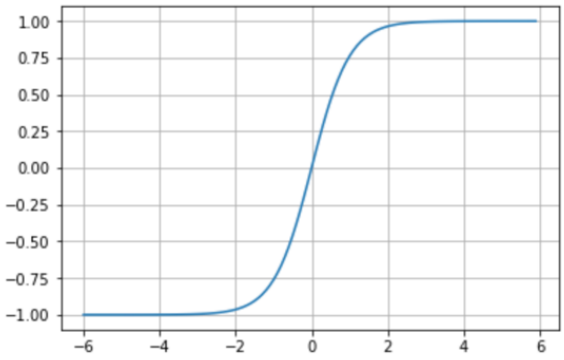

Figura 3.10: Función de activación tanh. Fuente: Jordi Torres (2018) [31]

– **Softmax**: se basa en calcular "las evidencias" de que una determinada imagen (en general, una entrada) pertenece a una clase en particular y luego se convierten estas evidencias en probabilidades de que pertenezca a cada una de las posibles clases. Para medir la evidencia de que una determinada imagen pertenece a una clase en particular, una aproximación consiste en realizar una suma ponderada de la evidencia de pertenencia de cada uno de sus píxeles a esa clase. Una vez se ha calculado la evidencia de pertenencia a cada una de las clases, estas se deben convertir en probabilidades cuya suma de todos sus componentes sume 1. Para ello softmax usa el valor exponencial de las evidencias calculadas y luego las normaliza de modo que sumen uno, formando una distribución de probabilidad. La probabilidad de pertenencia a la clase i es:

$$Softmax_i = \frac{e^{evidencia_i}}{\sum_j e^{evidencia_j}}$$

– **ReLU**: esta función activa un solo nodo si la entrada está por encima de cierto umbral. El comportamiento por defecto y más habitual es que mientras la entrada tenga un valor por debajo de cero, la salida será cero, pero cuando la entrada se eleva por encima, la salida es una relación lineal con la variable de entrada de la forma $f(x) = x$. Actualmente es de las funciones más usadas ya que permite el aprendizaje muy rápido en determinados tipos de redes neuronales.

$$f(x) = max(0, x) = \begin{cases} 0 & si \quad x < 0 \\ x & si \quad x \geq 0 \end{cases}$$

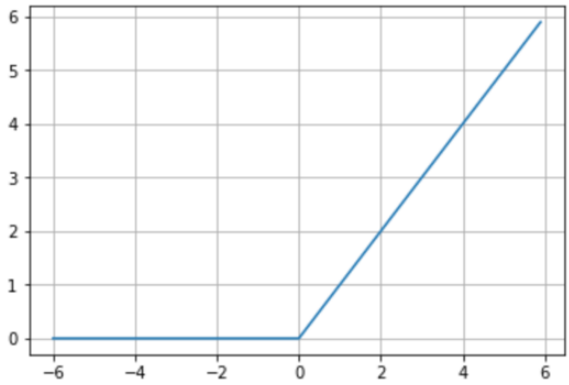

Figura 3.11: Función de activación ReLU. Fuente: Jordi Torres (2018) [31]

Redes neuronales convolucionales

Una red neuronal convolucional (*Convolutional Neural Networks* en inglés, con los acrónimos *CNNs* o *ConvNets*) es un tipo concreto de redes neuronales que suelen entrar dentro de lo que conocemos como Deep Learning, que se utilizaron a finales de la década de 1990, pero se ha vuelto muy popular en los últimos años debido a sus increíbles resultados en el reconocimiento de imágenes, que tienen un profundo impacto en el campo de la visión artificial.

Hasta el momento, la arquitectura de redes neuronales que hemos visto han sido modelos donde cada capa está conectada a cada una de las neuronas de las capas anterior y posterior (*dense layers*). Estos modelos no suelen ser eficaces a la hora de trabajar con imágenes. De hecho, debido a que la imagen consta de píxeles codificados de canales de color, cada píxel de la imagen requiere una neurona, por lo que la cantidad de neuronas y parámetros aumentará en gran medida el costo de carga computacional.

La característica distintiva de las CNN es que asumen explícitamente que la entrada es una imagen, lo que nos permite codificar ciertas propiedades en la estructura para reconocer elementos específicos en la imagen. Para ello, se valen de dos nuevos tipos de capas denominadas de convolución y de *pooling*.

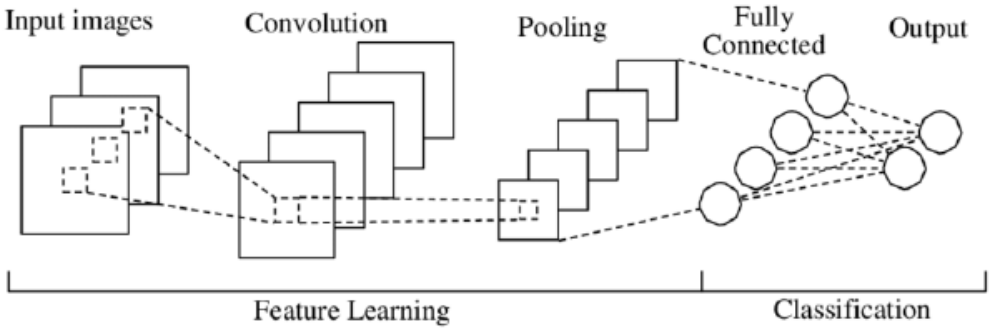

Figura 3.12: Diagrama de una red neuronal convolucional [6].

A través de estas capas, las redes neuronales convolucionales tratan de identificar en primer lugar líneas, bordes o formas similares a rasgos de un rostro como son nariz, ojos u

orejas que haya visto antes. Esto es lo que deben hacer las capas convoluciones de redes neuronales.

Pero identificar estos factores no es suficiente para poder decidir que algo es un rostro. Además, debemos poder determinar cómo se relacionan las partes de la cara entre sí, los tamaños relativos, etc., ya que si no la cara no se parecería a lo que estamos acostumbrados. Este ejemplo que vemos en la figura 3.13 [9], nos da una idea intuitiva de lo que van aprendiendo las capas.

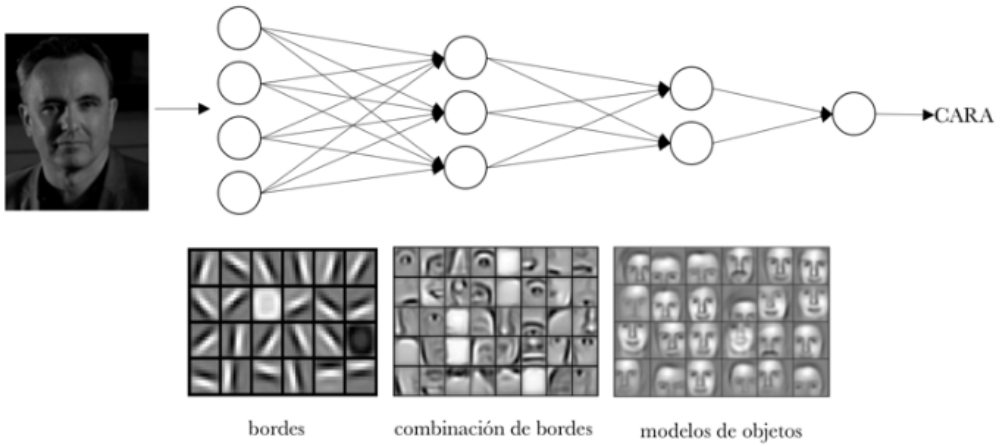

Figura 3.13: Ejemplo de aprendizaje de una red neuronal convolucional. Fuente: Jordi Torres (2018) [31]

La idea este ejemplo es que, de hecho, en una red neuronal convolucional, cada nivel aprende diferentes niveles de abstracción.

Capa de convolución

La principal diferencia entre una capa densamente conectada y una capa convolucional es que una capa densa aprende patrones globales en su espacio de entrada global, mientras que las capas convolucionales aprenden patrones locales en pequeñas ventanas de dos dimensiones.

Visualmente, podemos decir que el objetivo principal de una capa convolucional es detectar las características o rasgos visuales en una imagen, como bordes, líneas, gotas de color, etcétera. Esta es una propiedad muy interesante, porque una vez aprendida una característica en un punto concreto de la imagen la puede reconocer después en cualquier parte de la misma. Por el contrario, en una red neuronal densamente conectada, tiene que volver a aprender el modelo si aparece en una nueva ubicación de la imagen.

Otra ventaja importante es que las capas convolucionales pueden aprender jerarquías espaciales de patrones preservando relaciones espaciales. Es decir, por ejemplo, una primera capa convolucional puede aprender elementos básicos como aristas, y la segunda capa convolucional aprender patrones que incluyen los elementos básicos aprendidos en el nivel anterior. Y así sucesivamente hasta que se aprendan patrones muy complicados, cada vez más abstractos.

En general, las capas convolucionales operan en tensores tridimensionales, llamados mapas de características (*feature maps*), con dos ejes espaciales, alto (*height*) y ancho (*width*), así como un eje de canal (*channels*) también conocido como profundidad. Para las imágenes en color RGB, el tamaño del eje de profundidad es 3, porque la imagen tiene tres canales: rojo, verde y azul. En cambio, en las imágenes en blanco y negro el tamaño del eje de profundidad es 1 (nivel de gris).

Como hemos dicho, la capa de convolución se encarga de extraer los patrones de la matriz de datos de entrada. Para ello utiliza *kernels*, que son matrices de tamaño N×N que almacenan los valores de los pesos w de las neuronas y se utilizan para definir filtros. El kernel opera desplazándose a lo largo de la matriz de datos de entrada, ejecutando una serie de productos y sumas de filas por columna, y los resultados de su salida se almacenan en una nueva matriz llamada mapa de activación o mapa de características.

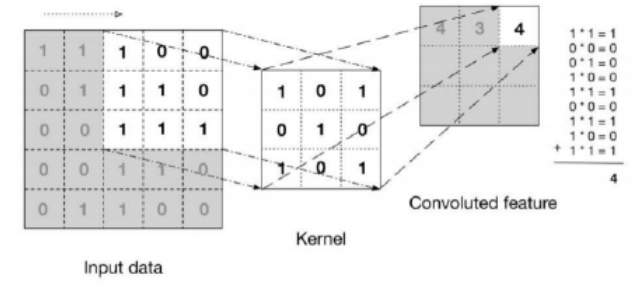

Figura 3.14: Ejemplo de la operación de convolución de un kernel de tamaño 3×3 [6].

Como resultado de este proceso, el tamaño del mapa de características se reduce en comparación con la matriz de datos de entrada. Para evitar esto, se realiza un proceso de "llenado" llamado *padding*, que consiste en agregar ceros alrededor de los bordes de la matriz de entrada. También existe otro parámetro que se conoce por *stride* (paso), que indica el número de celdas por las que se desplaza el kernel para cada paso del proceso realizado sobre la matriz de entrada. El tamaño del mapa de características a la salida se puede calcular de acuerdo con la siguiente fórmula:

$$F = \lfloor \frac{N + 2P - F}{S} + 1 \rfloor$$

Donde:

– F = Tamaño de la matriz de salida.

– N = Tamaño de la matriz de la matriz de entrada.

– P = Parámetro de padding, por defecto vale 0.

– S = Parámetro de stride, por defecto vale 1.

Por ejemplo, en el caso de tener una imagen de 160 × 160 píxeles y un kernel de 5 × 5, esto nos define un espacio de 156 × 156 neuronas en la primera capa oculta ya que solo podemos mover la ventana 155 neuronas hacia la derecha y 155 hacia abajo antes de chocar con algún lado de la imagen de entrada (como se puede ver en la figura 3.15). En este caso suponemos que el parámetro *stride* tiene un avance de 1 píxel.

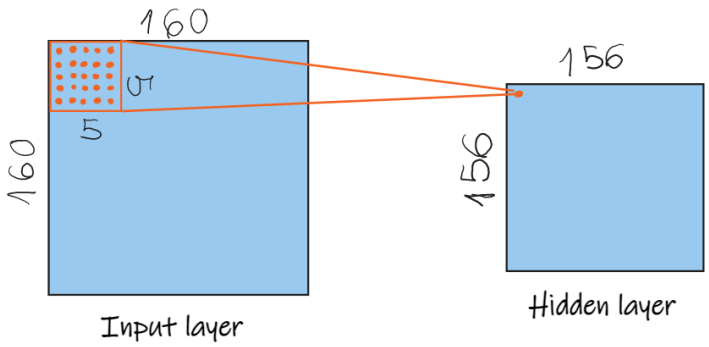

Figura 3.15: Ejemplo visual de convolución con un kernel de tamaño 5 × 5 en una imagen de dimensiones 160 × 160 píxeles. Fuente: Elaboración propia.

Capa de *pooling*

Estas capas se emplean para simplificar aquellos valores que se hallan semánticamente próximos dentro del mapa de características, es decir, condensan la información para evitar posibles problemas en el entrenamiento como el *overfitting*. Las capas de *pooling* suelen ser aplicadas justo después de las capas convolucionales.

En el siguiente ejemplo, elegimos una ventana de 2 × 2 de la capa convolucional y sintetizamos la información en un punto en la capa de pooling. De manera visual podemos verlo así:

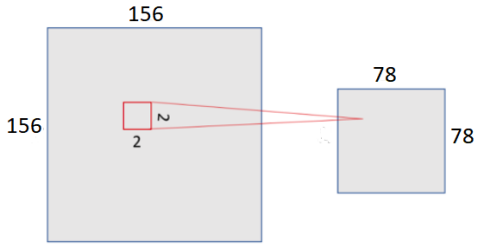

Figura 3.16: Ejemplo visual capa pooling con una ventana de tamaño 2×2. Fuente: Elaboración propia.

La función más empleada es conocida como *Max-pooling*, que define una submatriz de tamaño MxM con *stride* M sobre el mapa de características, donde cada grupo de puntos de entrada se transforma en el valor del máximo. Podemos ver en la figura 3.17 un ejemplo de ello.

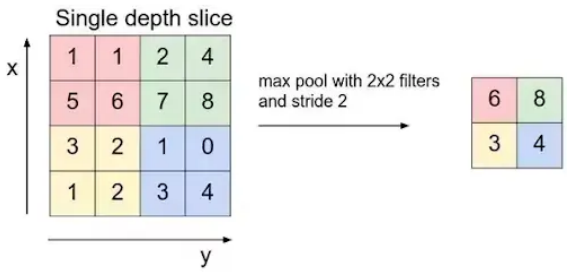

Figura 3.17: Ejemplo de Max pooling [6].

3.3.3. Método de los histogramas de gradientes orientados

Este algoritmo también es conocido como HOG, es un método más moderno que los anteriores y puede ser utilizado para la detección de cualquier tipo de objeto en una imagen a partir de unas plantillas. Aunque sea un método para la detección de objetos en general, presenta un buen comportamiento en la detección de rostros.

El método de los histogramas de gradientes orientados fue presentado por Navneet Dalal y Bill Triggs en 2005 y, a pesar de que lo utilizaron para la detección de peatones en imágenes, es un método que puede ser utilizado para la detección de cualquier objeto, según se modelize.

Este método se basa en la evaluación de histogramas locales normalizados de la orientación del gradiente de una imagen dividida en cuadrículas (que presentan un determinado solape entre ellas), que serán llamadas células. Para cada una de las divisiones de la imagen, se calcula el histograma de las direcciones de gradiente y, combinándolos, se obtiene la representación completa de la imagen. Además, con el objetivo de mejorar la invarianza frente a las condiciones de iluminación y sombras, se realiza una normalización de contraste a partir del cálculo de la energía de los histogramas locales de grandes regiones espaciales denominadas bloques. Los histogramas normalizados de los bloques son los conocidos como descriptores HOG y son los que proporcionan la información sobre cómo cambia la intensidad de la imagen debido a los bordes contenidos en ella.

3.4. Preprocesado

En el análisis de imágenes, los principales factores de variabilidad de patrones en una misma clase pueden ser el ruido, variabilidad inherente, traslaciones, rotaciones, cambio de escala, deformaciones, etc. Luego habrá que tener en cuenta estas posibles variaciones realizando un preproceso que compense la variabilidad existente.

Rotación

Aún estando en la misma posición, los objetos pueden haber girado respecto a su centro, de tal manera que una comparación directa de los mapas de bits pueden dar lugar a una clasificación errónea. Para solucionar esto hay algoritmos de rotación que normalizan la posición de la cara en la imagen. Dada una imagen $i(x, y)$, las nuevas coordenadas de los nuevos puntos que forman la nueva imagen x' e y' serán calculadas como:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.1}$$

Donde el ángulo de rotación viene definido por θ y, por lo tanto, la imagen girada $i_R(x, y)$ será:

$$i_R(x, y) = i(x', y') = i(xcos\theta - ysen\theta, xsen\theta + ycos\theta)$$

En reconocimiento facial, el cálculo del ángulo θ se puede calcular de diferentes formas, siendo muy común su obtención a partir de la posición de los ojos, labios y otros rasgos faciales.

Escalado

Puede haber cambios de escala según la distancia a la que se encuentre el objeto (por ejemplo puede haber zoom o no).

Los algoritmos para afrontar los problemas de escalado permiten reducir o aumentar el tamaño de la imagen, así como hacer zoom en ciertas partes de esta.

Dado un factor de escalado α para las coordenadas x y otro β para las coordenadas y, la nueva imagen $i_E(x, y)$ sería:

$$i_E(x, y) = i(\alpha x, \beta y)$$

Teniendo en cuenta que si x/α y y/β no son enteros, hay que recurrir a una interpolación.

Deformación o inclinación

Las imágenes pueden presentar deformaciones. Por ejemplo, en imágenes bidimensionales, las caras pueden aparecer de perfil, de frente, etc., dando lugar a deformaciones. La deformación de un objeto puede ser muy útil para adaptar un rostro que no esté de frente a la cámara, puesto que puede ser equivalente a un cambio de perspectiva. Una forma sencilla de deformación para el eje x (siendo equivalente para el eje y, cambiando las variables) vendría dada por:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & cot\theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.2}$$

Donde θ representa el ángulo entre el eje x y el nuevo punto tal y como se muestra en la figura 3.18:

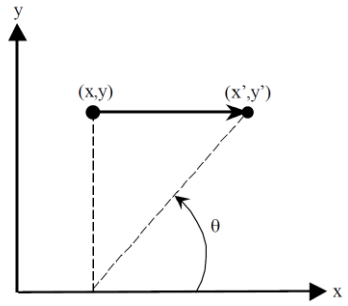

Figura 3.18: Deformación de un objeto en el eje x [16].

Por tanto, la nueva imagen podrá ser definida según la siguiente ecuación, teniendo que ser truncada la coordenada $x + ycot\theta$.

$$i_D(x, y) = i(x', y') = i(x + ycot\theta, y)$$

Recorte

El recorte sirve para quedarnos con la región de la imagen donde se encuentra un rostro. El procedimiento resulta sencillo, ya que una imagen en realidad se trata de una matriz numérica. Por tanto, se puede obtener el recorte tomando una submatriz a partir de los datos de la localización de la cara, que suelen ser el punto (x, y) correspondiente a la esquina superior izquierda de la imagen, y el tamaño de la ventana de detección.

Ecualización del histograma

Aunque no es una transformación geométrica, hay métodos de reconocimiento facial que son muy sensibles a las condiciones de luminosidad. Para ello se realiza una ecualización del histograma, que consiste en repartir los píxeles de forma que quede mejor distribuida la luminancia, por lo que se aumentará el contraste y mejorará la distinción de objetos.

Para ello, dada una imagen i con niveles de intensidad de píxel que van desde 0 hasta $L-1$ (donde normalmente L = 256), su histograma normalizado p_n seguirá la Ecuación 3.3:

$$p_n = \frac{N_n}{N} \quad n = 0, 1, .., L-1 \tag{3.3}$$

Donde N_n es el número de píxeles con intensidad n y N es el número total de píxeles. Luego, se calcula el histograma ecualizado g de la imagen tal que:

3.5. Extracción de características

La extracción de características consiste en la aplicación de algoritmos a imágenes digitales para reducir redundancias e irrelevancias que presenta.

Para reconocer e identificar los rostros en las imágenes es necesario obtener una serie de características que los describan y representen. En esta fase estamos en campos en los que se encuentra el análisis de datos, clasificación de patrones o inteligencia artificial (en especial el Machine Learning). Dado que es una fase muy amplia, existen gran cantidad de técnicas para la extracción de características.

Existen dos tipos de características presentes en las imágenes que pueden ser utilizadas para describir rostros: las geométricas, y las analíticas o basadas en apariencia.

– Características geométricas: son aquellas que miden las distancias entre ciertos rasgos faciales, como los ojos, la nariz o la boca, así como su localización. Estos componentes o puntos de rasgos faciales son extraídos para formar un vector de características. Los métodos que se basan en estas características suelen buscar puntos concretos en las caras para hacer mediciones y obtener dicho vector.

– Características de apariencia: estas describen el cambio en la textura de la cara en función de arrugas, regiones alrededor de la boca, los ojos, y otras propiedades globales de un rostro humano. Normalmente, los métodos holísticos que se basan en estas características codifican la matriz de intensidad de los píxeles sin basarse en rasgos faciales concretos. Además, suelen hacer uso de técnicas que convierten la imagen en un espacio de características de baja dimensión, puesto que las imágenes de caras contienen redundancias o regularidades estadísticas.

Análisis de Componentes Principales

El análisis de componentes principales (PCA), como hemos visto anteriormente en la sección 2.5, consiste en un método de reducción de la dimensionalidad de un problema con variables interrelacionadas.

Esta técnica es utilizada en una gran variedad de aplicaciones y desde que Turk y Pentland lo aplicaron al campo del reconocimiento facial se le conoce como Eigenfaces.

Análisis Discriminante Lineal

El análisis discriminante lineal (LDA), como explicamos anteriormente en la sección 2.5, se trata de un caso general del Discriminante Lineal de Fisher. Este método fue utilizado como base para el método conocido como Fisherfaces. Este método representa una mejora del método de Eigenfaces ya que pretende maximizar la dispersión tanto entre clases diferentes, como dentro de una misma clase, lo que aportará mayor capacidad de distinción entre unos rostros y otros. Además, este método presenta un mejor comportamiento ante diferentes cambios de iluminación en una imagen y es invariante frente a diferentes expresiones faciales.

Local Binary Pattern Histograms

El método de patrones locales binarios (Local Binary Pattern) consiste en que la imagen de una cara puede ser analizada como una composición de pequeños patrones que son invariantes si los transformamos en escala de grises. La imagen se divide en pequeñas regiones, y a partir de ellas son extraídas las características de LBP para ser concatenadas en un único histograma que represente la imagen de la cara de manera eficiente.

Este método parte del operador LBP, cuyo objetivo es analizar la textura en la cercanía de un píxel. Haciendo uso de este operador se obtienen histogramas que contienen información sobre bordes, puntos y zonas planas de la imagen. Finalmente, con esta información podemos comparar y ver la similitud entre histogramas y, entre distintas caras.

3.6. Problemas y dificultades asociadas al reconocimiento facial

El reconocimiento facial es complejo debido a muchos factores. Un grupo de rostros de diferentes personas puede tener grandes similitudes, por lo que distinguir con precisión la identidad de cada persona es un gran desafío. Además, a medida que trabajamos con bases de datos más grandes, la tarea de separar y clasificar se vuelve más compleja. Todo ello al margen de otros factores que nada tienen que ver con el propio algoritmo o la naturaleza de la imagen.

En el trabajo “A Survey of Face Recognition Techniques” [15], se elabora una clasificación de factores relacionados con el problema del reconocimiento facial.

– Factores intrínsecos: estos corresponden totalmente a la naturaleza física y fisiológica del rostro, son independientes del observador. A su vez, estos factores se dividen en dos categorías: intrapersonales e interpersonales. Los intrapersonales corresponden a un cambio de apariencia en la misma persona y pueden ser factores como la edad, las expresiones faciales, la barba, el peinado, etcétera. Por otro lado, los factores interpersonales

corresponden a diferencias faciales que hacen referencia a un grupo de personas, como raza, región de origen o género.

– Factores extrínsecos: se refieren al cambio en la forma de la cara debido a la interacción del observador e incluyen aspectos como la iluminación, la orientación de la cara, la resolución y enfoque de la imagen, el ruido que haya, etcétera.

Básicamente, un sistema de reconocimiento facial necesita nuestros datos y de nuestro rostro para hacer su trabajo. El punto es que hacerlo viola la privacidad de todos, a menos que los participantes expresen formalmente su consentimiento y la ley regional lo permita. Aquí entramos, por tanto, en un terreno jurídico, ético y legal bastante complejo y que no siempre atrae a todo el mundo. Cualquier sistema de reconocimiento debe cumplir los siguientes principios: consentimiento, transparencia, seguridad de los datos, privacidad, integridad y acceso a los datos, responsabilidad proactiva para avisar de posibles brechas de seguridad y capacidad de eliminar completamente todos los datos.

Otro gran problema con el reconocimiento facial es la gran cantidad de almacenamiento de datos, ya que para poder obtener buenos resultados se necesita una gran cantidad de datos para su entrenamiento. Hay que añadir que estos datos son representados mediante imágenes digitales o incluso en vídeos y, dependiendo de la calidad que tengan, pueden ocupar una gran cantidad de memoria, no apta para todos los dispositivos (ya que deben ser potentes y con gran capacidad de almacenamiento). Por ejemplo, en este trabajo en el que el número de imágenes no era excesivamente grande, para entrenar los modelos que hemos realizado hemos tenido que esperar una gran cantidad de horas.

3.7. Seguridad frente a suplantación de identidad

Otro problema del reconocimiento facial es la suplantación de identidad de los piratas informáticos o personas interesadas en robar información confidencial y cualquier otro servicio, poniendo en riesgo a las víctimas de este suceso. A través de un sistema básico de reconocimiento facial 2D, con tan solo una simple foto o máscara, se puede suplantar la identidad de una persona dejando datos, información y servicios privados o secretos al alcance de cualquier delincuente. Para prevenir este tipo de delitos, existen muchos métodos de seguridad, algunos de los cuales vamos a citar.

Una forma de evitar el robo de identidad es mediante un método muy conocido llamado *Eye Blink*, que como su propio nombre indica es capaz de detectar el parpadeo de los ojos en tiempo real en una secuencia de vídeo capturada con una cámara estándar. Este algoritmo tiene múltiples funciones en el reconocimiento facial, como detectar si un conductor está dormido o comprobar el nivel de fatiga de una persona. Aunque fundamentalmente interesa para verificar que la persona que accede a un servicio a través del reconocimiento facial es real y no una foto. En el artículo "Real-Time Eye Blink Detection using Facial Landmarks" [8] se presenta un algoritmo para la detección de parpadeo basado en el valor de EAR (*Eye Aspect Ratio*), que es un indicador que facilita la distancia entre los párpados (basado en 6 puntos) y que caracteriza la apertura del ojo a cada instante.

El uso de cámaras 3D también es otra solución al problema del robo de identidad. El hecho de trabajar en el espacio 3D permite analizar la profundidad de los rasgos faciales, lo que proporciona un método más eficiente y también más seguro porque ya no se podría suplantar la identidad con alguna imagen impresa a papel. Otra ventaja del reconocimiento 3D es que también es capaz de reconocer caras de perfil o inclinadas, a diferencia de los métodos 2D tradicionales que permiten analizar solo la parte frontal de la cara.

La tercera alternativa propuesta es agregar un segundo o tercer sistema de control biométrico. Esto conduce a la verificación de identidad mediante el análisis de más de un atributo físico. Por ejemplo, dicho sistema puede incluir reconocimiento facial parcial y reconocimiento parcial de huellas dactilares, de modo que si las dos partes coinciden en la misma identidad, se permite el acceso al servicio. Este es sin duda uno de los métodos más seguros, ya que es muy complicado falsificar varias propiedades físicas al mismo tiempo.

Capítulo 4

Diseño de un clasificador para reconocimiento de personas.

4.1. Planteamiento del problema

Pretendemos que con este estudio se consiga diseñar un modelo que permita clasificar a las personas mediante técnicas de reconocimiento facial. En este caso utilizaremos redes neuronales convolucionales y para ello haremos uso de una librería llamada `Keras`.

Vamos a construir un modelo de clasificación de imágenes, empleando un algoritmo de aprendizaje automático supervisado entrenado con imágenes que han sido etiquetadas. De hecho, se irán construyendo sucesivos modelos hasta dar con la mejor solución que hemos podido crear. Esto significa que el modelo solo puede identificar a las personas con las que se entrenó pero se podrá clasificar imágenes no pertenecientes al entrenamiento sin necesidad de decirle qué buscar. Esta es la razón por la cual el aprendizaje automático ha revolucionado la visión por computadora: no hay necesidad de codificar reglas (como "si es una mujer rubia, podría ser Charlize Theron, excepto si..."), el sistema aprende de forma cuasi-autónoma.

Como podemos ver en algunos blogs, como por ejemplo en R bloggers [5], hasta hace unos años, la clasificación de imágenes era una tarea difícil que solo podían realizar expertos altamente capacitados con acceso a costosos sistemas informáticos. En los últimos años, frameworks como `Keras` han permitido que estos algoritmos sean implementados básicamente por cualquier persona con un ordenador portátil y acceso a Internet.

En los siguientes apartados mostraremos todo el proceso desde la búsqueda de datos hasta la comparativa y análisis de los distintos modelos que se han ido diseñando y que serán explicados de forma iterativa.

4.2. Búsqueda y preparación de datos inicial

La búsqueda de un conjunto de datos para abordar nuestro estudio, anteriormente explicado, ha sido complicada puesto que aunque hubiera bases de datos de imágenes de personas, estas no estaban clasificadas o bien, existían solamente una o dos fotografías de cada persona. Se pretendía encontrar un conjunto de imágenes que estuviera clasificado y que contuviera un número considerable de imágenes por persona.

Tras la búsqueda de un conjunto de datos que cumpliera estas características se encontró uno en Kaggle [24].

En este conjunto de datos hay 2562 imágenes de 31 personas famosas. Podemos encontrar una carpeta que se llama `Original Images` y a su vez una carpeta por persona, en la que podemos encontrar estas fotografías sin un preprocesado previo. Por otra parte, en la carpeta `Faces` están todas las imágenes juntas (no hay una carpeta por persona). Además, dichas imágenes son el resultado de haber realizado la detección facial y un preprocesado y están listas para diseñar un modelo de clasificación. Aunque cada imagen tiene nombre con la persona que corresponde, en este conjunto de datos encontramos también un archivo csv en el que hay dos variables, una con el nombre de la imagen y otra con la clasificación de esta, facilitándonos bastante la identificación de las personas como veremos más adelante.

Vamos a ver unas capturas de pantalla de la distribución de dichas carpetas para una mejor comprensión:

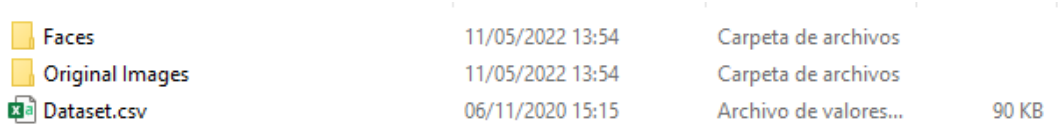

Figura 4.1: Documentos conjunto de datos.

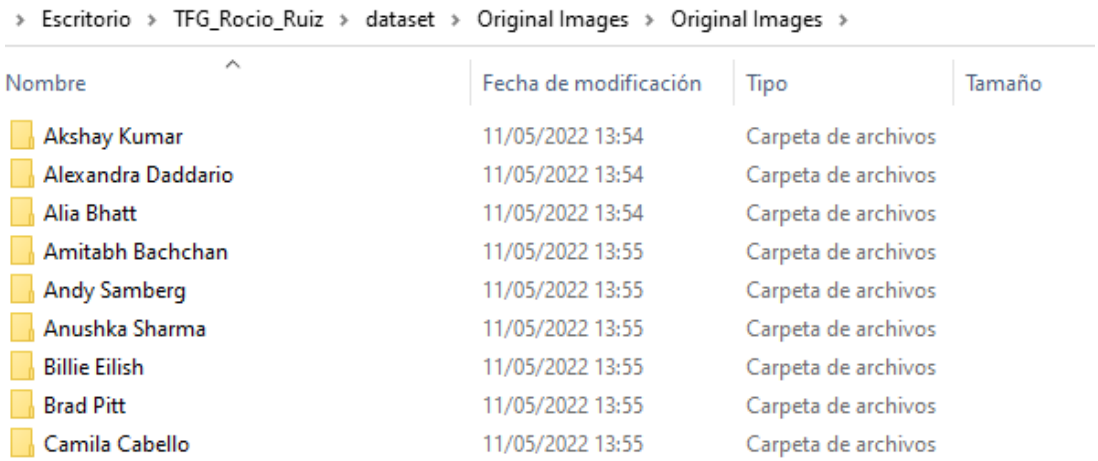

Figura 4.2: Distribución de la carpeta Original Images.

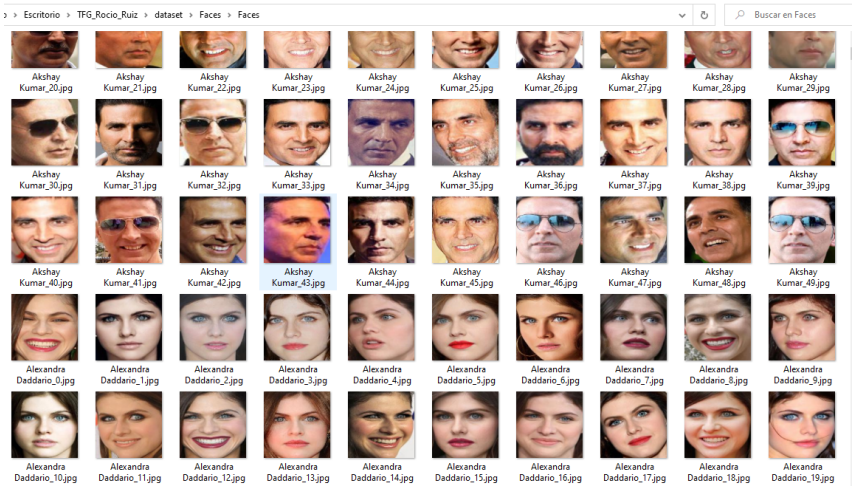

Figura 4.3: Distribución de la carpeta Faces.

4.3. Detección facial y preprocesado

Como se explicó en el capítulo 3 la detección facial es el proceso mediante el cual el sistema localiza la posición de los rostros de personas en una imagen o fotograma. Aunque ya tengamos el resultado de la detección de caras en la carpeta `Faces`, listas para el diseño de un modelo, dentro de nuestros objetivos estaba el ser capaces de realizar el proceso completo, y así se ha hecho con ejemplos que mostraremos a continuación. Una vez aprendido y probado el proceso de detección, y puesto que el tiempo y la disponibilidad de recursos computacionales es finito, aprovechamos el dataset ya preparado con las imágenes detectadas, para poder centrar nuestros esfuerzos en obtener la mejor solución posible para la clasificación de personas.

Dichos ejemplos se han realizado mediante R con la librería `opencv` [21]. En un principio cargamos la librería en cuestión:

```
library(opencv)
```

Empecemos con una fotografía en la que hay varias personas. La imagen de la que hablamos corresponde con la Figura 4.4, es la siguiente:

Figura 4.4: Imagen con grupo de personas.

La librería `opencv` usa para la detección de caras clasificadores en cascada basados en características de Haar que son un método efectivo propuesto por Paul Viola y Michael Jones, como hemos explicado en la sección 3.3.1.

En primer lugar, para leer una imagen con esta librería hay que hacer uso de la función `ocv_read` y, posteriormente, con `ocv_face` podemos detectar las distintas caras que hay en una imagen. Para guardar esta detección como imagen se utiliza el comando `ocv_write`.

```
personas1<-ocv_read('opencv/grupodepersonas.jpg')
faces_personas1<-ocv_face(personas1)
ocv_write(faces_personas1,'opencv/opencv_example2.jpg')
```

Gráficamente se vería de la siguiente forma:

Figura 4.5: Caras detectadas de un grupo de personas. Fuente: Elaboración propia.

Además, con la función `ocv_facemask` obtenemos la localización de cara detectada. Si usamos la función `attr` con el argumento "faces", obtendremos la posición en la que se encuentra cada cara, obteniendo las coordenadas y el radio que se obtienen con `ocv_facemask`.

```
facemask_personas <- ocv_facemask(personas1)
knitr::kable(attr(facemask_personas, 'faces'))
```

Tabla 4.1: Posición detección caras

radius	x	y
20	230	40
20	146	26
26	86	98
23	94	53
22	260	78
24	208	110
22	278	34

A continuación haremos el proceso de detección facial con una imagen de la carpeta `Original Images`. Al igual que antes, primero detectamos dónde está la cara y lo vemos gráficamente.

```
persona2 <- ocv_read('dataset/Original Images/Original Images/
                     Tom Cruise/Tom Cruise_46.jpg')
faces <- ocv_face(persona2)
ocv_write(faces, 'opencv/opencv_example1.jpg')
```

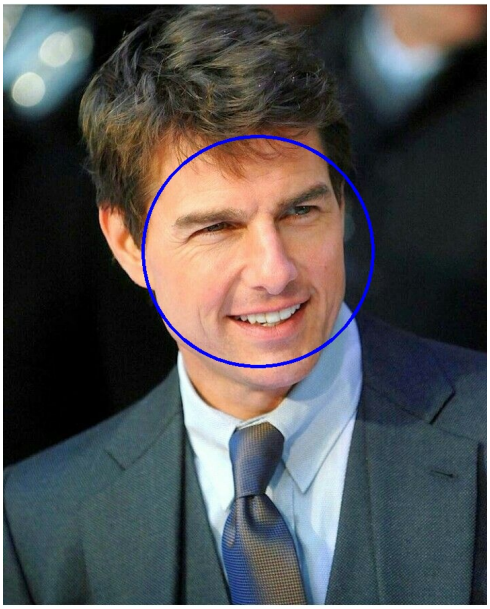

Figura 4.6: Detección facial. Fuente: Elaboración propia.

A continuación construimos un cuadro delimitador, para pintarlo sobre la imagen usamos la librería `magick` [2]:

```
facemask <- ocv_facemask(persona2)
df = attr(facemask, 'faces')
rectX = (df$x - df$radius)
rectY = (df$y - df$radius)
x = (df$x + df$radius)
y = (df$y + df$radius)
library(magick)
# Dibujamos el cuadro delimitador donde
# se encuentra la cara y guardamos la imagen.
imh  = image_draw(image_read('dataset/Original Images/Original Images/
                             Tom Cruise/Tom Cruise_46.jpg'))
rect(rectX, rectY, x, y, border = "red", lty = "dashed", lwd = 2)
image_write(imh,
            path = 'opencv/opencv_example1_box.jpg',
            format = "jpg")
```

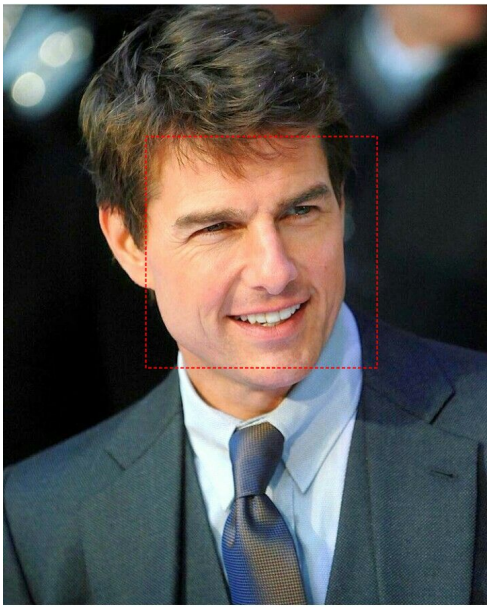

Figura 4.7: Cuadro delimitando la cara. Fuente: Elaboración propia.

Y recortamos dicha imagen por el cuadro, lista para utilizarla en el reconocimiento facial. Para ello usamos el siguiente código:

```
edited = image_crop(imh,
                    paste(x-rectX+1,'x',x-rectX+1,'+',rectX, '+',
                          rectY,sep = ''))
imgedited<-image_convert(edited,"jpg")
image_write(imgedited,
            path = 'opencv/opencv_example1_edited.jpg',
            format = "jpg")
```

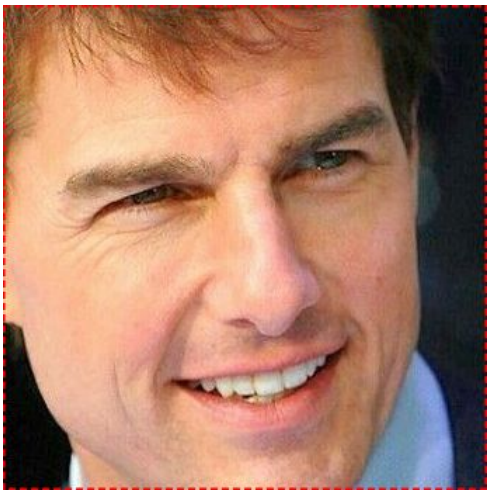

Figura 4.8: Extracción cara. Fuente: Elaboración propia.

Con la librería `magick` podemos hacer múltiples transformaciones en las imágenes como son como rotar, cambiar el tamaño, recortar y girar. Con dicha librería podemos realizar el preprocesado de imágenes previo al diseño del modelo. Lo que hemos realizado con esta

imagen deberíamos hacerlo con todas para quitar el ruido de la imagen original pero, como en la carpeta `Faces` ya tenemos hecho este proceso, no es necesario realizarlo. Se ha hecho con esta imagen a muestra de ejemplo.

Una vez sabemos que tenemos todas estas imágenes procesadas en la carpeta de `Faces`, tenemos que preparar la ordenación de carpetas para poder implementar `Keras`, ya que deben tener una estructura determinada. Tenemos por tanto que tener una carpeta de entrenamiento y otra de test y, en cada una de ellas una carpeta por cada clase, es decir, una carpeta por persona, tal y como se muestra en [12].

Para crear dichas carpetas y separar en entrenamiento y test, usamos el siguiente código en R, ya que es mucho más eficiente que realizarlo a mano. Además, así podemos separar con la librería `tidymodels` en entrenamiento y test de forma aleatoria gracias a la función `initial_split`.

En primer lugar creamos las carpetas, y posteriormente añadiremos las imágenes a sus correspondientes carpetas. Inicialmente vamos a leer el fichero `Dataset.csv`, que contiene dos columnas: una con el nombre de la imagen y otra con la persona que corresponde. Después creamos una carpeta que se llama `data` en la cuál crearemos también las carpetas `train` y `test`.

```
Dataset <- read.csv("dataset/Dataset.csv")
Dataset$label=as.factor(Dataset$label)
levelspeople<-levels(Dataset$label)
#Se usa dir.create() para crear un directorio
dir.create("data")
setwd("data")
```

La función siguiente que se ha definido, `create_dir`, crea una carpeta por cada nivel, es decir, por cada persona. Esto es para no crear nosotros mismos una carpeta por persona tanto en `train` como en `test`.

```
create_dir=function(niveles){
  for (i in niveles) {
    dir.create(i)
  }
}
create_dir(levelspeople)
```

Ya tenemos las carpetas necesarias para implementar `Keras`, pero tenemos que añadir las imágenes. Como comentamos anteriormente, usaremos la librería `tidymodels` que contiene la función `initial_split` que separará los datos en entrenamiento y test, con un 75 % y 25 % de las imágenes respectivamente. Además fijaremos una semilla.

```
library(tidymodels)
set.seed(1234)
split<-initial_split(Dataset)
split_training<-training(split)
split_testing<-testing(split)
```

A continuación creamos un bucle para copiar todas las imágenes de entrenamiento o test en sus respectivas carpetas, a la persona con la que corresponda.

```
todasimagenes<-"dataset/Faces/Faces"
# Con los datos de entrenamiento
dir2<-"data/train"
for (i in 1:nrow(split_training)) {
  file.copy(from = paste0(todasimagenes,"/", split_training$id[i]),
            to = paste0(dir2,"/",split_training$label[i]) )
}
# Con los datos test.
dir1<-"C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/test"
for (i in 1:nrow(split_testing)) {
  file.copy(from = paste0(todasimagenes,"/", split_testing$id[i]),
            to = paste0(dir1,"/",split_testing$label[i]) )
}
```

Ya hemos obtenido la estructura de carpetas con imágenes tal y como se pretendía. Ahora sí podemos realizar la lectura de imágenes con `Keras`. En primer lugar importaremos algunas librerías que utilizaremos.

```
library(tidyverse)
```

Al igual que hemos importado `tidyverse`, repetimos el proceso con las librerías `keras`, `knitr` y `kableExtra`.

Sabemos que nuestras imágenes son a color de dimensión 160 x 160 píxeles.

```
width <- 160
height<- 160
target_size <- c(width, height)
rgb <- 3 #color channels
```

Con el comando `dir`, obtenemos una lista de todas las personas en el orden correcto y la guardamos para usarlo más adelante. En `output_n` guardamos el número de clases, que ya sabemos que es 31, pues nuestro conjunto de datos contiene imágenes de 31 personas distintas. No obstante, el código está naturalmente listo para procesar el número de clases que haya en nuestro caso, coincidente en este caso con el número de subcarpetas.

```
label_list <- dir("C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/train/")
output_n <- length(label_list)
```

Después de establecer la ruta de los datos de entrenamiento (`path_train`), usamos la función `image_data_generator` para definir el preprocesamiento de los datos. Con ella reescalaremos los valores de los píxeles a valores entre 0 y 1 e indicamos que un 25 % de los datos será para validación:

```
path_train <- "C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/train"
train_data_gen <- image_data_generator(rescale = 1/255,
  validation_split = .25)
```

La función `flow_images_from_directory` procesa por lotes las imágenes con la función generadora definida anteriormente. Para cada carpeta de *train* (entrenamiento), asigna una clase, es decir, cada clase es una persona. Creamos dos objetos para los datos de entrenamiento y validación, respectivamente. Notar que no usamos distintas carpetas para

validación y entrenamiento ya que con `Keras` podemos hacerlo automáticamente tal y como hemos definido `train_data_gen`, que reserva un 25 % de los datos de entrenamiento para la validación. Para indicarlo al procesar las imágenes simplemente tenemos que poner dicho generador y en el parámetro `subset` indicar `training` o `validation`. Aunque también podríamos haber tenido carpetas distintas para entrenamiento y validación, teniendo presente que habría que cambiar la ruta y el generador.

```
train_images <- flow_images_from_directory(path_train,
  train_data_gen,
  subset = 'training',
  target_size = target_size,
  class_mode = "categorical",
  shuffle=F,
  classes = label_list,
  seed = 1234)
validation_images <- flow_images_from_directory(path_train,
 train_data_gen,
  subset = 'validation',
  target_size = target_size,
  class_mode = "categorical",
  classes = label_list,
  seed = 1234)
```

Téngase presente que este código asigna a cada persona una clase de 0 a 30, ya que hay 31 personas distintas. Se puede observar a qué persona corresponde cada clase en la tabla 4.2.

Podemos ver el número de imágenes que hay de cada clase tanto en el entrenamiento como en la validación. Con el código `table(train_images$classes)` obtenemos la tabla de frecuencias para la muestra de entrenamiento representada en la tabla 4.2. De igual modo, obtenemos también la tabla de frecuencias de la muestra de validación reflejada en la misma tabla.

Esto corresponde con el número de imágenes de cada clase que, como se ve, no es el mismo en todas; de hecho, varía en algunos casos considerablemente. Es por ello que puede que al ajustar el modelo que diseñaremos clasifique mejor a algunas personas que a otras porque podríamos obtener más información según el tamaño (número de imágenes) que hay en el entrenamiento.

Repetimos el proceso anteriormente descrito pero para los casos del conjunto de prueba, teniendo en cuenta que estas imágenes están en otro directorio.

```
path_test <- "C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/test/"
test_data_gen <- image_data_generator(rescale = 1/255)
test_images <- flow_images_from_directory(path_test,
   test_data_gen,
   target_size = target_size,
   class_mode = "categorical",
   classes = label_list,
   shuffle = F,
   seed = 1234)
```

Tabla 4.2: Clase que corresponde a cada persona y número de imágenes utilizadas para entrenamiento, validación y test.

	Clase	Entrenamiento	Validación	Test
Akshay.Kumar	0	30	10	10
Alexandra.Daddario	1	59	19	14
Alia.Bhatt	2	44	14	21
Amitabh.Bachchan	3	45	15	14
Andy.Samberg	4	51	16	25
Anushka.Sharma	5	38	12	18
Billie.Eilish	6	54	18	26
Brad.Pitt	7	71	23	26
Camila.Cabello	8	48	15	24
Charlize.Theron	9	45	14	19
Claire.Holt	10	57	18	21
Courtney.Cox	11	47	15	18
Dwayne.Johnson	12	36	11	14
Elizabeth.Olsen	13	39	12	20
Ellen.Degeneres	14	48	16	11
Henry.Cavill	15	65	21	20
Hrithik.Roshan	16	51	17	33
Hugh.Jackman	17	60	20	32
Jessica.Alba	18	61	20	27
Kashyap	19	18	6	6
Lisa.Kudrow	20	37	12	21
Margot.Robbie	21	38	12	22
Marmik	22	18	5	9
Natalie.Portman	23	63	20	22
Priyanka.Chopra	24	56	18	28
Robert.Downey.Jr	25	63	21	29
Roger.Federer	26	48	15	14
Tom.Cruise	27	31	10	17
Vijay.Deverakonda	28	55	18	42
Virat.Kohli	29	30	9	10
Zac.Efron	30	48	15	28

En la tabla 4.2 podemos observar también la tabla de frecuencia para los datos test.

4.4. Diseño del modelo

En esta sección vamos a ir desarrollando los modelos construidos para la clasificación de imágenes de forma progresiva. Vamos a partir de un modelo más simple y que se irá complicando paulatinamente para ir mejorando el resultado, hasta un modelo final bastante más complejo pero cuyo resultado justifica el esfuerzo invertido en la mejora de todos los aspectos relacionados con el proceso de modelización. Como hemos dicho anteriormente, para construir estos modelos usaremos `Keras`, que es la librería más recomendada para principiantes. La razón para ello es que su curva de aprendizaje es muy suave en comparación con otras, a la vez que es, sin duda, una de las herramientas para implementar redes neuronales de mayor popularidad en el momento después de TensorFlow.

Esta librería está tanto en Python como en R, nosotros la usaremos en lenguaje R. No tiene sentido compararla con Tensorflow porque, de hecho, Keras proporciona una interfaz de más alto nivel y usa internamente el propio Tensorflow, pero evitando la necesidad de proporcionar determinados detalles en nuestro código. Desde R lo que se hace es, a través del paquete `reticulate`, llamar al keras de Python. Para inicializarnos y aprender cómo funciona esta librería hacemos referencia a [31], [3] y [5].

4.4.1. Línea base. Redes densamente conectadas

La estructura de datos principal en `Keras` es la clase `Sequential`, que permite la creación de una red neuronal básica e introducir capas de manera secuencial, tal y como indica su nombre. Se considera como una secuencia de capas en la que se van "destilando" gradualmente los datos de entrada para obtener la salida deseada.

Definición del modelo

Inicialmente vamos a partir de una red neuronal definida como una secuencia de dos capas (más otra inicial para convertir los datos a una dimensión) que están densamente conectadas, es decir, que todas las neuronas de cada capa están conectadas con todas las neuronas de la capa siguiente, lo que habitualmente denominamos una red simple hacia delante, *feed forward* o perceptrón multicapa (*multilayer perceptron*).

Para poder realizar esta red neuronal tendremos que facilitarle los datos de entrada en la capa inicial, realizando una transformación del tensor (imagen) a una dimensión. Queremos convertir nuestros datos de entrada de dimensión $160 \times 160 \times 3$ en un vector (array) de 76800 números. Esto lo podemos realizar con el comando `layer_reshape` en el que tendremos que indicar la salida que queremos (un vector de 76800 números) y cómo es la forma de datos de entrada (basta con poner en la primera capa `input_shape=c(160,160,3)`).

Una característica de `Keras` es que esta deducirá automáticamente la forma de los tensores entre capas después de la primera, es decir, solo tendremos que especificar la información en la primera capa. Además, para cada capa hay que indicar el número de nodos que posee y la función de activación que se aplicará.

En la primera capa tendremos 31 nodos con la función de activación `sigmoid`. La segunda, es una capa `softmax` de 31 neuronas, lo que significa que devolverá una matriz de 31 valores de probabilidad que representan a las 31 posibles personas (en general, la capa de salida de una red de clasificación tendrá tantas neuronas como clases, menos en una

clasificación binaria, en donde solo necesita una neurona). Cada valor será la probabilidad de que la imagen de la persona actual pertenezca a cada una de ellas.

```
model1 = keras_model_sequential()%>%
  layer_reshape(76800, input_shape = c(160,160,3)) %>% #capa inicial
  layer_dense(units = 31, activation = "sigmoid" ) %>% #primera capa
  layer_dense(units = 31, activation = "softmax") #segunda capa
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  reshape (Reshape)                  (None, 76800)                   0
##  dense_1 (Dense)                    (None, 31)                      2380831
##  dense (Dense)                      (None, 31)                      992
## ================================================================================
## Total params: 2,381,823
## Trainable params: 2,381,823
## Non-trainable params: 0
## ________________________________________________________________________________
```

Para nuestro primer modelo se requieren $2,381.823$ parámetros, $2,380.831$ parámetros para la primera capa y 992 para la segunda.

En la primera capa, por cada neurona i (entre 0 y 30) requerimos 76.800 parámetros para los pesos w_{ij} y por tanto 31×76.800 parámetros para almacenar los pesos de las 31 neuronas. Además de los 31 parámetros adicionales para los 31 sesgos b_j correspondientes a cada una de ellas. En la segunda capa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 31 nodos de la capa anterior, y por tanto se requieren 31×31 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

Configuración del proceso de aprendizaje

A partir del modelo `Sequential`, podemos definir nuevas capas gracias a las tuberías (`%>%`) de la librería `tidyverse` heredando el símbolo del paquete `magrittr`. Una vez definamos un modelo, podemos configurar cómo será su proceso de aprendizaje con la función `compile`, con la que podemos especificar algunas propiedades a través de los argumentos de dicho comando. Se van a especificar los argumentos que se han utilizado.

El argumento `loss` es la función que se usa para evaluar el grado de error entre salidas calculadas y las salidas deseadas de los datos de entrenamiento. Por otro lado, se especifica un optimizador (`optimizer`) para especificar el algoritmo de optimización que permite a la red neuronal calcular los pesos de los parámetros a partir de los datos de entrada y de la función de `loss` definida.

Finalmente debemos indicar la métrica que usaremos para monitorizar el proceso de aprendizaje (y prueba) de nuestra red neuronal. Por ejemplo, en nuestro primer ejemplo especificamos que la función de `loss` es `categorical_crossentropy`, el optimizador es `stocastic gradient descent` (sgd) y la métrica es `accuracy` (porcentaje de imágenes que son correctamente clasificadas).

La función `categorical_crossentropy` calcula la pérdida de intersección entre las etiquetas y las predicciones y, es usada cuando existen dos o más clases de etiquetas.

El optimizador, es el encargado de generar pesos cada vez mejores. Su funcionamiento esencial se basa en calcular el gradiente de la función de coste (derivada parcial) por cada peso de la red. El objetivo es minimizar el error, por lo que se modifica cada peso en la dirección (negativa) del gradiente: $W_{t+1} = W_t - df(coste)/dW * lr$. Para agilizar la convergencia de la función de coste hacia su mínimo, se multiplica el vector de gradiente por un factor llamado *learning rate*. El conjunto de métodos iterativos de reducción de la función de error (búsqueda de un mínimo local), son conocidos cómo los método de optimización basados en el gradiente descendente. Como el cálculo de la derivada parcial de la función de coste respecto a cada uno de los pesos de la red para cada observación es inviable, SGD introduce un comportamiento estocástico (aleatorio), limitando el cálculo de la derivada a una observación (por batch).

```
model1 %>% compile(
  optimizer = "sgd",
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)
```

Entrenamiento del modelo

Una vez definido nuestro modelo y configurado su método de aprendizaje, está listo para ser entrenado. Para ello podemos entrenar o "ajustar" el modelo a los datos de entrenamiento con `fit`, al que podemos pasarle argumentos como `batch_size` (tamaño de los lotes que se usará al ajustar el modelo en una iteración del entrenamiento para actualizar el gradiente), `epochs` (número de veces en las que todos los datos de entrenamiento deben pasar por la red neuronal en el proceso de entrenamiento) y `validation_data` (conjunto de datos que se usan para la validación).

En nuestro caso usaremos un tamaño de `batch_size` de 32, 800 `epochs`, los datos de validación que ya habíamos preparado anteriormente y `verbose` igual a 2 para que muestre cómo va yendo nuestro ajuste en cada `epoch`. En `history1` guardaremos el progreso de este modelo.

```
batch_size=32
history1 <- model1 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=800,
    validation_data = validation_images,
    verbose = 2
  )
```

Obtenemos el gráfico 4.9 que muestra la evolución de la precisión (`accuracy`) y la pérdida (función `loss`) dentro y fuera de la muestra (validación) en cada paso de ajuste de los pesos de nuestra red. En nuestro caso, en este modelo, logramos alrededor del 51 % de precisión en el conjunto de datos de validación después de 800 `epochs`. Sabiendo que existen 31 clases (31 personas distintas), no está mal el modelo dada la complejidad de clasificación y el número de imágenes que tenemos, que podría ser mayor. En las siguientes secciones vamos a intentar mejorar dicho modelo.

Al guardar el ajuste en `history1`, podemos acceder al registro de los valores de `loss` y `accuracy` para los datos de entrenamiento y validación en cada `epoch`, así como otras métricas del proceso. La elección de 800 iteraciones ha sido debido a que como estamos trabajando con una red sencilla, cada `epoch` solo tarda dos segundos. Podríamos seguir entrenando la red pero como queremos estudiar otros modelos no nos pararemos más en este. En redes más complejas no podremos realizar tantas iteraciones ya que como van siendo más complejas, se requerirá de más tiempo para entrenar la red, llegando a tardar alrededor de 3 minutos por `epoch`.

```
history1
```

```
##
## Final epoch (plot to see history):
##         loss: 0.5574
##     accuracy: 0.9072
##     val_loss: 1.791
## val_accuracy: 0.5075
```

Las curvas del proceso de aprendizaje son las siguientes:

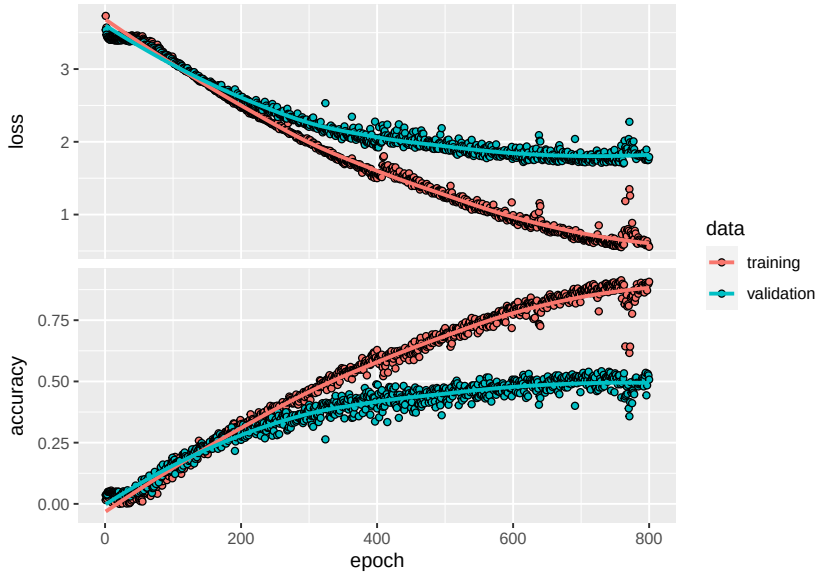

Figura 4.9: Curvas de la función loss y accuracy tras ajustar el modelo inicial de dos capas densamente conectas. Fuente: Elaboración propia.

Evaluación del modelo y predicciones

Una vez hemos entrenado este modelo, podemos evaluar cómo se comporta con los datos de prueba, datos `test`, con el comando `evaluate` y, nos devuelve dos valores, el valor de la función `loss` y el valor del porcentaje de clasificación correcta de los nuevos datos, `accuracy`.

```
model1 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.8050480 0.4586583
```

Tabla 4.3: Porcentaje de acierto en las predicciones con una red de dos capas densamente conectadas mostrando las personas con menor y mayor tasa de acierto

Persona	Porcentaje de acierto
Akshay Kumar	0.00
Margot Robbie	9.09
Anushka Sharma	11.11
Charlize Theron	15.79
Camila Cabello	79.17
Kashyap	83.33
Alexandra Daddario	92.86
Amitabh Bachchan	100.00

Vemos que en este primer modelo que hemos creado con dos capas densamente conectadas (más una inicial para transformar los tensores a una dimensión) obtenemos que con los datos test (imágenes que no han sido utilizadas en el entrenamiento), dicho modelo es capaz de clasificar aproximadamente el 46 % de las imágenes correctamente. Parece que este modelo es bastante malo dado el porcentaje de acierto de clasificación de las imágenes test pero, como hemos dicho anteriormente tenemos 31 clases (bastante más complejo, naturalmente, que un modelo de clasificación binaria) y además no tenemos tantas imágenes por persona como nos gustaría tal y como vimos en la tabla 4.2. Este modelo es mejorable por lo que nuestro propósito en las siguientes secciones será conseguirlo, probando con otras funciones de activación y optimizadores.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test* en la tabla 4.3. En ella, existen personas como Akshay Kumar en las que no se clasifica ninguna imagen bien, mientras que hay otras personas como Amitabh Bachchanen, que es identificado correctamente en todas las imágenes.

Finalmente podemos observar la matriz de confusión de la figura 4.10 para ver mejor los fallos que se producen en la predicción para poder ver si es por alguna causa en concreto, por ejemplo si dos personas se parecen mucho, los fallos estarán concentrados ahí, como es el caso de Margot Robbie (nivel 21) y Claire Holt (nivel 10) que ambas son mujeres rubias con ojos claros y esto provoca confusión. Otro ejemplo a destacar, es la confusión entre Alia Bhatt (nivel 2) y Alexandra Daddario (nivel 1), ambas mujeres de pelo castaño pero distinto color de ojos.

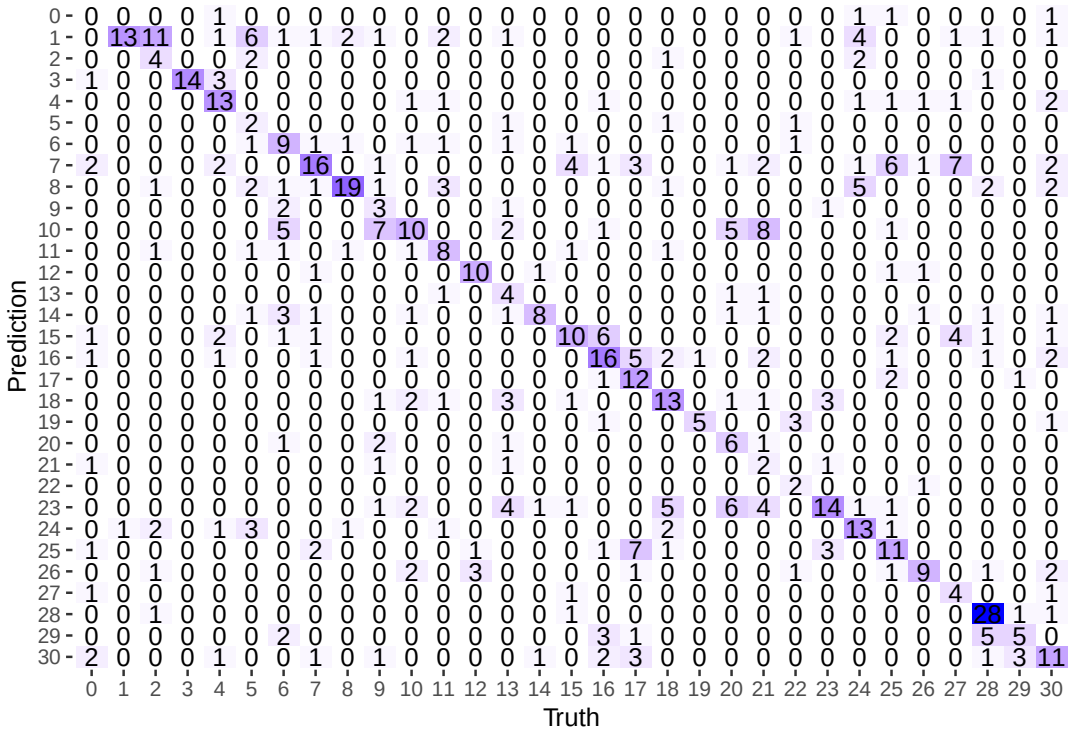

Figura 4.10: Matriz de confusión de las predicciones del modelo inicial de dos capas densamente conectadas. Fuente: Elaboración propia.

4.4.2. Modelo inicial con otra función de activación

Como antes, se usa la clase `Sequential` para añadir las capas secuencialmente. En este caso usaremos otra función de activación, la función `relu` que se mencionará a continuación en qué consiste.

Definición del modelo

Al igual que en el modelo anterior, partiremos de una red neuronal definida como una secuencia de dos capas (más otra inicial para convertir los datos a una dimensión) que están densamente conectadas. La forma de definir las capas es idéntica, a excepción de la primera capa en la que cambiamos la función de activación `sigmoid` por la función de activación `relu`, es decir, pondríamos `layer_dense(units = 31, activation = "relu")`. El número de parámetros es el mismo que en el modelo anterior.

Configuración del proceso de aprendizaje

A partir del modelo diseñado, configuramos el proceso de aprendizaje con la función `compile` de la misma forma que en el modelo anterior: la función de `loss` es `categorical_crossentropy`, el optimizador es `stocastic gradient descent` (sgd) y la métrica es `accuracy`.

Entrenamiento del modelo

Siguiendo con el mismo proceso que antes, ahora procedemos a entrenar el modelo con el comando `fit`. En este caso, solo usaremos 50 `epochs`, a continuación explicaremos el por qué. Asimismo, el valor de `batch_size` es el mismo y las imágenes de validación también las agregamos.

```
history2 <- model2 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=50,
    validation_data = validation_images,
    verbose = 2
  )
```

Obtenemos el gráfico 4.11 que muestra la evolución de la precisión y la pérdida durante el proceso de ajuste de nuestra red. En este caso conseguimos un 5 % de precisión en el conjunto de datos de validación después de 50 `epochs`. Pero además, aunque hubiéramos puesto 200 `epochs` hubiéramos obtenido lo mismo ya que este modelo se estabiliza en ese punto debido a la función de activación `relu`. Obviamente con esta precisión tenemos un ajuste bastante malo.

La función de activación rectified linear unit (`ReLU`) activa un solo nodo si la entrada está por encima de cierto umbral. El comportamiento por defecto y más habitual es que mientras la entrada tenga un valor por debajo de cero, la salida será cero, pero cuando la entrada se eleva por encima, la salida es una relación lineal con la variable de entrada de la forma $f(x) = x$.

En cambio, la función `sigmoid` convierte variables independientes de rango casi infinito en probabilidades simples entre 0 y 1.

Por lo que la función `relu`, al tener solo dos capas en esta red, puede que no se activen los nodos de entrada al no traspasar el umbral en cada iteración y es por ello que, no va a haber mejora en el modelo por muchos `epochs` que añadamos.

Al guardar el ajuste en `history2`, podemos acceder al registro de los valores de `loss` y `accuracy` para los datos de entrenamiento y validación en cada `epoch`, así como otras métricas del proceso.

```
history2
```

```
##
## Final epoch (plot to see history):
##         loss: 3.409
##     accuracy: 0.04883
##     val_loss: 3.403
## val_accuracy: 0.04925
```

Las curvas del proceso de aprendizaje son las siguientes, y vemos como el modelo no va mejorando, se queda estable desde el principio.

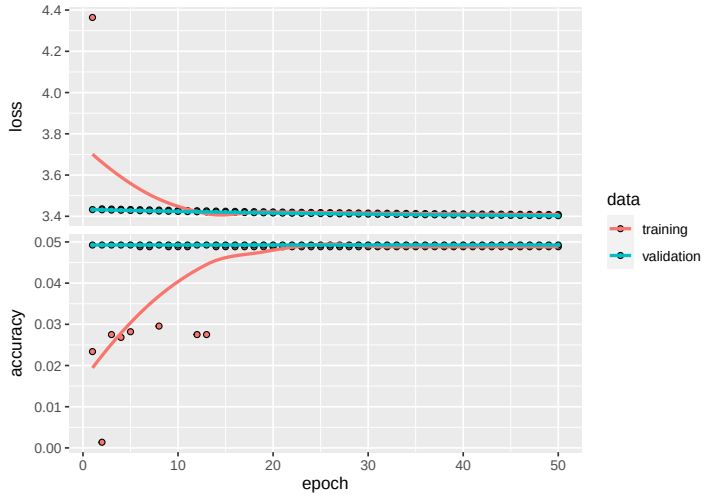

Figura 4.11: Curvas de la función loss y accuracy tras ajustar el modelo inicial de dos capas densamente conectas con la función de activación relu. Fuente: Elaboración propia.

Evaluación del modelo

Una vez hemos entrenado este modelo lo evaluamos. Obviamente nos dará un porcentaje de acierto extremadamente bajo.

```
model2 %>% evaluate(test_images)
```

```
##       loss   accuracy
## 3.40666127 0.04056162
```

Como era de esperar a la luz de los resultados anteriores sobre el conjunto de validación, este modelo con esta función de activación (`relu`) clasifica bien solamente un 4 % de las imágenes de prueba. Aunque en esta red neuronal no haya funcionado dicha función de activación no significa que esta sea peor que otras, depende de la situación. Es más, la función de activación `relu` ha demostrado funcionar en muchas situaciones diferentes, y actualmente es muy usada.

4.4.3. Modelo inicial con otro optimizador

En este modelo lo que cambia respecto al modelo inicial es el optimizador que se utiliza al configurar el proceso de aprendizaje.

Definición del modelo

La definición de este modelo es exactamente igual que en el modelo inicial, con las mismas funciones de activación.

Configuración del proceso de aprendizaje

En este modelo que hemos diseñado utilizamos el optimizador `adam` que es un algoritmo muy popular en el campo del aprendizaje profundo porque logra muy buenos resultados de forma rápida.

```
model3 %>% compile(
  optimizer = optimizer_adam(learning_rate = 0.01),
```

```
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)
```

Entrenamiento del modelo

A continuación entrenamos el modelo con este optimizador, con 200 epochs, mismo `batch_size` (32) y mismas imágenes de validación.

```
history3 <- model3 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=200,
    validation_data = validation_images,
    verbose = 2
  )
```

En el gráfico 4.12 se puede ver la evolución de la precisión y la pérdida durante el proceso de ajuste de esta red. Obtenemos un 5 % de precisión en el conjunto de datos de validación después de 200 epochs. Obtenemos un ajuste muy malo de nuevo.

```
history3
```

```
##
## Final epoch (plot to see history):
##         loss: 3.713
##     accuracy: 0.008941
##     val_loss: 3.409
## val_accuracy: 0.04925
```

Después de realizar 200 `epochs` obtenemos las curvas del proceso de aprendizaje, en las que se ve que no va mejorando el modelo, la función `loss` no decrece y el `accuracy` no crece.

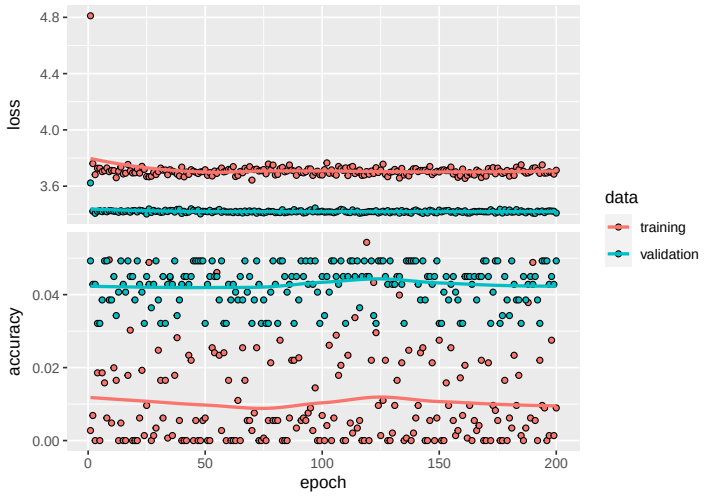

Figura 4.12: Curvas de la función loss y accuracy tras ajustar el modelo inicial de dos capas densamente conectas con el optimizador adam. Fuente: Elaboración propia.

Evaluación del modelo

Al evaluar el modelo es de esperar una tasa de acierto bastante baja.

```
model3 %>% evaluate(test_images)
```

```
##       loss   accuracy
## 3.43319798 0.04056162
```

Se podría pensar que este optimizador no es eficaz, pero utilizando grandes modelos (modelos más complejos que este) y conjuntos de datos, está demostrado que el optimizador Adam puede resolver de manera eficiente problemas prácticos de aprendizaje profundo. Además, si hubiéramos configurando un `learning_rate` más bajo en la configuración del proceso de aprendizaje quizás hubiéramos conseguido un mejor ajuste. Aunque no nos pararemos en este detalle ya que estos modelos son introductorios a nuestro problema, más adelante trabajaremos con redes más complejas.

4.4.4. Modelo con tres capas densamente conectadas

A continuación vamos a diseñar un modelo con tres capas densamente conectadas (más una capa inicial para la conversión de datos a una dimensión). Este modelo será igual que el modelo inicial pero le hemos añadido una capa densa.

Definición del modelo

En este modelo que diseñamos, la primera capa es una capa densa con 1.000 nodos y función de activación `relu`, la segunda es una capa con 31 nodos y función de activación `sigmoid` y la tercera es una capa `softmax` de 31 neuronas. Realmente es el mismo modelo inicial solo que hemos añadido esa capa densa de 1.000 nodos y función de activación `relu`, las otras dos capas son las mismas.

```
model4 = keras_model_sequential()%>%
  layer_reshape(76800, input_shape = c(160,160,3)) %>%
  layer_dense(units = 1000,activation = "relu") %>%
  layer_dense(units = 31, activation = "sigmoid" ) %>%
  layer_dense(units = 31, activation = "softmax")
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  reshape (Reshape)                  (None, 76800)                   0
##  dense_2 (Dense)                    (None, 1000)                    76801000
##  dense_1 (Dense)                    (None, 31)                      31031
##  dense (Dense)                      (None, 31)                      992
## =========================================================================
## Total params: 76,833,023
## Trainable params: 76,833,023
## Non-trainable params: 0
## _________________________________________________________________________
```

Para este modelo se requieren 76, 833.023 parámetros, 76, 801.000 parámetros para la primera capa, 31.031 parámetros para la segunda capa y 992 para la tercera.

En la primera capa, por cada neurona i (entre 0 y 999) requerimos 76.800 parámetros para los pesos w_{ij} y por tanto 1.000 x 76.800 parámetros para almacenar los pesos de las 1000 neuronas. Además de los 1000 parámetros adicionales para los 1.000 sesgos b_j correspondientes a cada una de ellas.

En la segunda capa, por cada neurona i (entre 0 y 30) requerimos 1.000 y por tanto 31 x 1.000 parámetros para almacenar los pesos de las 31 neuronas además de los 31 parámetros adicionales para los sesgos.

En la última capa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 31 nodos de la capa anterior, y por tanto se requieren 31×31 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

Configuración del proceso de aprendizaje

En este modelo la función de `loss` es `categorical_crossentropy`, el optimizador es `stocastic gradient descent` (sgd) y la métrica es `accuracy`.

Entrenamiento del modelo

A continuación entrenamos el modelo con 200 `epochs`, el optimizador `sgd`, mismo batch_size (32) y mismas imágenes de validación.

```
history4 <- model4 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=200,
    validation_data = validation_images,
    verbose = 2
  )
```

Obtenemos nuevamente un gráfico (gráfico 4.13) que muestra la evolución de la precisión y la pérdida dentro y fuera de la muestra en cada paso de ajuste de los pesos de nuestra red. En este caso, logramos una precisión del 43 % en el conjunto de datos de validación después de 200 `epochs`. Con lo cuál vemos que es un ajuste un poco peor comparándolo con el modelo inicial en cuanto a tasa de acierto.

```
history4
```

```
##
## Final epoch (plot to see history):
##         loss: 1.727
##     accuracy: 0.5502
##     val_loss: 2.079
## val_accuracy: 0.4304
```

En el siguiente gráfico podemos ver las curvas asociadas a este modelo.

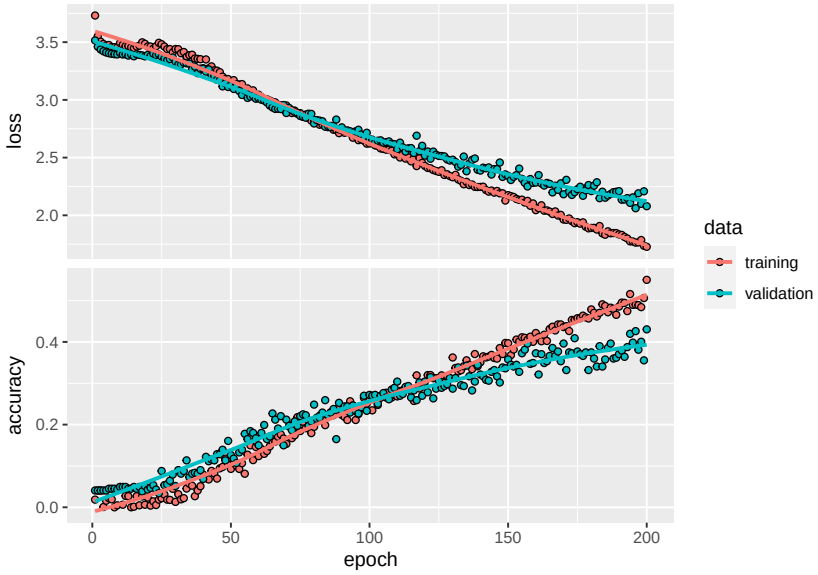

Figura 4.13: Curvas de la función loss y accuracy tras ajustar un modelo con tres capas densamente conectadas. Fuente: Elaboración propia.

Se puede observar que si hubiéramos seguido entrenando el modelo, añadiendo más `epochs`, este podría haber mejorado sin realizarse todavía un sobreajuste de dicho modelo, pero no merece la pena extenderse más aquí al haber modelos mejores más adelante.

Evaluación del modelo y predicciones

Procedemos a evaluar el modelo con las imágenes de prueba, *test*.

```
model4 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 2.0363531 0.4180967
```

Luego obtenemos con este modelo de tres capas densamente conectadas, una tasa de acierto de un 42 % aproximadamente. Es algo menor al primer modelo, pero como hemos dicho antes si hubiéramos seguido entrenando el modelo puede llegar a ser prácticamente igual.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test* en la tabla 4.4. En esta tabla podemos ver que entre las 4 personas con menor tasa de acierto, en tres de ellas no se acierta ninguna imagen mientras que la otra presenta un 4.55 %. En cambio, todas las fotos de Amitabh Bachchan las clasifica correctamente como pasaba con el modelo inicial.

Finalmente podemos observar la matriz de confusión de la figura 4.14. En este caso, por ejemplo podemos resaltar que confunde a Anushka Sharma (nivel 5) con Alexandra Daddario (nivel 1), ambas chicas morenas pero con distinto color de ojos; también confunde bastante a Tom Cruise (nivel 27) con Andy Samberg (nivel 4) ya que son hombres de pelo castaño y ojos marrones que suelen salir sonriendo en las fotos.

Tabla 4.4: Porcentaje de acierto en las predicciones con una red de tres capas densamente conectadas mostrando las personas con menor y mayor tasa de acierto

Persona	Porcentaje de acierto
Akshay Kumar	0.00
Dwayne Johnson	0.00
Elizabeth Olsen	0.00
Margot Robbie	4.55
Ellen Degeneres	81.82
Camila Cabello	87.50
Alexandra Daddario	92.86
Amitabh Bachchan	100.00

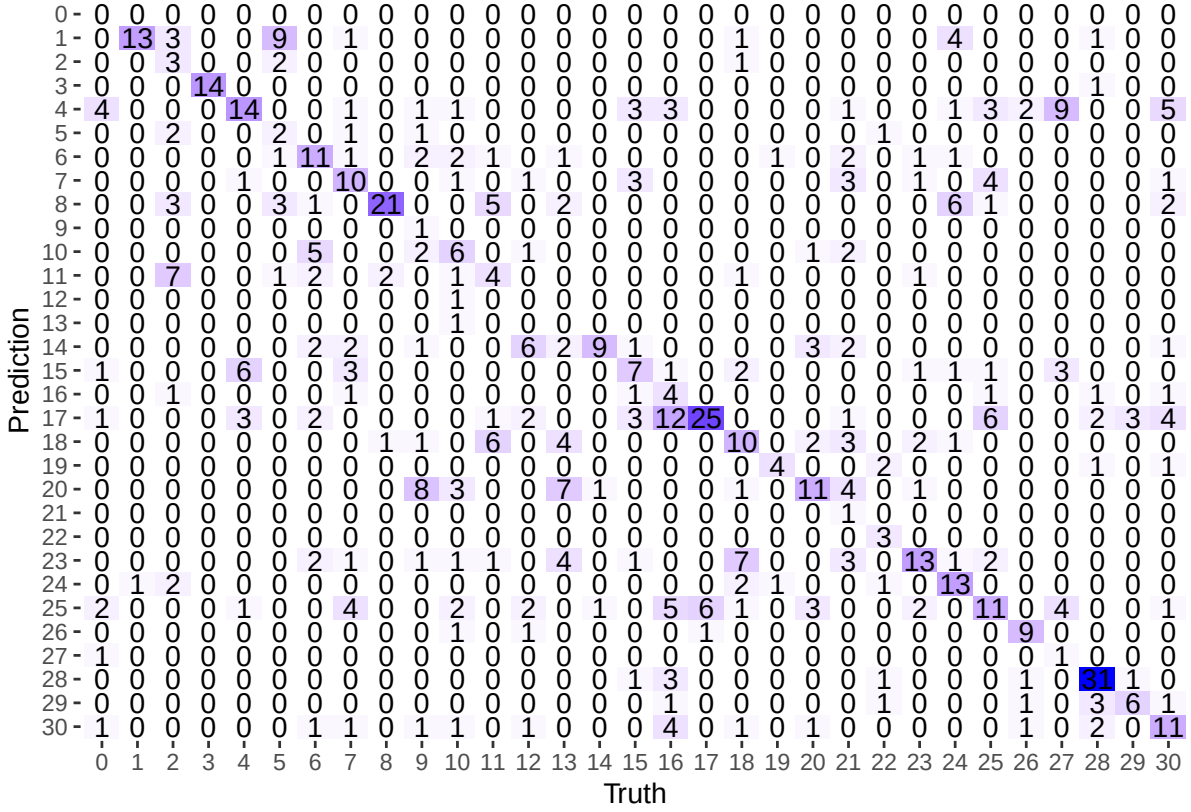

Figura 4.14: Matriz de confusión de las predicciones una red con tres capas densamente conectadas. Fuente: Elaboración propia.

4.4.5. Modelo con tres capas densamente conectadas con otros parámetros

Este modelo va a ser igual que el anterior, con tres capas densamente conectas, pero vamos a realizar pequeños cambios en los parámetros del optimizador.

Definición del modelo

Definimos el mismo modelo que en el apartado anterior.

Configuración del proceso de aprendizaje

A continuación, seguimos usando el mismo optimizador, `stocastic gradient descent` (sgd), pero con otro valor en los hiperparámetros. El argumento `learning_rate` es el mismo que en el modelo anterior ya que por defecto su valor es 0.01. Al parámetro `momentum` (float >= 0), parámetro que acelera SGD en la dirección correspondiente y amortigua las oscilaciones, le damos el valor 0.1. Al parámetro `decay` (float>=0), con el que la tasa de aprendizaje disminuye con cada actualización, le damos el valor `0.01`, para permitir que el aprendizaje avance más rápido al principio con *learning rates* más grandes aunque hay que tener cuidado ya que en ocasiones nos hace converger más rápido, pero en otras ocasiones nos hace dar saltos sin sentido e incluso diverger.

```
model5 %>% compile(
  optimizer = optimizer_sgd(learning_rate = 0.01,
                            decay = 0.01, momentum = 0.1),
  loss = "categorical_crossentropy",
  metrics = "accuracy"
)
```

Entrenamiento del modelo

Ahora procedemos a entrenar el modelo con estos parámetros. Realizará el ajuste con 200 `epochs`, tamaño 32 de `batch_size` y las imágenes de validación que tenemos.

```
history5 <- model5 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=200,
    validation_data = validation_images,
    verbose = 2
  )
```

```
##
## Final epoch (plot to see history):
##         loss: 3.059
##     accuracy: 0.2256
##     val_loss: 3.092
## val_accuracy: 0.2184
```

El gráfico 4.15 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión del 21 % en el conjunto de datos de validación después de 200 `epochs`.

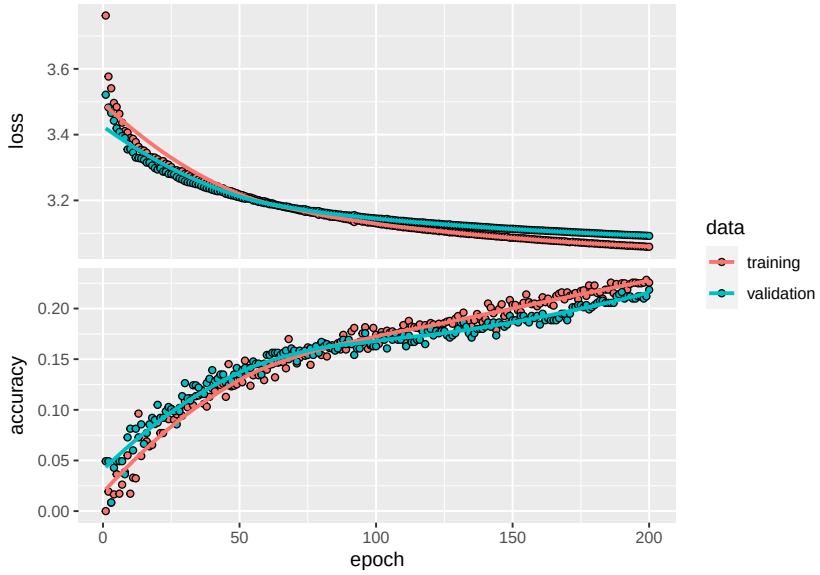

Figura 4.15: Curvas de la función loss y accuracy de un modelo formado por tres capas densamente conectadas con otros parámetros y 200 epochs. Fuente: Elaboración propia.

Podríamos añadir más `epochs` y conseguir una mejor precisión. En el gráfico 4.16 podemos ver el ajuste de este modelo si lo hubiéramos con 600 `epochs`.

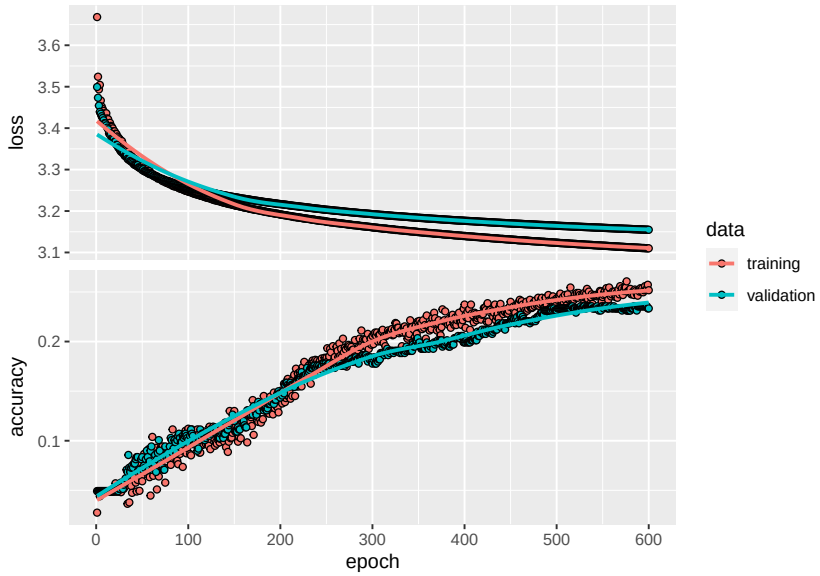

Figura 4.16: Curvas de la función loss y accuracy de un modelo formado por tres capas densamente conectadas con otros parámetros y 600 epochs. Fuente: Elaboración propia.

En este último gráfico vemos que tras 600 `epochs`, sigue sin haber un sobreajuste del modelo, es decir, podríamos seguir entrenándolo porque parece que el `accuracy` tanto en los datos de entrenamiento como de validación van a seguir creciendo pero, lo hace de manera muy lenta. Probaremos con más modelos en las siguientes secciones.

Evaluación del modelo

Procedemos a evaluar el modelo con 200 `epochs`.

```
model5 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 3.0895946 0.1965679
```

Ahora evaluamos el modelo con 600 `epochs`. A lo que llamamos `model5_1` es `model5`, pero entrenándolo con 600 `epochs`.

```
model5_1 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 3.1494327 0.2043682
```

Vemos que tanto con 200 o 600 `epochs`, en este modelo de tres capas densas con los parámetros de defecto cambiados, obtenemos que con los datos test, dicho modelo es capaz de clasificar aproximadamente el 20 % de las imágenes correctamente. Hemos visto que estos parámetros que hemos tomado no son los mejores, ya que han empeorado respecto al modelo de tres capas densas con los parámetros por defecto.

4.4.6. Primera red neuronal convolucional

Hasta ahora solo hemos usado redes neuronales hacia delante (*feed forward*), a partir de ahora trabajaremos con redes neuronales convolucionales, es decir, introduciremos capas que definen a este tipo de redes, como son las capas de convolución y *pooling*. En general las capas convoluciones operan sobre tensores de 3D, llamados *feature maps*, con dos ejes espaciales de altura y anchura (*height* y *width*), además de un eje de canal (*channels*) también llamado profundidad (*depth*). En nuestro caso tenemos imagenes a color (RGB), luego la dimensión del eje *depth* es 3, pues la imagen tiene tres canales: rojo, verde y azul.

Definición del modelo

Pasemos a implementar nuestra primera red neuronal convolucional, que consistirá en una convolución seguida de un max-pooling, seguida de otra convolución y otro max-pooling. En nuestro caso, en la primera convolución tendremos 32 filtros usando una ventana de 5×5, en el primer max-pooling una ventana de 2×2, en la segunda convolución tendremos 64 filtros usando otra vez una ventana de 5×5 y el segundo max-pooling con una ventana de 2×2. Usaremos la función de activación `ReLU`. En este caso, estamos configurando una red neuronal convolucional para procesar un tensor de entrada de tamaño (160, 160, 3), que es el tamaño de nuestras imágenes a color y lo especificamos mediante el valor del argumento `input_shape=(160, 160, 3)` en nuestra primera capa. Después de esto necesitamos pasarle el comando `layer_flatten` para unidimensionalizar la entrada multidimensional, lo cual se usa para la transición de la capa convolucional a la capa completamente conectada (no afecta el tamaño del lote). Finalmente tendremos nuestra capa densa con 31 neuronas y función de activación `softmax` que devolverá una matriz de 31 valores de probabilidad que representan a las 31 posibles personas.

```
model6<-keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(5,5),
                activation = "relu",input_shape = c(160,160,3)) %>%
```

```
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(5,5),
                activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(units = 31,activation = "softmax")
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  conv2d_1 (Conv2D)                  (None, 156, 156, 32)            2432
##  max_pooling2d_1 (MaxPooling2D)     (None, 78, 78, 32)              0
##  conv2d (Conv2D)                    (None, 74, 74, 64)              51264
##  max_pooling2d (MaxPooling2D)       (None, 37, 37, 64)              0
##  flatten (Flatten)                  (None, 87616)                   0
##  dense (Dense)                      (None, 31)                      2716127
## ================================================================================
## Total params: 2,769,823
## Trainable params: 2,769,823
## Non-trainable params: 0
## ________________________________________________________________________________
```

El número de parámetros de la primera capa *conv2D* corresponde a la matriz de pesos W de 5×5 y un sesgo b para cada uno de los filtros. Esto supone un total de 2432 parámetros en esta capa, teniendo en cuenta que nuestras imágenes tienen 3 canales al ser a color (((5×5×3)+1)×32). El max-pooling no requiere parámetros puesto que es una operación matemática de encontrar el máximo tal y como vimos en la sección 3.3.2.

En este caso el tamaño de la segunda capa de convolución resultante es de 74×74 dado que ahora partimos de un espacio de entrada de 78 × 78 × 32 y una ventana deslizante de 5×5, teniendo en cuenta que tiene un *stride* de 1. El número de parámetros 51.264 corresponde a que la segunda capa tendrá 64 filtros con 801 parámetros cada uno (1 corresponde al sesgo), y luego tenemos la matriz W de 5×5 para cada una de las 32 entradas (((5×5×32)+1)×64=51264).

La salida de las capas *Conv2D* y *MaxPooling2D* es un tensor 3D de forma (*height*, *width*, *channels*). Las dimensiones width y height tienden a reducirse a medida que nos adentramos en las capas ocultas de la red neuronal. El número de kernels/filtros es controlado a través del primer argumento pasado a la capa *Conv2D* (habitualmente de tamaño 32 o 64).

El siguiente paso, ahora que tenemos 64 filtros de 37×37, consiste en añadir una capa *flatten* para aplanar el tensor a un vector de una dimesión de tamaño 87.616 (37×37×64=87616).

Finalmente el número de parámetros de la capa `softmax` es 87.616 × 31 + 31 = 2, 716.127, con una salida de un vector de tamaño 31.

A modo ilustrativo, las capas de *convolución* y *pooling* serían de la siguiente forma:

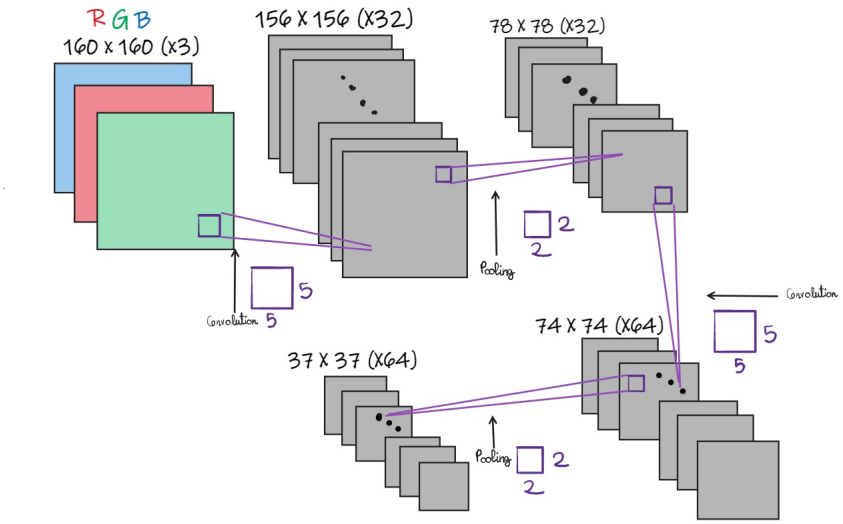

Figura 4.17: Capas de convolución y pooling. Fuente: Elaboración propia.

Configuración del proceso de aprendizaje

Como de costumbre, la función de `loss` es `categorical_crossentropy`, el optimizador es `stocastic gradient descent` (sgd) y la métrica es `accuracy`.

Entrenamiento del modelo

Para entrenar un modelo de redes neuronales convolucionales hay que tener en cuenta que en las capas convolucionales es donde se requiere más memoria y computación para almacenar los datos. En cambio, en la capa densamente conectada `softmax`, se necesita menos memoria pero utiliza muchísimos parámetros que deben ser aprendidos. Hay que ser conscientes del tamaño de este tipo de redes, ya que tienen un gran costo computacional y no podemos entrenar con tantos `epochs` como hemos hecho anteriormente ya que tardarán en entrenarse muchísimas horas en un equipo convencional como el empleado en este trabajo. En este caso entrenaremos con 80 `epochs`.

```
history6 <- model6 %>%
  fit(train_images,
    batch_size=batch_size,
    epochs=80,
    validation_data = validation_images,
    verbose = 2
  )
```

```
##
## Final epoch (plot to see history):
##         loss: 0.002805
##     accuracy: 1
##     val_loss: 2.774
## val_accuracy: 0.5739
```

El gráfico 4.18 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión del 57 % en el conjunto de datos de validación después de 80 `epochs`.

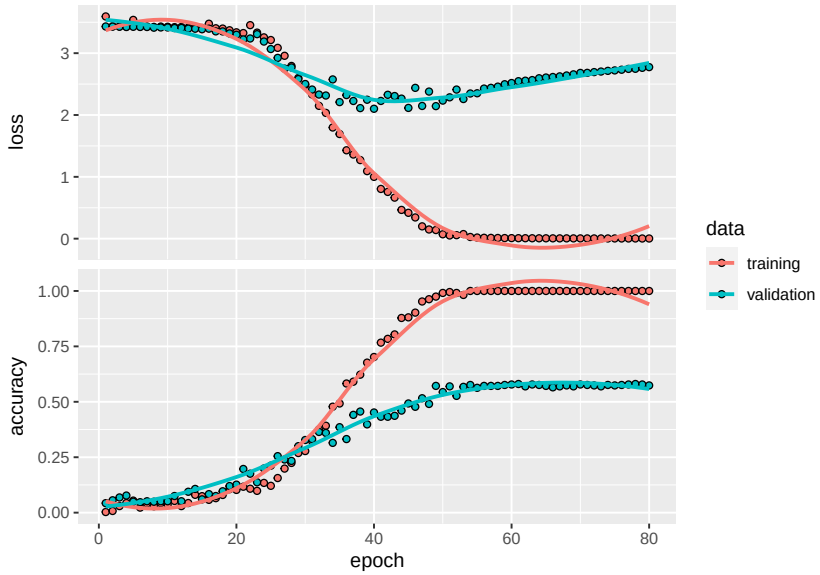

Figura 4.18: Curvas de la función loss y accuracy de una red neuronal convolucional con 2 capas de convolución y 2 capas de pooling. Fuente: Elaboración propia.

Si miramos el comportamiento general de ambas gráficas vemos que son un ejemplo característico de cuando se produce un sobreajuste (*overfitting*). Por un lado la función `loss` de los datos de validación alcanzan su mínimo después de aproximadamente unos 40 `epochs` y a partir de ahí empieza a crecer. En cambio la función `loss` de los datos de entrenamiento disminuye linealmente hasta llegar prácticamente a 0, donde se mantiene. Por otra parte, la `accuracy` de los datos de validación se estabiliza a partir de los 50 `epochs` con un 57 %, mientras la `accuracy` de los datos de entrenamiento crece linealmente con las `epochs` hasta alcanzar el 100 %.

A pesar de todo, los resultados no son tan malos, si consideramos que tenemos muy pocas imágenes y 31 clases.

Evaluación del modelo y predicciones

Procedemos a evaluar este modelo de redes neuronales convolucionales con las imágenes test.

```
model6 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 2.5184309 0.5522621
```

En los datos test conseguimos una tasa de acierto del 55 %, que no está mal para los modelos que hemos visto hasta ahora.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test* en la tabla 4.5. Aquí podemos observar que el

Tabla 4.5: Porcentaje de acierto en las predicciones con la red neuronal convolucional entrenada.

Persona	Porcentaje de acierto
Akshay Kumar	30.00
Billie Eilish	34.62
Margot Robbie	36.36
Tom Cruise	41.18
Virat Kohli	80.00
Ellen Degeneres	81.82
Kashyap	83.33
Amitabh Bachchan	85.71

porcentaje mínimo de acierto en una persona es de un 30 % mientras que el más alto es un 85.71 %.

Además, podemos observar la matriz de confusión de la figura 4.19 para ver mejor los fallos que se producen en las predicciones. En esta matriz de confusión los fallos están más dispersos y no es entre dos personas, a excepción de que suele confundir a Camila Cabello (nivel 8) con Jessica Alba (nivel 18) que son ambas mujeres con orígenes cubanos y mexicanos, de cabello castaño y ojos marrones.

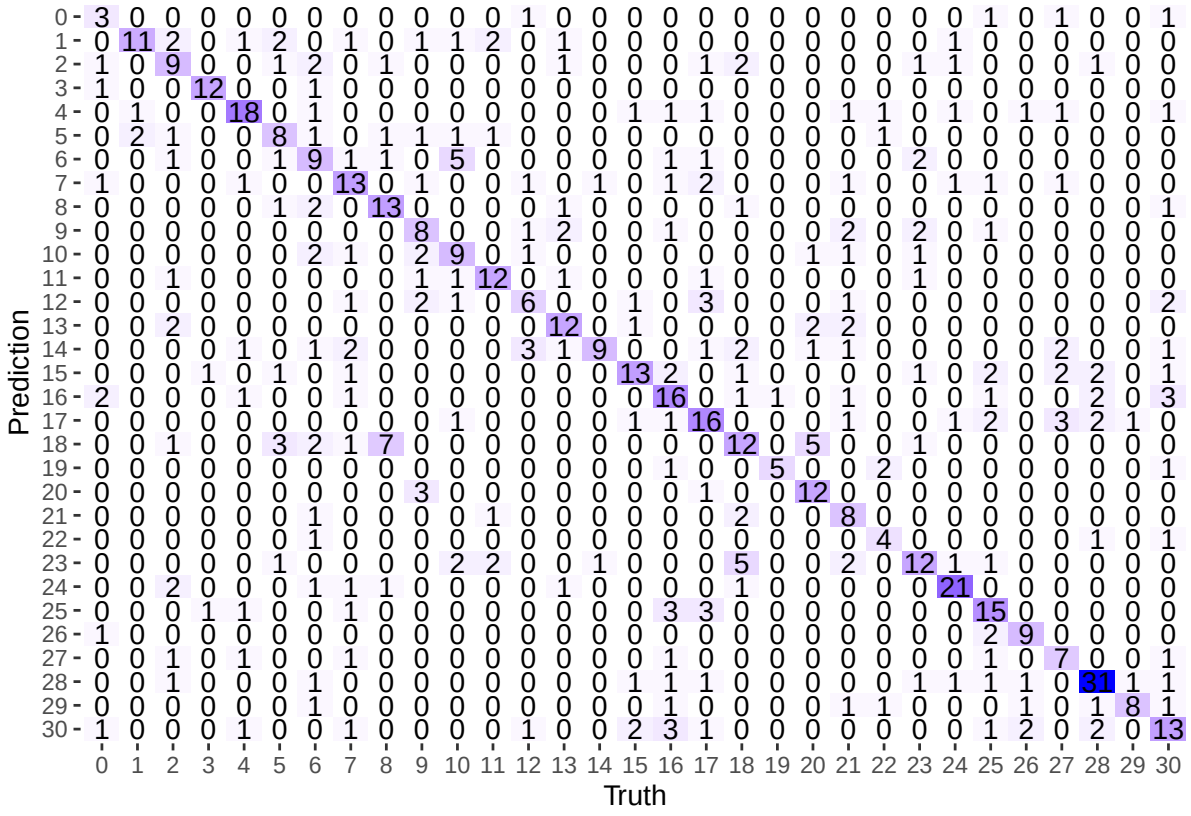

Figura 4.19: Matriz de confusión de las predicciones mediante la red neuronal convolucional diseñada. Fuente: Elaboración propia.

4.4.7. Red neuronal convolucional con un modelo preentrenado

Como se puede observar, Keras permite construir redes neuronales de manera muy flexible, hay millones de formas de configurar modelos y no siempre sabemos si este tendrá una buena o mala precisión. Es por ello que es bastante complejo saber con certeza qué red ajustará mejor o peor nuestros datos. Esto es un problema ya que, a medida que vamos añadiendo capas, sobre todo capas convolucionales o de *pooling*, necesitará más memoria y utilizará más parámetros y al final repercute en el tiempo de computación.

Afortunadamente, existe una manera de generar rápidamente buenos resultados de referencia: cargar modelos previamente entrenados que han demostrado su eficacia en competencias a gran escala, como `ImageNet`. Para realizar esta primera red convolucional preentrenada nos hemos fijado en [5]. Aquí, cargamos la red `xception` con los pesos previamente entrenados en el conjunto de datos de `ImageNet`, excepto la capa final (que clasifica las imágenes en el conjunto de datos de ImageNet) que entrenaremos en nuestro propio conjunto de datos. Lo logramos configurando "include_top" con el valor FALSE.

Definición del modelo

En primer lugar, para definir este modelo con la arquitectura `Xception` y los pesos de `ImageNet`, necesitamos usar la función `application_xception`, en la que indicamos la forma de los datos de entrada, que como ya sabemos es de 160 X 160 X 3. Posteriormente, con la función `freeze_weights` congelamos los pesos extraídos de este modelo preentrenado.

```
mod_base <- application_xception(weights = 'imagenet',
   include_top = FALSE, input_shape = c(width, height, 3))
freeze_weights(mod_base)
```

Ahora definimos una función que construye una capa sobre la red previamente entrenada y establece algunos parámetros en las variables que luego podemos usar para ajustar el modelo como queramos.

En este caso, después de la red entrenada añadimos la capa `layer_global_average_pooling_2d`, que es una operación de agrupación diseñada para reemplazar capas completamente conectadas en las CNN (*Convolutional Neural Network*) clásicas. Con ellos conseguimos que los datos estén en una dimensión. Posteriormente añadimos una capa densa con un número de neuronas que depende del valor `n_dense`. A continuación una capa de activación `relu` seguida de una capa `dropout` que busca regularizar la red, previniendo el posible sobreajuste al otorgar a cada neurona una probabilidad del 20 % (valor por defecto en este caso) de no activarse durante la fase de entrenamiento. Finalmente una capa `softmax` con 31 neuronas para poder obtener la matriz de 31 valores de probabilidad que representan a las 31 posibles personas.

Además en esta función va incluido cómo va a ser el proceso de aprendizaje.

```
model_function <- function(learning_rate = 0.001,
  dropoutrate=0.2, n_dense=1024){

    k_clear_session()

  model <- keras_model_sequential() %>%
```

```
    mod_base %>%
    layer_global_average_pooling_2d() %>%
    layer_dense(units = n_dense) %>%
    layer_activation("relu") %>%
    layer_dropout(dropoutrate) %>%
    layer_dense(units=output_n, activation="softmax")

  model %>% compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = learning_rate),
    metrics = "accuracy"
  )

  return(model)

}
```

Luego, por tanto, podemos definir este modelo usando la función anteriormente definida.

```
model7 <- model_function()
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                   Output Shape                Param #    Trainable
## =========================================================================
##  xception (Functional)          (None, 5, 5, 2048)          20861480   N
##  global_average_pooling2d (Glo  (None, 2048)                0          Y
##  balAveragePooling2D)
##  dense_1 (Dense)                (None, 1024)                2098176    Y
##  activation (Activation)        (None, 1024)                0          Y
##  dropout (Dropout)              (None, 1024)                0          Y
##  dense (Dense)                  (None, 31)                  31775      Y
## =========================================================================
## Total params: 22,991,431
## Trainable params: 2,129,951
## Non-trainable params: 20,861,480
## _________________________________________________________________________
```

Podemos observar como en este modelo se necesitan $22,991.431$ parámetros, que es una barbaridad. Pero, como hemos utilizado una red ya entrenada, solo necesitamos entrenar $2,129.951$ parámetros, pues el resto ya lo estarían. Es decir, la red ya tiene entrenados $20,861.480$ parámetros.

En la primera capa densa, por cada neurona i (entre 0 y 1023) requerimos 2.048 parámetros para los pesos w_{ij} y por tanto 1.024×2.048 parámetros para almacenar los pesos de las 1.024 neuronas. Además de los 1.024 parámetros adicionales para los 1.024 sesgos b_j correspondientes a cada una de ellas. En la última capa densa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 1.024 nodos de la capa anterior, y por tanto se requieren 31×1.024 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

El resto de capas que no necesitan parámetros son porque estas capas no modifican la forma de la red, simplemente estas capas utilizan operaciones matemáticas sin necesidad de nuevos parámetros para hallar los pesos y sesgos.

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function`, ya indicamos cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

En este caso, para entrenar esta red neuronal, el tamaño de `batch_size` sigue siendo 32, y el número de `epochs` es 50. Añadimos además dos nuevos argumentos, `steps_per_epoch`, número total de pasos antes de declarar finalizada un `epoch` e iniciar el siguiente y `validation_steps`, número total de pasos para la validación antes de parar.

```
batch_size <- 32
epochs <- 50
hist7 <- model7 %>% fit(
  train_images,
  steps_per_epoch = train_images$n / batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.1162
##     accuracy: 0.9873
##     val_loss: 1.526
## val_accuracy: 0.6071
```

El gráfico 4.20 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 61 % en el conjunto de datos de validación después de 50 `epochs`. Hasta ahora este es el modelo que más tasa de acierto ha tenido en los datos de validación.

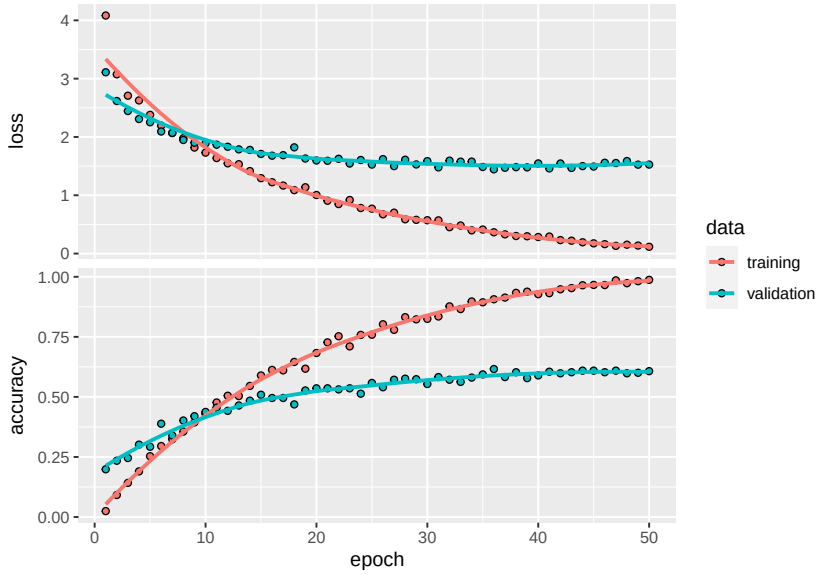

Figura 4.20: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada xception. Fuente: Elaboración propia.

Pese a la medida de *dropout* el modelo sigue sobreajustando, como observamos en el hecho de que los datos sobre la validación empeoran significativamente respecto a los del conjunto de entrenamiento.

Evaluación del modelo y predicciones

A continuación evaluamos el modelo con los datos *test*.

```
model7 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.4680020 0.6162246
```

Hemos conseguido el modelo más eficaz hasta ahora, obteniendo alrededor de un 62 % de acierto en la clasificación de las imágenes de prueba. Los modelos preentrenados son muy útiles y además suelen tener buenos resultados, ya que suelen ser proporcionados por estudios de grandes compañías que han podido extraer características de conjuntos enormes de imágenes como entrada a redes entrenadas durante un periodo considerable de tiempo y con grandes recursos computacionales.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test*, reflejado en la tabla 4.6. A simple vista parece que obtenemos mejores resultados que en el modelo anterior ya que el porcentaje mínimo de acierto en una persona es de un 30 %, al igual que antes pero aquí el más alto es un 92.86 %

Adicionalmente, podemos observar la matriz de confusión de la figura 4.21 para ver mejor los fallos que se producen en las predicciones. Se vuelve a dar el caso de confundir a Margot Robbie (nivel 21) y Claire Holt (nivel 10), ambas mujeres rubias con ojos claros.

Tabla 4.6: Porcentaje de acierto en las predicciones con la red neuronal convolucional con un modelo preentrenado.

Persona	Porcentaje de acierto
Akshay Kumar	30.00
Marmik	33.33
Anushka Sharma	33.33
Elizabeth Olsen	35.00
Vijay Deverakonda	83.33
Dwayne Johnson	85.71
Roger Federer	92.86
Amitabh Bachchan	92.86

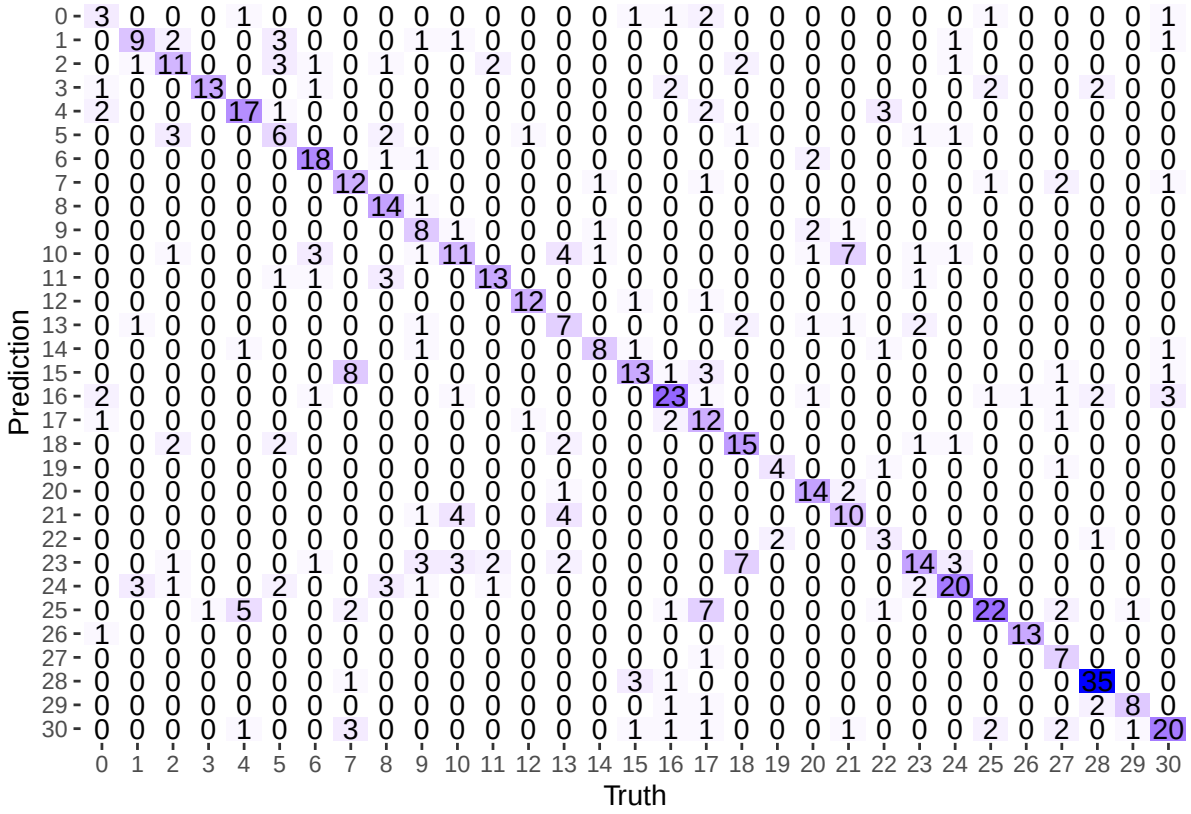

Figura 4.21: Matriz de confusión de las predicciones mediante la red neuronal convolucional con un modelo preentrenado. Fuente: Elaboración propia.

4.4.8. Red con arquitectura Xception y distinto batch size

Nuevamente volvemos a utilizar una red entrenada con la misma arquitectura que en la sección anterior. Aquí la diferencia va a ser el tamaño de `batch_size` para ver cómo afecta este parámetro. Además tendremos más `epochs`.

Definición del modelo

Ya hemos explicado el diseño de este modelo en la sección anterior ya que es exactamente el mismo.

```
model8 <- model_function()
```

Configuración del proceso de aprendizaje

Al igual que antes, sabemos que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Procedemos a entrenar el modelo con tamaño de `batch_size` 50 y con 70 `epochs`.

```
batch_size <- 50
epochs <- 70
hist8 <- model8 %>% fit(
  train_images,
  steps_per_epoch = train_images$n / batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.1308
##     accuracy: 0.9795
##     val_loss: 1.502
## val_accuracy: 0.625
```

El gráfico 4.22 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 63 % en el conjunto de datos de validación después de 70 `epochs` y tamaño de `batch_size` 50. Esto supone una pequeña mejora en la precisión del modelo de validación con respecto al modelo anterior, pero no hay gran diferencia.

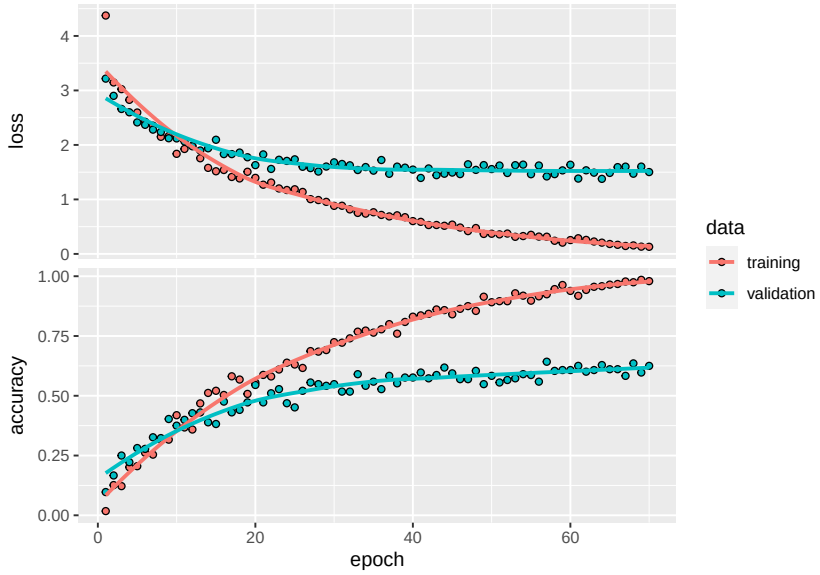

Figura 4.22: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada xception y tamaño de batch size 50. Fuente: Elaboración propia.

Evaluación del modelo

Procedemos a evaluar este modelo con las imágenes del conjunto de prueba.

```
model8 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.4538277 0.5881435
```

Aunque el `accuracy` de las imágenes de validación era más alto que en el modelo anterior, en los datos test esta tasa de acierto es menor, ronda un 59 % mientras que el modelo anterior estaba en un 62 % aproximadamente. Es decir, no se han producido cambios significativos entre un modelo y otro. Da de hecho la sensación de que este modelo presenta un sobreajuste mayor.

4.4.9. Red Xception y otras capas

En este modelo volvemos a usar los pesos de la red `xception` ya entrenada, lo único que cambiamos son las capas de después, tendremos más capas densas, para ver si este cambio en la arquitectura de las capas finales puede ayudar a la mejora de los resultados.

Definición del modelo

En este caso, las capas posteriores a la red preentrenada son las siguientes: la capa `layer_global_average_pooling_2d`, como antes, seguida de una capa densa con 1024 neuronas (es el valor por defecto, que se puede cambiar dándole otro valor al parámetro `n_dense`), otra capa densa con 750 neuronas, una capa de activación `relu`, una capa `dropout`, otra capa densa de 250 neuronas y una última capa `softmax` que nos devolverá una matriz con las probabilidades de cada clase.

```
model_function3 <- function(learning_rate = 0.001,
  dropoutrate=0.2, n_dense=1024){

  k_clear_session()

  model <- keras_model_sequential() %>%
    mod_base %>%
    layer_global_average_pooling_2d() %>%
    layer_dense(units = n_dense) %>%
    layer_dense(units=750) %>%
    layer_activation("relu") %>%
    layer_dropout(dropoutrate) %>%
    layer_dense(units=250) %>%
    layer_dense(units=output_n, activation="softmax")

  model %>% compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = learning_rate),
    metrics = "accuracy"
  )

  return(model)

}
```

```
model9 <- model_function3()
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #     Trainable
## ================================================================================
##  xception (Functional)              (None, 5, 5, 2048)              20861480    N
##  global_average_pooling2d (Glo      (None, 2048)                    0           Y
##  balAveragePooling2D)
##  dense_3 (Dense)                    (None, 1024)                    2098176     Y
##  dense_2 (Dense)                    (None, 750)                     768750      Y
##  activation (Activation)            (None, 750)                     0           Y
##  dropout (Dropout)                  (None, 750)                     0           Y
##  dense_1 (Dense)                    (None, 250)                     187750      Y
##  dense (Dense)                      (None, 31)                      7781        Y
## ================================================================================
## Total params: 23,923,937
## Trainable params: 3,062,457
## Non-trainable params: 20,861,480
## ________________________________________________________________________________
```

Podemos observar como en este modelo se necesitan $23,923.937$ parámetros, que es una gran cantidad. Pero, como hemos utilizado una red ya entrenada solo necesitamos entrenar $3,062.457$ parámetros pues el resto ya lo estarían. Es decir, la red ya tiene entrenados $20,861.480$ parámetros.

En la primera capa densa, tal y como hemos visto anteriormente (en el modelo de la sección 4.4.7), requerimos de 2,098.176 parámetros.

En la segunda capa densa necesitamos 750 × 1.024 parámetros para almacenar los pesos de las 750 neuronas más los 750 parámetros adicionales para los sesgos.

En la tercera capa densa requerimos 250 × 750 parámetros para almacenar los pesos de las 250 neuronas además de los 250 parámetros adicionales para los sesgos.

En la última capa densa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 250 nodos de la capa anterior, y por tanto se requieren 31×250 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

El resto de capas que no necesitan parámetros son porque estas capas no modifican la forma de la red, simplemente estas capas utilizan operaciones matemáticas sin necesidad de nuevos parámetros para hallar los pesos y sesgos.

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function3`, ya indicamos cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Procedemos a realizar el entrenamiento de la red, con tamaño de `batch_size` 50 y 70 `epochs`.

```
batch_size <- 50
epochs <- 70
hist9 <- model9 %>% fit(
  train_images,
  steps_per_epoch = train_images$n / batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.7499
##     accuracy: 0.7407
##     val_loss: 2.589
## val_accuracy: 0.5
```

El gráfico 4.23 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de un 50 % en el conjunto de datos de validación. Esto empeora los modelos que habíamos visto con esta arquitectura `xception`. El añadir estas capas densas no ha sido eficaz en este caso.

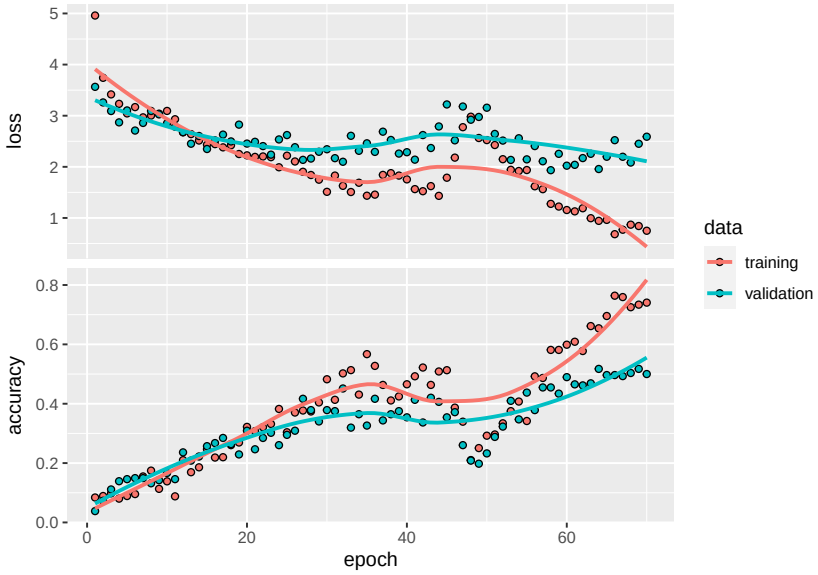

Figura 4.23: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada xception y algunas modificaciones añadiendo más capas. Fuente: Elaboración propia.

Viendo los gráficos podríamos haber añadido más `epochs`, es decir, seguir entrenando el modelo, pero no nos garantiza una gran mejora. Además, como tenemos una red más compleja, es más costoso en cuanto a tiempo de computación y memoria.

Evaluación del modelo

Vamos a evaluar esta red con los datos de prueba.

```
model9 %>% evaluate(test_images)
```

```
##     loss accuracy
## 2.315252 0.500780
```

Hemos conseguido una tasa de acierto de un 50 % en la clasificación de las imágenes de prueba. Es un poco peor a los modelos con redes preentrenadas que hemos realizado.

4.4.10. Red neuronal convolucional con la arquitectura DenseNet

Como ya sabemos, existen redes preentrenadas, otra red muy popular es `DenseNet`. Es un tipo de red neuronal convolucional que utiliza conexiones densas entre capas, a través de bloques densos, donde conectamos todas las capas directamente entre sí.

Definición del modelo

En `DenseNet` existen distintas redes. En este caso cargamos la red `densenet121` con los pesos previamente entrenados en el conjunto de datos de `ImageNet`, excepto la capa final (que clasifica las imágenes en el conjunto de datos de `ImageNet`) que entrenaremos en nuestro propio conjunto de datos. Una vez obtenemos los pesos los "congelamos" para que en la red convolucional que diseñemos esta parte esté ya entrenada.

Tenemos, como siempre, que indicar la forma de entrada de nuestros datos, es decir, $160 \times 160 \times 3$, ya que nuestras imágenes son de dimensión 160×160 píxeles y como son a color tienen 3 canales.

```
mod_base2 <- application_densenet121(weights = 'imagenet',
   include_top = FALSE, input_shape = c(width, height, 3))
freeze_weights(mod_base2)
```

A continuación creamos la función `model_function4`, que es igual a funciones anteriores vistas, lo único que cambia es la red preentrenada inicialmente, que como ya hemos dicho esta será `densenet121`. En este caso, en lugar de usar `mod_base`, usaremos `mod_base2`.

```
model10 <- model_function4()
```

```
## Model: "sequential"
## _________________________________________________________________________
##  Layer (type)                   Output Shape              Param #    Trainable
## =========================================================================
##  densenet121 (Functional)       (None, 5, 5, 1024)        7037504    N
##  global_average_pooling2d (Glo  (None, 1024)              0          Y
##  balAveragePooling2D)
##  dense_1 (Dense)                (None, 1024)              1049600    Y
##  activation (Activation)        (None, 1024)              0          Y
##  dropout (Dropout)              (None, 1024)              0          Y
##  dense (Dense)                  (None, 31)                31775      Y
## =========================================================================
## Total params: 8,118,879
## Trainable params: 1,081,375
## Non-trainable params: 7,037,504
## _________________________________________________________________________
```

Podemos observar como en este modelo se necesitan $8,118.879$ parámetros, que es una gran cantidad pero comparando con la arquitectura `Xception` son muchísimos menos con diferencia. Al usar esta red preentrenada solo necesitamos entrenar $1,081.375$ parámetros pues el resto ya lo estarían. Es decir, la red ya tiene entrenados $7,037.504$ parámetros.

En la primera capa densa, tal y como hemos visto anteriormente (en el modelo de la sección 4.4.7), requerimos de $2,098.176$ parámetros. En la última capa densa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 1.024 nodos de la capa anterior, y por tanto se requieren 31×1.024 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

El resto de capas que no necesitan parámetros son porque estas capas no modifican la forma de la red, simplemente estas capas utilizan operaciones matemáticas sin necesidad de nuevos parámetros para hallar los pesos y sesgos.

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function3`, ya indicamos cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Procedemos a entrenar este modelo con tamaño de `batch_size` 32 y con 70 `epochs`.

```
batch_size <- 32
epochs <- 70
hist10 <- model10 %>% fit(
  train_images,
  steps_per_epoch = train_images$n / batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.2979
##     accuracy: 0.9198
##     val_loss: 0.9392
## val_accuracy: 0.7098
```

El gráfico 4.24 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 71 % en el conjunto de datos de validación después de 70 `epochs`.

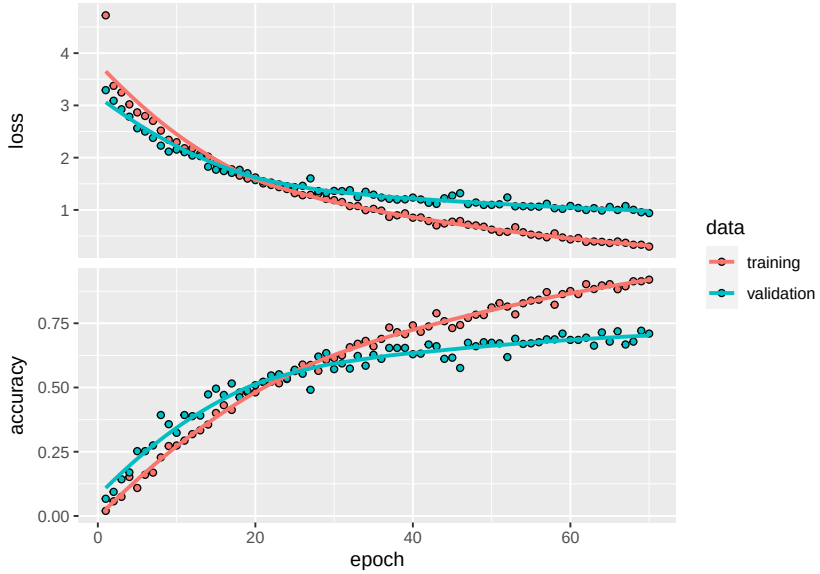

Figura 4.24: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada densenet121. Fuente: Elaboración propia.

Tabla 4.7: Porcentaje de acierto en las predicciones con una red neuronal convolucional con la arquitectura DenseNet (densenet121).

Persona	Porcentaje de acierto
Margot Robbie	13.64
Akshay Kumar	40.00
Marmik	44.44
Elizabeth Olsen	45.00
Ellen Degeneres	90.91
Dwayne Johnson	92.86
Roger Federer	92.86
Amitabh Bachchan	92.86

Evaluación del modelo y predicciones

Procedemos a evaluar el modelo con los datos de prueba, como siempre:

```
model10 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.0220077 0.6770671
```

Hemos conseguido el modelo más eficaz hasta ahora, obteniendo alrededor de un 68 % de acierto en la clasificación de las imágenes de prueba. Parece que esta arquitectura de red neuronal convolucional es más eficiente que `Xception`.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test*, reflejado en la tabla 4.7. La tasa de acierto más baja es de un 13.64 %, seguida de porcentajes de acierto que rondan del 40 % al 45 % (siendo estos más precisos que en otros modelos vistos).

Finalmente podemos observar la matriz de confusión de la figura 4.25 para ver mejor los fallos que se producen en las predicciones. Se vuelve a dar el caso de confundir a Margot Robbie (nivel 21) y Claire Holt (nivel 10), ambas mujeres rubias con ojos claros. Es por ello que la tasa de acierto de las imágenes de Margot Robbie es peor, como se ve en la tabla 4.7.

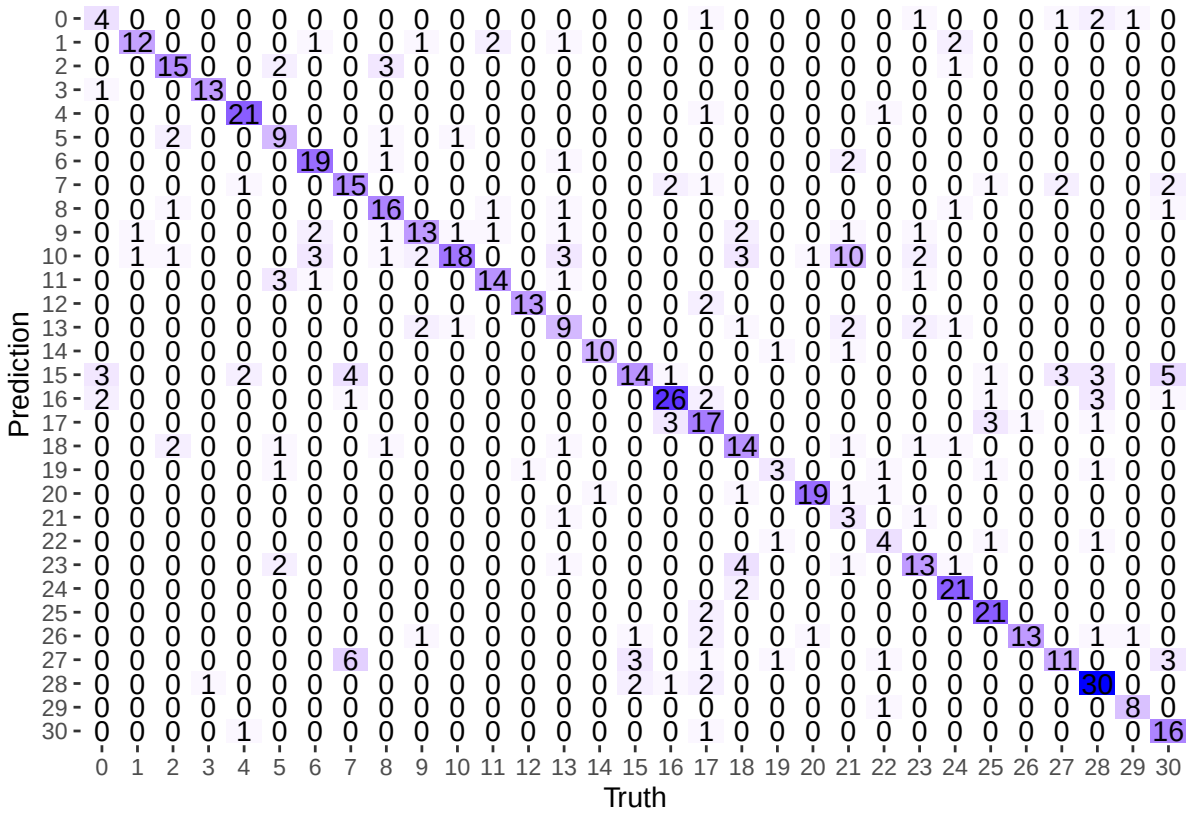

Figura 4.25: Matriz de confusión de las predicciones mediante la red neuronal convolucional diseñada con la arquitectura DenseNet (densenet121). Fuente: Elaboración propia.

4.4.11. Red neuronal convolucional con otra arquitectura de DenseNet

Como hemos dicho anteriormente, en DenseNet existen distintas redes. En este caso usaremos la red `densenet169`. Probamos con esta nueva arquitectura por si arroja resultados aún más prometedores que los anteriores, ya que en la documentación de Keras [3] aparece que dicha arquitectura suele tener algo más de precisión que `densenet121`.

Definición del modelo

A continuación definimos la red neuronal convolucional como siempre, pero esta vez usando la red `densenet169`.

```
mod_base3 <- application_densenet169(weights = 'imagenet',
   include_top = FALSE, input_shape = c(width, height, 3))
freeze_weights(mod_base3)
```

```
model_function5<- function(learning_rate = 0.001,
  dropoutrate=0.2, n_dense=1024){

  k_clear_session()

  model <- keras_model_sequential() %>%
    mod_base3 %>%
```

```
    layer_global_average_pooling_2d() %>%
    layer_dense(units = n_dense) %>%
    layer_activation("relu") %>%
    layer_dropout(dropoutrate) %>%
    layer_dense(units=output_n, activation="softmax")

  model %>% compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_adam(learning_rate = learning_rate),
    metrics = "accuracy"
  )

  return(model)

}
```

```
model11 <- model_function5()
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #     Trainable
## ================================================================================
##  densenet169 (Functional)           (None, 5, 5, 1664)              12642880    N
##  global_average_pooling2d (Glo      (None, 1664)                    0           Y
##  balAveragePooling2D)
##  dense_1 (Dense)                    (None, 1024)                    1704960     Y
##  activation (Activation)            (None, 1024)                    0           Y
##  dropout (Dropout)                  (None, 1024)                    0           Y
##  dense (Dense)                      (None, 31)                      31775       Y
## ================================================================================
## Total params: 14,379,615
## Trainable params: 1,736,735
## Non-trainable params: 12,642,880
## ________________________________________________________________________________
```

Podemos observar como en este modelo se necesitan $14,379.615$ parámetros. Al usar esta red preentrenada solo necesitamos entrenar $1,736.735$ parámetros pues el resto ya lo estarían. Es decir, la red ya tiene entrenados $12,642,880$ parámetros.

En la primera capa densa, por cada neurona i (entre 0 y 1023) requerimos 1.664 parámetros para los pesos w_{ij} y por tanto 1.664×1.024 parámetros para almacenar los pesos de las 1.024 neuronas. Además de los 1.024 parámetros adicionales para los 1.024 sesgos b_j correspondientes a cada una de ellas. En la última capa densa, al ser una función `softmax`, se requiere conectar todos sus 31 nodos con los 1.024 nodos de la capa anterior, y por tanto se requieren 31×1.024 parámetros w_i además de los correspondientes 31 sesgos b_j correspondientes a cada nodo.

El resto de capas que no necesitan parámetros son porque estas capas no modifican la forma de la red, simplemente estas capas utilizan operaciones matemáticas sin necesidad de nuevos parámetros para hallar los pesos y sesgos.

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function5`, ya indicamos cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Procedemos a entrenar este modelo con tamaño de `batch_size` 32 y con 80 epochs.

```
batch_size <- 32
epochs <- 80
hist11 <- model11 %>% fit(
  train_images,
  steps_per_epoch = train_images$n / batch_size,
  epochs = epochs,
  validation_data = validation_images,
  validation_steps = validation_images$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.1347
##     accuracy: 0.9733
##     val_loss: 0.9871
## val_accuracy: 0.7299
```

El gráfico 4.26 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 73 % en el conjunto de datos de validación después de 80 `epochs`. Esto supone una pequeña mejora en la precisión que el modelo anterior con la red `densenet121`.

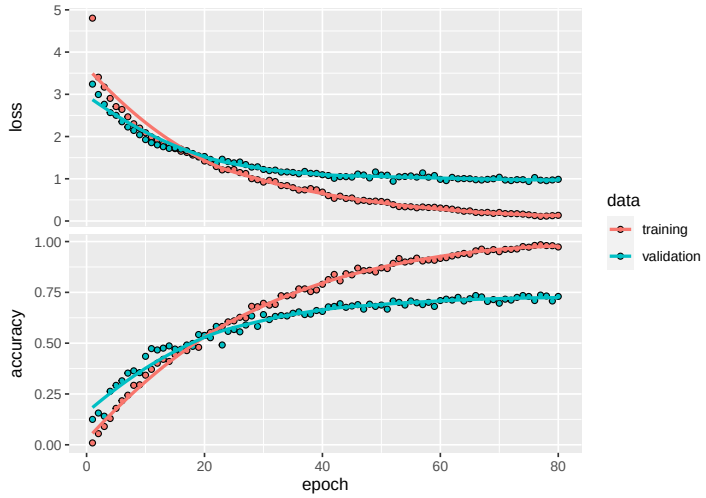

Figura 4.26: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada densenet169. Fuente: Elaboración propia.

Tabla 4.8: Porcentaje de acierto en las predicciones con una red neuronal convolucional con la arquitectura DenseNet (densenet169).

Persona	Porcentaje de acierto
Alia Bhatt	14.29
Charlize Theron	26.32
Akshay Kumar	40.00
Elizabeth Olsen	40.00
Dwayne Johnson	92.86
Kashyap	100.00
Roger Federer	100.00
Amitabh Bachchan	100.00

Evaluación del modelo y predicciones

Ahora procedemos a evaluar el modelo con las imágenes *test*.

```
model11 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.0932726 0.6911076
```

Obtenemos alrededor de un 69 % de acierto en la clasificación de las imágenes *test*. Parece que esta red de `Densenet` es algo mejor que la anterior, aunque no con mucha diferencia.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test*, reflejado en la tabla 4.8. Aquí conseguimos varias personas en las que todas sus imágenes son clasificadas correctamente.

Además, podemos observar la matriz de confusión de la figura 4.27 para ver mejor los fallos que se producen en las predicciones. Podemos observar que este modelo suele confundir a Alia Bhatt (nivel 2) con Anushka Sharma (nivel 5). Estas mujeres son muy parecidas ya que ambas tienen una forma del rostro similar y además, son las dos morenas con ojos castaños.

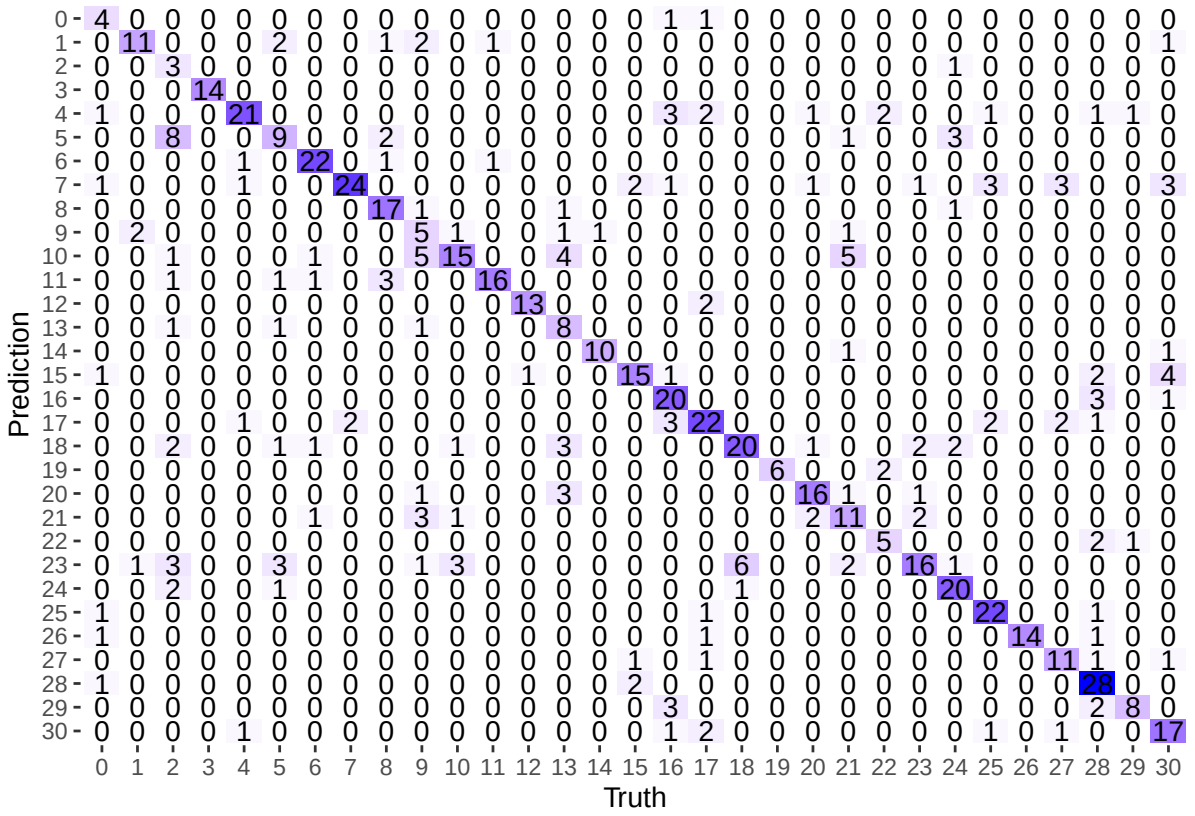

Figura 4.27: Matriz de confusión de las predicciones mediante la red neuronal convolucional diseñada con la arquitectura DenseNet (densenet169). Fuente: Elaboración propia.

4.4.12. Modelo con los datos de entrenamiento equilibrados

En esta sección lo que hacemos es equilibrar los datos de entrenamiento, es decir, queremos haya la misma proporción de imágenes por persona. En otras palabras, queremos que haya el mismo número de imágenes de cada persona. En la tabla 4.2 podemos observar que el número de imágenes por persona no está igualado, la persona que tiene menos imágenes juntando los datos de entrenamiento y validación es Marmik (23 imágenes) y la persona con más es Brad Pitt (94 imágenes).

Luego, en vez de reducir el número de imágenes, que nos produciría perdida de información, aumentaremos el número de imágenes en el resto de personas, copiando aleatoriamente imágenes que ya teníamos hasta llegar a 94.

En primer lugar copiamos y pegamos la carpeta `train`, a esta copia le llamaremos `train2` y sobre ella haremos que se ajuste la proporción de imágenes por persona.

Preparación y procesamiento de datos

Vamos a preparar los datos para su posterior implementación en Keras. En primer lugar, creamos un vector llamado `number_photos` que va almacenando en orden el número de imágenes de cada perosna. Una vez tenemos el vector, hallamos el máximo que en este caso es 94.

```
path<-"C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/train2"
```

```
ld<-list.dirs(path)[-1]

number_photos<-c()
for (i in ld) {
 number_photos<-c(number_photos,length(list.files(i)))
}
number_photos
max(number_photos)
```

A continuación creamos un `data.frame` con 3 variables: una corresponde con el nombre de la persona, otra con el número de imágenes que tiene dicha persona y la última con el número de fotos que hay que añadir para tener 94 fotos de esa persona.

```
listnames<-list.files(path)
photos_people<-cbind(listnames,number_photos) %>% as.data.frame()
photos_people$number_photos<-as.numeric(photos_people$number_photos)
photos_people$add_photos<-94-number_photos
```

Con el siguiente bucle hacemos que todas las personas tengan el mismo número de imágenes. El bucle escoge imágenes aleatorias con reemplazamiento y las copia en su carpeta correspondiente.

```
set.seed(1234)
for (i in ld){
  lf<-list.files(i)
  size_to_copy<-94-length(lf)
  if (size_to_copy > 0) {
    a<-sample(lf,size_to_copy,replace = TRUE)
    for (j in 1:size_to_copy){
      file.copy(paste0(i,"/",a[j]),paste0(i,"/",a[j],"_",j,".jpg"))
    }
  }
}
```

A continuación, en el vector `number_photos2` almacenamos cuantas imágenes hay de cada persona. Si cada coordenada del vector es 94 significa que hemos hecho bien el proceso de equilibrar el número de fotos por persona. Efectivamente, lo hemos hecho bien.

```
number_photos2<-c()
for (i in ld) {
  number_photos2<-c(number_photos2,length(list.files(i)))
}
number_photos2
```

Vamos a cargar los datos de entrenamiento (que separemos una parte para validación) y test, lo único que cambiamos es la ruta de la carpeta de entrenamiento, que seleccionaremos la carpeta que acabamos de construir con la misma proporción de imágenes por persona. El generador es el mismo que antes, que reescala los valores de los píxeles a valores entre 0 y 1 y, reserva un 25 % de los datos para validación.

```
path_train2 <- "C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/train2"
train_data_gen <- image_data_generator(rescale = 1/255,
  validation_split = .25)
```

A continuación, almacenamos todas las imágenes para el entrenamiento y la validación, al igual que en la sección 4.3.

```
train_images2 <- flow_images_from_directory(path_train2,
  train_data_gen,
  subset = 'training',
  target_size = target_size,
  class_mode = "categorical",
  shuffle=F,
  classes = label_list,
  seed = 1234)

validation_images2 <- flow_images_from_directory(path_train2,
 train_data_gen,
  subset = 'validation',
  target_size = target_size,
  class_mode = "categorical",
  classes = label_list,
  seed = 1234)
```

El conjunto de imágenes *test* no hace falta volverlo a definir puesto que es el mismo que ya utilizábamos.

Definición del modelo

El modelo que definimos es el mismo que ya hemos usado anteriormente, con la arquitectura `Densenet` y la red `densenet169`, ya que parece que ha sido la red que se ajusta mejor a las imágenes.

```
model12 <- model_function5()
```

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function5`, ya estaba indicado cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Hay que tener en cuenta que al igualar el número de imágenes por persona, esto hará que tengamos muchas más imágenes en el entrenamiento y ello provocará que dicho entrenamiento en la red neuronal convolucional sea más lento y costoso computacionalmente. En este caso el tamaño de `batch_sizees` 32 y el número de `epochs` es 80.

```
batch_size <- 32
epochs <- 80
hist12 <- model12 %>% fit(
  train_images2,
```

```
  steps_per_epoch = train_images2$n / batch_size,
  epochs = epochs,
  validation_data = validation_images2,
  validation_steps = validation_images2$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.0286
##     accuracy: 0.9977
##     val_loss: 1.313
## val_accuracy: 0.7031
```

El gráfico 4.28 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 70 % en el conjunto de datos de validación después de 80 `epochs`.

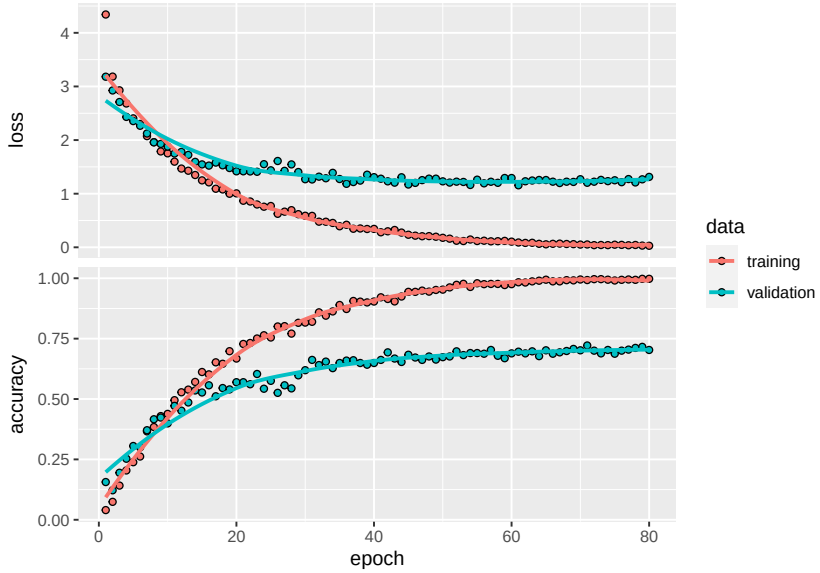

Figura 4.28: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada densenet169 y datos de entrenamiento equilibrados. Fuente: Elaboración propia.

Evaluación del modelo y predicciones

Procedemos a evaluar las imágenes de prueba.

```
model12 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.1829401 0.6942278
```

Obtenemos alrededor de un 69 % de acierto en la clasificación de las imágenes *test*.

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test*, reflejado en la tabla 4.9. Destacar que está

Tabla 4.9: Porcentaje de acierto en las predicciones con una red neuronal convolucional con la arquitectura DenseNet (densenet169) entrenada con el mismo número de imágenes por persona.

Persona	Porcentaje de acierto
Akshay Kumar	30.00
Elizabeth Olsen	30.00
Alia Bhatt	33.33
Anushka Sharma	38.89
Roger Federer	92.86
Brad Pitt	96.15
Kashyap	100.00
Amitabh Bachchan	100.00

bastante bien, comparando con modelos anteriores, que las tasas más bajas de acierto ronden el 30 %.

Finalmente podemos observar la matriz de confusión de la figura 4.29 para ver mejor los fallos que se producen en las predicciones. Se suele confundir a Jessica Alba (nivel 18) con Natalie Portman (nivel 23), ambas mujeres morenas con ojos castaños.

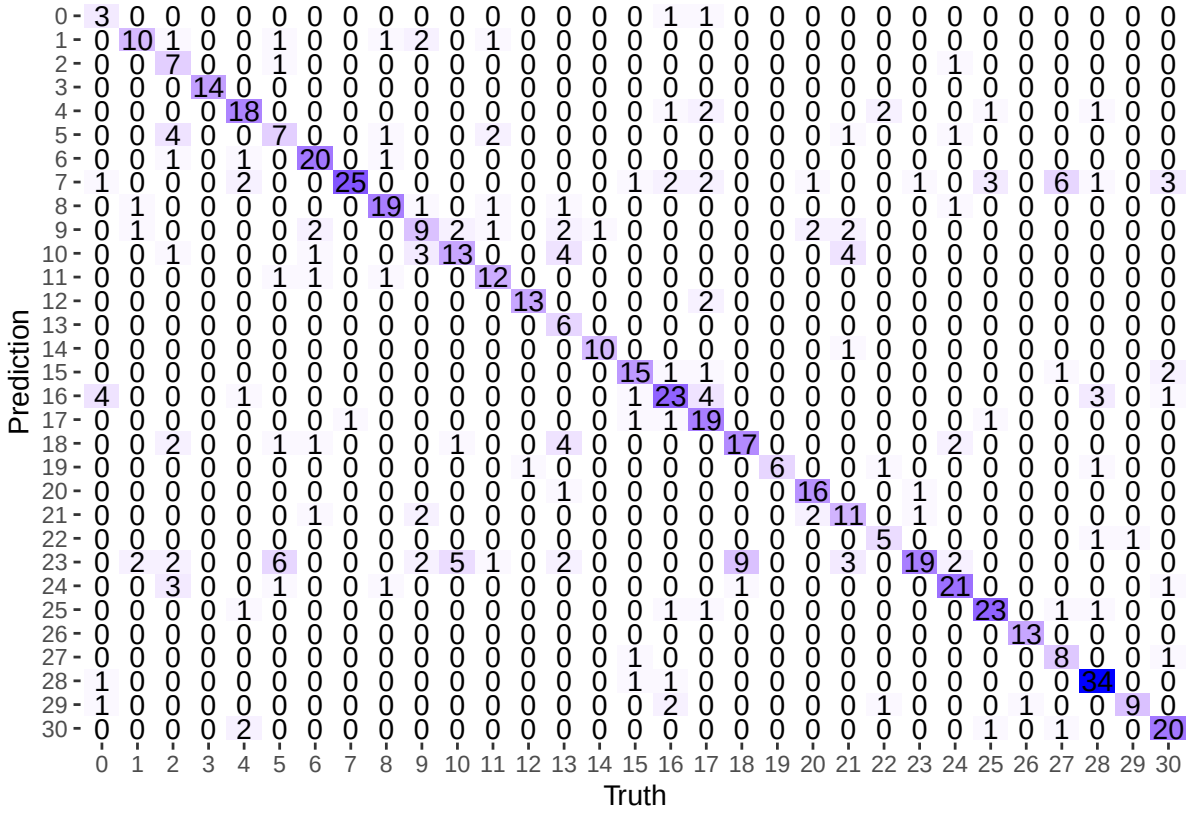

Figura 4.29: Matriz de confusión de las predicciones mediante la red neuronal convolucional diseñada con la arquitectura DenseNet (densenet169) entrenada con el mismo número de imágenes por persona. Fuente: Elaboración propia.

4.4.13. Modelo usando data augmentation

En el modelo anterior lo que hemos hecho ha sido igualar las proporciones de imágenes por persona, ahora lo que hacemos va a ser aumentar el número de imágenes modificando las imágenes originales mediante transformaciones como son hacer *zoom*, escalado, *flip* horizontal y un largo etcétera.

Esta técnica es usada para *dataset* poco numerosos, permitiendo disponer de un número mayor de imágenes a entrenar y permitiendo al modelo reconocer imágenes "no perfectas" al realizar las transformaciones.

Preparación y procesado de datos

Esta vez modificamos el generador de imágenes, usando la función `image_data_generator` para definir el preprocesamiento de los datos. Con ella reescalaremos los valores de los píxeles a valores entre 0 y 1, le pedimos que aparte de entrenar la imagen original, entrene con la misma imagen pero transformada mediante *zoom*, rotándola 5 grados, haciendo un *flip* horizontal e indicamos que un 25 % de los datos será para validación.

```
path_train <- "C:/Users/usuario/Desktop/TFG_Rocio_Ruiz/data/train"
train_data_gen3 <- image_data_generator(rescale = 1/255, zoom_range = 0.2,
                                        rotation_range = 5,
                                        horizontal_flip = TRUE,
                                        validation_split = .25)
```

Luego ahora leemos nuestras imágenes de entrenamiento y validación con este generador.

```
train_images3 <- flow_images_from_directory(path_train,
  train_data_gen3,
  subset = 'training',
  target_size = target_size,
  class_mode = "categorical",
  shuffle=F,
  classes = label_list,
  seed = 1234)

validation_images3 <- flow_images_from_directory(path_train,
 train_data_gen3,
  subset = 'validation',
  target_size = target_size,
  class_mode = "categorical",
  classes = label_list,
  seed = 1234)
```

Definición del modelo

El modelo que definimos es el mismo que ya hemos usado anteriormente, con la arquitectura `Densenet` y la red `densenet169`, ya que parece que ha sido la red que se ajusta mejor a las imágenes.

```
model13 <- model_function5()
```

Configuración del proceso de aprendizaje

En la función que hemos definido, `model_function5()`, ya estaba indicado cómo va a ser el proceso de aprendizaje. Podemos observar que la función de `loss` es `categorical_crossentropy`, el optimizador es `adam` con `learning rate` 0.001 y la métrica es `accuracy`.

Entrenamiento del modelo

Hay que tener en cuenta que al aumentar el número de imágenes (ya que aparte de entrenar la imagen original, entrenamos la misma imagen pero transformada mediante *zoom*, rotándola 5 grados y haciendo un *flip* horizontal), provocará que dicho entrenamiento de la red neuronal convolucional sea más costoso computacionalmente en cuanto a tiempo y memoria.

```
batch_size <- 32
epochs <- 80
hist13 <- model13 %>% fit(
  train_images3,
  steps_per_epoch = train_images3$n / batch_size,
  epochs = epochs,
  validation_data = validation_images3,
  validation_steps = validation_images3$n / batch_size,
  verbose = 2
)
```

```
##
## Final epoch (plot to see history):
##         loss: 0.6502
##     accuracy: 0.7855
##     val_loss: 1.162
## val_accuracy: 0.6518
```

El gráfico 4.30 muestra la evolución de la precisión y la pérdida en el conjunto de entrenamiento y validación. En este caso, logramos una precisión de aproximadamente un 65 % en el conjunto de datos de validación después de 80 `epochs`. Esto supone una pequeña mejora en la precisión del modelo de validación con respecto al modelo anterior, pero no hay gran diferencia.

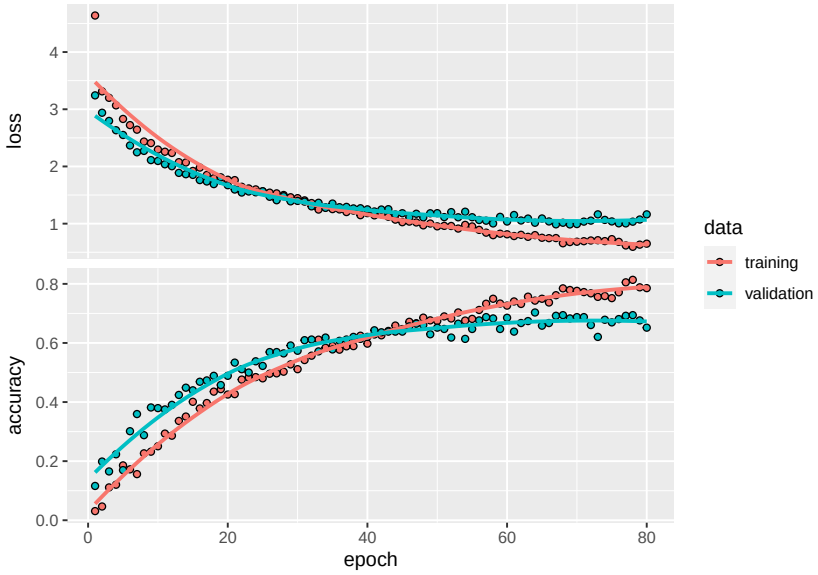

Figura 4.30: Curvas de la función loss y accuracy de una red neuronal convolucional con la red preentrenada densenet169 y realizado data augmentation en los datos de entrenamiento y validación. Fuente: Elaboración propia.

Si hubiéramos añadido 20 `epochs` más, tendríamos el siguiente resultado:

```
##
## Final epoch (plot to see history):
##         loss: 0.4399
##     accuracy: 0.8608
##     val_loss: 0.9305
## val_accuracy: 0.7098
```

Es decir, aumentando en 20 el número de `epochs` entrenados, obtenemos una precisión alrededor del 71 %, que está bastante bien.

Si volviésemos a aumentar en 20 `epochs`:

```
##
## Final epoch (plot to see history):
##         loss: 0.3464
##     accuracy: 0.8924
##     val_loss: 0.9349
## val_accuracy: 0.7098
```

Seguimos obteniendo una precisión del 71 %.

Luego podemos quedarnos con el modelo que hemos diseñado al haberlo entrenado durante 100 `epochs`.

Tabla 4.10: Porcentaje de acierto en las predicciones con una red neuronal convolucional con la arquitectura DenseNet (densenet169) y data augmentation.

Persona	Porcentaje de acierto
Elizabeth Olsen	25.00
Anushka Sharma	27.78
Claire Holt	28.57
Kashyap	50.00
Dwayne Johnson	92.86
Roger Federer	92.86
Ellen Degeneres	100.00
Amitabh Bachchan	100.00

Evaluación del modelo y predicciones

Procedemos a evaluar la red con las imágenes del conjunto de prueba (*test*).

```
model13 %>% evaluate(test_images)
```

```
##      loss  accuracy
## 1.2417477 0.6146646
```

Obtenemos alrededor de un 61 % de acierto en la clasificación de las imágenes *test*.

Con 20 `epochs` más, logramos aproximadamente un 72 % de precisión.

```
##      loss  accuracy
## 0.9953518 0.7223089
```

Procedemos a ver el porcentaje de acierto de las personas con más y menos acierto en cuanto a la clasificación de imágenes *test*, reflejado en la tabla 4.10. Nos centraremos en la red tras hacer un total de 100 `epochs`. Aunque la tasa más baja de acierto en una persona ronde el 25 %, menos que en el modelo anterior, en general, este modelo es más preciso en cuanto a la clasificación de todas las imágenes.

Adicionalmente, podemos observar la matriz de confusión de la figura 4.31 para ver mejor los fallos que se producen en las predicciones. Podemos ver como es usual confundir a Elizabeth Olsen (nivel 13) con Jessica Alba (nivel 18).

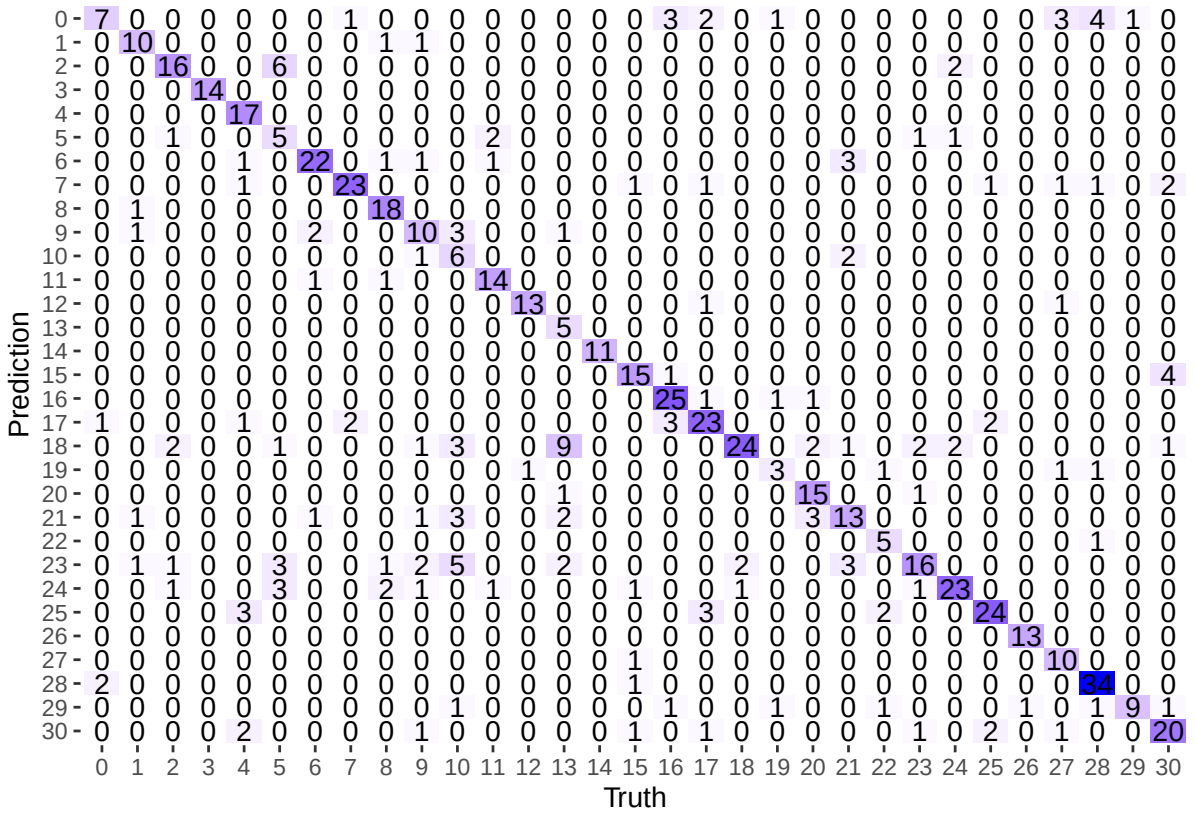

Figura 4.31: Matriz de confusión de las predicciones mediante la red neuronal convolucional diseñada con la arquitectura DenseNet (densenet169) y data augmentation. Fuente: Elaboración propia.

4.4.14. Comparativa y análisis de los modelos anteriores

En esta sección vamos a comparar los modelos en los que hemos obtenido mejores resultados. Estos modelos son los de las secciones 4.4.10, 4.4.11, 4.4.12 y 4.4.13. Además, expondremos los resultados más llamativos.

Podemos ver en la tabla 4.11 un resumen con los resultados obtenidos al evaluar cada uno de estos modelos.

Tabla 4.11: Resultados de la evaluación de los modelos con mejores resultados.

	Loss	Accuracy
Modelo 10	1.0220077	0.6770671
Modelo 11	1.0932726	0.6911076
Modelo 12	1.1829401	0.6942278
Modelo 13_2	0.9953518	0.7223089

Como podemos observar, estos modelos tienen un porcentaje de acierto bastante parecido y los valores de la función `loss` también son similares. Pero podemos apreciar que el último modelo es algo más preciso. Recordemos que este modelo era la red neuronal convolucional con la arquitectura de la red preentrenada `densenet169` realizando `Data Augmentation`. Esta técnica nos permite aumentar en tamaño y en diversidad nuestros datos generando nuevos ejemplos a partir de transformaciones en los originales, siendo en nuestro caso esas transformaciones hacer `zoom`, rotarla 5 grados y realizar un *flip* horizontal. En general, utilizar `data augmentation` nos permite mejorar sustancialmente nuestros modelos y, en este estudio, hemos visto que con esta técnica obtenemos los mejores resultados.

Si nos fijamos en el porcentaje de acierto por persona, hemos visto cómo han ido mejorando en algunos casos. Hay veces que el modelo no consigue acertar ninguna imagen de algunas personas, como es el caso del modelo de la sección 4.4.4, en la que tenemos tres personas (Akshay Kumar, Dwayne Johnson y Elizabeth Olsen) para las que ninguna de sus imágenes en el conjunto test es clasificada correctamente. En cambio, todas las fotos de Amitabh Bachchan las clasifica correctamente.

A su vez, hay otros modelos como el de la sección 4.4.6, que aciertan clasificando al menos 2 imágenes de cada persona. Los porcentajes de acierto más bajos en este caso varían del 30 % al 36.36 % (que se corresponden con las personas Akshay Kumar, Billie Eilish y Margot Robbie). Sin embargo, el porcentaje más alto de acierto no es el 100 %, sino el 85.71 % (este porcentaje también coincide con Amitabh Bachchan).

Si nos fijamos, hay personas que suelen ser casi siempre fáciles de identificar como son el caso de Roger Federer y Amitabh Bachchan. Esto puede ser debido a que en todas las imágenes que hay en la base de datos, Roger Federer sale con una banda en la cabeza y, Amitabh Bachchan sale siempre con una barba blanca y gafas negras muy características (figura 4.32).

Figura 4.32: Ejemplo de personas que son fáciles de identificar. Fuente: Elaboración propia.

En cambio, hay personas que cuestan más ser identificadas debido a que las imágenes que disponemos son muy variadas: con o sin gafas de sol, con o sin barba, con el pelo corto o largo, de diferentes épocas, etc; un ejemplo de ello son las imágenes de Akshay Kumar (figura 4.33).

Figura 4.33: Ejemplo de persona difícil de identificar. Fuente: Elaboración propia.

Si nos fijamos en las distintas matrices de confusión que hemos obtenido de cada modelo, podemos ver que son habituales fallos en personas que se parecen mucho, como es el caso de Margot Robbie (nivel 21) con Claire Holt (nivel 10) que ambas son mujeres rubias con ojos claros. Otro caso habitual de fallo es confundir Jessica Alba (nivel 18) con Natalie Portman (nivel 23).

Para poder aplicar nuestro modelo en aeropuertos, podríamos estudiar la probabilidad que tiene esa persona de parecerse a un delincuente de nuestra base de datos. Se debería poner un umbral, es decir, si por ejemplo dicha persona coincide en un 65 % o más con un delincuente de nuestro sistema, se le retendría para proceder a una mejor identificación para asegurarnos, o incluso desde un umbral más pequeño podría saltarle una alarma a los vigilantes de seguridad para que atendieran inmediatamente a las imágenes antes de realizar ninguna acción. Como ejemplo tomamos una imagen aleatoria de nuestros datos y vemos el porcentaje de parecido con cada persona. En este caso tomaremos una imagen de Brad Pitt obteniendo los siguientes resultados:

Tabla 4.12: Porcentaje de parecido con cada persona.

	Persona	Probability
8	Brad Pitt	77.9109418 %
16	Henry Cavill	12.2676618 %
28	Tom Cruise	6.3594572 %
15	Ellen Degeneres	2.6804425 %
18	Hugh Jackman	0.3986722 %

Figura 4.34: Imagen de Brad Pitt [24].

Es decir, en este caso se retendría a la persona por parecerse en más de un 65 % a una de nuestra base de imágenes de delincuentes.

Capítulo 5

Herramientas software

En este capítulo se pretende explicar las herramientas en las cuales se ha apoyado el desarrollo de este trabajo.

5.1. Rstudio

RStudio [26] es un entorno de desarrollo integrado (IDE) para R. Incluye una consola, un editor de resaltado de sintaxis que admite la ejecución directa de código, así como herramientas para el trazado, el historial, la depuración y la gestión del espacio de trabajo.

RStudio está disponible en ediciones comerciales y de código abierto y se ejecuta en el escritorio (Windows, Mac y Linux) o en un navegador conectado a RStudio Server o RStudio Workbench (Debian/Ubuntu, Red Hat/CentOS y SUSE Linux).

La misión de RStudio es crear software gratuito y de código abierto para la ciencia de datos, la investigación científica y la comunicación técnica. Esto se hace para mejorar la producción y el consumo de conocimiento por parte de todos, independientemente de los medios económicos, y para facilitar la colaboración y la investigación reproducible, los cuales son fundamentales para la integridad y eficacia del trabajo en la ciencia, la educación, el gobierno y la industria.

RStudio también produce RStudio Team , una plataforma modular de productos de software comercial que brinda a las organizaciones la confianza para adoptar R, Python y otro software de ciencia de datos de código abierto a escala, en beneficio de muchas personas, para aprovechar grandes cantidades de datos, para integrar con los sistemas, plataformas y procesos empresariales existentes, o cumplir con las prácticas y estándares de seguridad, junto con los servicios en línea para que sea más fácil aprenderlos y usarlos en la web.

Juntos, el software de fuente abierta y el software comercial de RStudio forman un círculo virtuoso: la adopción de software de ciencia de datos de fuente abierta a escala en las organizaciones crea demanda para el software comercial de RStudio; y los ingresos del software comercial, a su vez, permiten una mayor inversión en el software de código abierto que beneficia a todos.

Este trabajo está realizado en RStudio, todo el desarrollo tanto teórico como práctico. Para ello, hemos hecho uso de distintas bibliotecas como son:

– `opencv`: biblioteca libre de visión artificial que nos ha servido para la detección facial. Se describe en detalle en la siguiente sección.

– `magick`: es una biblioteca de código abierto para gráficos avanzados y procesamiento de imágenes en R. Admite muchos formatos comunes (png, jpeg, tiff, pdf, etc.) y manipulaciones (rotar, escalar, recortar, recortar, voltear, desenfocar, etc.). Las imágenes se previsualizan automáticamente cuando se imprimen en la consola, lo que da como resultado un entorno de edición interactivo.

– `tidyverse`: es una colección de paquetes de R diseñados para la ciencia de datos. Todos los paquetes comparten una filosofía de diseño, gramática y estructuras de datos subyacentes orientados a la manipulación, importación, exploración y visualización de datos. Está compuesto de los por los paquetes `readr`, `dplyr`, `ggplot2`, `tibble`, `tidyr`, `purr`, `stringr` y `forcats`. Para más información visitar su página web o artículos sobre esta biblioteca ([34]).

– `dplyr`: es uno de los paquetes principales de `tidyverse`. Es principalmente un conjunto de funciones diseñadas para permitir la manipulación de marcos de datos de una manera intuitiva y fácil de usar. Los analistas de datos suelen utilizar `dplyr` para transformar los conjuntos de datos existentes en un formato más adecuado para algún tipo particular de análisis o visualización de datos. Algunas de las funciones que hemos usado han sido `select`, `arrange`, `mutate` y `summarise`.

– `tidymodels`: el ecosistema `tidymodels` es una colección de paquetes para modelización y aprendizaje automático utilizando los principios de `tidyverse`. Dentro de esta biblioteca encontramos paquetes como `rsample` (para realizar distintos tipos de remuestreo), `recipes` (para organizar las transformaciones del procesamiento de datos para que sea reproducible, soportando lo que solemos conocer como ingeniería de características), `parsnip` (para crear modelos) y `yardstick` (para las métricas del modelo). En este trabajo como se explicó anteriormente, usábamos la función `initial_split` para poder separar nuestras imágenes en entrenamiento y test. En su página web podemos hallar más información de esta colección de paquetes y sus funciones [18].

– `knitr`: es una herramienta para generación de informes dinámicos en R. Es un paquete en el lenguaje de programación estadístico R que permite integrar R en documentos LaTeX, LyX, HTML, Markdown, AsciiDoc, y reStructuredText. En nuestro caso, hemos usado este paquete para incluir imágenes o tablas, con las funciones `include_graphics` y `kable` respectivamente.

– `kableExtra`: el objetivo de esta biblioteca es construir tablas complejas y manipular estilos de tablas. En nuestro caso hemos mejorado el estilo de algunas tablas construidas con `kable` del paquete `knitr`, gracias a funciones como `kable_styling`.

– `ggplot2`: es un paquete de gráficos que proporciona comandos útiles para poder realizar gráficos complejos de los datos. Proporciona una interfaz más programática para especificar qué variables trazar, cómo se muestran y las propiedades visuales generales. Con esta biblioteca los gráficos que se realizan tienen una mejor visualización y entendimiento.

– `keras`: biblioteca que nos permite diseñar las redes neuronales. Está explicada con más detalle en la sección 5.3.

– `tensorflow`: es una biblioteca de código abierto para el cálculo numérico mediante gráficos de flujo de datos. Los nodos del gráfico representan operaciones matemáticas, mientras que los bordes del gráfico representan las matrices de datos multidimensionales (tensores) que se comunican entre ellos. La arquitectura flexible le permite implementar computación en una o más CPU o GPU en una computadora de escritorio, servidor o dispositivo móvil con una sola API. La hemos implementado para que funcione correctamente `keras`. Para más documentación se puede ver en la página de TensorFlow para Rstudio [4].

– `reticulate`: este paquete lo hemos ejecutado para el correcto funcionamiento de Keras. Proporciona un conjunto completo de herramientas para intercambiar información entre Python y R.

Para más información sobre RStudio, consultar la página oficial [26].

Añadir que se ha utilizado una plantilla para escribir el Trabajo de Fin de Grado con R Markdown (dicha plantilla la podemos encontrar en la bibliografía [20]).

5.2. OpenCV

OpenCV [35] es una biblioteca libre de visión artificial originalmente desarrollada por Intel. El nombre de esta biblioteca significa *Open Computer Vision* (Visión Artificial Abierta). La biblioteca tiene más de 2500 algoritmos optimizados, que incluyen un conjunto completo de algoritmos de aprendizaje automático y visión por computadora clásicos y de última generación. Se ha usado en muchísimas aplicaciones, siendo esta biblioteca la más popular de visión artificial.

Empresas conocidas como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda o Toyota emplean esta biblioteca. Algunos de los usos de OpenCV son unir imágenes de *streetview*, detectar intrusiones en vídeos de vigilancia en Israel, monitorear equipos mineros en China, ayudar a los robots a navegar y recoger objetos en Willow Garage, detectar accidentes por ahogamiento en piscinas en Europa, detección de movimiento, reconocimiento de objetos, reconstrucción 3D a partir de imágenes y un largo etcétera.

Su popularidad es debida a que:

– Es una biblioteca libre que sirve para ser utilizada con propósitos comerciales y de investigación.

– Es multiplataforma, estando disponible para los sistemas operativos GNU/Linux, Mac OS X, Windows y Android, y para diversas arquitecturas de hardware como x86, x64 (PC), ARM (celulares y Raspberry Pi).

– Está documentada y existen tutoriales, libros y diversos sitios para la formación.

OpenCV está totalmente desarrollado en C++, orientado a objetos y con alta eficiencia computacional. Su API es C++ pero incluye conectores para otros lenguajes, como R, que es el que hemos utilizado en este trabajo.

Las funciones que hemos utilizado están explicadas en la sección 4.3.

Toda la documentación de OpenCV se encuentra en su página oficial [1]. Para el ejemplo realizado nos ha servido el blog para R indicado en [2].

5.3. Keras

Toda la documentación de Keras en RStudio la podemos encontrar en su manual [3], en el que podemos encontrar documentación acerca de todas sus funciones, tutoriales, artículos y ejemplos.

Keras es una API de redes neuronales de alto nivel desarrollada con un enfoque basado en permitir la experimentación rápida. Tiene las siguientes características:

– Permite que el mismo código se ejecute en la CPU o en la GPU, sin problemas.

– Es una API fácil de usar que facilita la creación rápida de prototipos de modelos de aprendizaje profundo.

– Presenta una alta compatibilidad integrada con redes convolucionales (para visión artificial), redes recurrentes (para procesamiento de secuencias) y cualquier combinación de ambas.

– Admite arquitecturas de red arbitrarias: modelos de múltiples entradas o múltiples salidas, uso compartido de capas, uso compartido de modelos, etc. Esto significa que Keras es apropiado para construir esencialmente cualquier modelo de aprendizaje profundo, desde una red de memoria hasta una máquina neuronal de Turing.

Para la documentación completa de Keras podemos visitar el sitio web de `tensorflow` [4].

A continuación se explican los parámetros que más han sido utilizados en la construcción de redes con Keras:

– `input_shape`: en Keras, la capa de entrada es un tensor. Tenemos que especificar siempre en la primera capa la forma del tensor inicial, que debe tener la misma forma que los datos de entrenamiento. En nuestro caso, teníamos imágenes a color (3 canales según sistema RGB) de dimensiones 160×160 píxeles, es decir, nuestro tensor es de la forma `(ancho, alto, canales)=(160, 160, 3)`. Una vez demos la forma del tensor de entrada, Keras calculará el resto de formas en las posteriores capas automáticamente.

– `units`: son la cantidad de neuronas que indicamos que tiene la capa en cuestión. Las unidades de cada capa definirán la forma del tensor en esa capa.

– `activation`: cada neurona tiene una función de activación que define su salida. La función de activación se usa para introducir la no linealidad en las capacidades de modelización de la red. Las más utilizadas que están implementadas en Keras son *linear*, *sigmoid*, *tanh*, *softmax* y *ReLU*.

– `epochs`: con este parámetro estamos indicando el número de veces que usaremos todos los datos en el proceso de aprendizaje.

– `batch_size`: con este parámetro se indica el número de datos que usaremos para cada actualización de los parámetros del modelo.

– `loss`: es un parámetro que representa la penalización de una predicción. Si su valor es cero significaría que la predicción es perfecta. Es por ello, que la función `loss` se intenta minimizar y, en cambio, tratamos de obtener el máximo `accuracy` (la máxima precisión). Algunas funciones de `loss` que encontramos en Keras son `binary_crossentropy`, `categorical_crossentropy` (la que hemos usado),

`mean_absolute_error`, `mean_absolute_percentage_error`, `mean_squared_error` y `mean_squared_logarithmic_error`.

– `optimizer`: es la manera que tenemos de especificar el algoritmo de optimización que permite a la red neuronal calcular los pesos de los parámetros a partir de los datos de entrada y de la función de `loss` definida. Keras dispone de algunos optimizadores como son SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam. Los que hemos usado nosotros en este trabajo han sido SGD (Stochastic Gradient Descent) y Adam.

– `metrics`: una métrica sirve para juzgar el rendimiento del modelo. La usamos para monitorizar el proceso de aprendizaje (y prueba) de nuestra red neuronal. En nuestro caso hemos utilizado `accuracy`, que nos muestra la fracción de imágenes que han sido correctamente clasificadas. Pero existen muchas otras métricas que ofrece Keras como son `AUC`, falsos negativos, falsos positivos, la media y el error absoluto medio, aunque existen muchas más.

– `weights`: los pesos se calcularán de forma totalmente automática en función de las formas de entrada y salida. Existen pesos de redes neuronales ya entrenadas, algo que hemos utilizado en nuestro estudio, ya que nos ahorra bastante tiempo a la hora de entrenar las redes.

Algunas de las funciones que hemos utilizado han sido las siguientes:

– `image_data_generator`: genera lotes de imagen con aumento de datos en tiempo real. Los datos se repetirán (en lotes). Con aumento de datos nos referimos a que esta función tiene argumentos para poder realizar `data augmentation` como hicimos en la sección 4.4.13. En el caso en que estemos definiendo un generador de imágenes para el entrenamiento, tenemos la opción de indicarle que guarde una proporción de datos para la validación.

– `flow_images_from_directory`: genera lotes de datos a partir de imágenes en un directorio (con datos aumentados/normalizados opcionales). En nuestro caso, teníamos directorios diferentes para las imágenes de entrenamiento y las imágenes test. Además, en el caso de usar el generador de entrenamiento, podíamos acceder a los datos de entrenamiento o datos de validación.

– `keras_model_sequential`: es la estructura principal de Keras, que permite la creación de una red neuronal básica, pudiendo añadir una capa detrás de otra de forma sencilla. Se considera como una secuencia de capas que cada una de ellas va "destilando" gradualmente los datos de entrada para obtener la salida deseada.

– `layer_reshape`: reforma una salida a una determinada forma.

– `layer_dense`: sirve para añadir una capa densa en nuestra red. Debemos indicarle el número de neuronas que tendrá esa capa.

– `compile`: configura el modelo creado con Keras para el entrenamiento. Los argumentos básicos son los que hemos estado utilizando, `optimizer`, `loss` y `metrics`.

– `fit`: con esta función se entrena el modelo para un número fijo de `epochs` (iteraciones completas sobre el conjunto de datos entrenamiento). Tiene distintos argumentos opcionales que se pueden añadir como ya hemos visto.

– `evaluate`: evalúa un modelo de Keras. Para evaluar un modelo que ya ha sido entrenado, lo que se utiliza es un conjunto de datos test, es decir, un conjunto de imágenes que no hayan aparecido durante el entrenamiento del modelo.

– `layer_conv_2d`: una capa convolucional 2D se define con esta función en Keras. Esta capa crea un núcleo de convolución que se convoluciona con la entrada de la capa para producir un tensor de salidas. Los dos argumentos principales para la capa son `filters` y `kernel_size`.

– `layer_max_pooling_2d`: es una capa *pooling*, reduce la muestra de la entrada a lo largo de sus dimensiones espaciales (alto y ancho) tomando el valor máximo sobre una ventana de entrada (de tamaño definido por *pool_size*) para cada canal de la entrada.

– `layer_flatten`: esta capa se utiliza para hacer que la entrada multidimensional sea unidimensional, comúnmente utilizada en la transición de la capa de convolución a la capa conectada completa.

– `application_xception`: las aplicaciones Keras son modelos de aprendizaje profundo que están disponibles junto con pesos preentrenados. Estos modelos se pueden utilizar para la predicción, la extracción de características y el ajuste fino. Un ejemplo de ello nos proporciona esta aplicación que instancia la arquitectura `Xception`.

– `freeze_weights`: congela los pesos en un modelo o capa para que ya no se puedan entrenar. Esta función la usábamos para congelar los pesos de redes ya entrenadas.

– `k_clear_session`: borra el gráfico del modelo actual y crea uno nuevo. Útil para evitar el desorden de modelos y/o capas antiguos.

– `layer_global_average_pooling_2d`: es una capa de operación de agrupación promedio global para datos espaciales.

– `layer_activation`: esta capa aplica una función de activación a una salida.

– `layer_dropout`: esta capa busca regularizar la red, previniendo el posible sobreajuste al otorgar a cada neurona una probabilidad de no activarse durante la fase de entrenamiento.

– `application_densenet121`: esta es otra aplicación que instancia la arquitectura `DenseNet`.

– `application_densenet169`: esta aplicación también instancia la arquitectura `DenseNet` pero, en general, suele obtener mayor precisión que la anterior.

Capítulo 6

Conclusiones

A lo largo del trabajo, hemos analizado el estado del arte de la biometría y, en particular, del reconocimiento facial. Además de estudiar las tecnologías necesarias para desarrollar un sistema de reconocimiento facial, se ha puesto en práctica el proceso completo para desarrollar dicho sistema, diseñando un clasificador que identifique a las personas.

Como hemos podido observar, no es nada fácil la tarea de construir dicho modelo de reconocimiento facial, ya que hay que tener en cuenta muchísimos factores, disponer de una buena base de datos de imágenes y tener un dispositivo con gran almacenamiento y potencia capaz de construir un modelo para la clasificación de personas. Además, existen diversos problemas asociadas al reconocimiento facial como son los cambios de apariencia (edad, barba, peinado, gafas, gorra, etc.), la iluminación del momento de la captura de la imagen, la orientación de la cara, problemas de suplantación, etc.

Con el presente trabajo hemos logrado conocer los distintos tipos de biometrías y sus características, así como las técnicas utilizadas en el proceso de reconocimiento de patrones. También hemos logrado poner en práctica, con ayuda de R, OpenCV y Keras, el proceso completo de un sistema de reconocimiento facial mediante redes neuronales convolucionales.

Además, hemos entrenando distintos modelos variando algunos parámetros como son las funciones de activación, el optimizador y sus hiperparámetros (`learning rate`, `momentum` y `decay`), el tamaño de `batch size` y el número de `epochs`.

En los modelos de redes neuronales construidos hemos hecho uso de distintas capas (capas densas, capas `dropout`, capas `flatten`, capas de convolución, capas de `max-pooling`, etc.), pero ha sido fundamental la utilización de técnicas como `Transfer Learning` (utilizando redes preentrenadas) y `Data Augmentation` para lograr mejores resultados. El uso de redes preentrenadas, como `DenseNet`, hizo mejorar la precisión en la identificación de personas y al incorporar `Data Augmentation`, logramos optimizarla (obteniendo una tasa de acierto del 72 % sobre el conjunto de imágenes de prueba).

Existen numerosas formas de aumentar tanto el rendimiento como la eficiencia de las redes construidas para el reconocimiento facial expuestas en este trabajo.

Como propuesta podríamos obtener más imágenes de cada persona, pero no solo con el rostro frontal, sino en distintas posiciones como de perfil, inclinadas, etcétera. También sería útil tener las fotos actualizadas, ya que la base de imágenes disponía de distintas fotografías de diferentes épocas y, como la apariencia no es invariante respecto al tiempo,

puede causar problemas a la hora del reconocimiento facial. Más aún, ya que el modelo en el que hemos obtenido mejores resultados ha sido en el que hemos aplicado `data augmentation`, sería interesante poder realizar más transformaciones aparte de las ya realizadas para intentar mejorar la precisión de dicho modelo.

Para llevar a cabo esta propuesta, sería de gran ayuda disponer de un equipo más potente, ya que he trabajado en la CPU debido a que mi dispositivo no disponía de una GPU con las características necesarias. Es por ello, que para entrenarse las redes se han necesitado numerosas horas. Disponer de GPU nos proporciona que el trabajo pueda distribuirse fácilmente entre múltiples chips o tarjetas gráficas, por lo que se reduce bastante el tiempo de estos procesos.

Si se quisiera llegar más lejos, podríamos poner en práctica tener otro sistema de identificación por si el reconocimiento facial fallara. Dicho sistema podría ser el reconocimiento de la huella digital, pero para ello sería conveniente tener un sensor y recopilar las huellas personalmente pidiendo autorización a las personas que se ofrezcan voluntarias para llevar a cabo dicho estudio.

Bibliografía

[1] (2014). *The OpenCV Reference Manual*. OpenCV, 2.4.13.7.ª edición.

[2] Abdullayev, Turgut (2020). «RStudio AI Blog: Deepfake detection challenge from R».
https://blogs.rstudio.com/ai/posts/2020-08-18-deepfake/.

[3] Allaire, JJ y Chollet, François (2022). *keras: R Interface to 'Keras'*.
https://keras.rstudio.com. R package version 2.8.0.9000.

[4] Allaire, JJ; Eddelbuettel, Dirk; Golding, Nick y Tang, Yuan (2015). *R Interface to TensorFlow*.
https://tensorflow.rstudio.com/.

[5] ApokalypseParty Team (2021). «How to build your own image recognition app with R!» https://www.r-bloggers.com/2021/03/how-to-build-your-own-image-recognition-app-with-r-part-1/.

[6] Bautista Gómez, Alejandro. *Detección, reconocimiento y seguimiento de rostros aplicando Redes Neuronales Convolucionales*. Tesis doctoral, Universidad de Sevilla.

[7] BeeDIGITAL (2019). «Historia y evolución del reconocimiento facial».

[8] Cech, Jan y Soukupova, Tereza (2016). «Real-Time Eye Blink detection using Facial Landmarks». *Cent. Mach. Perception, Dep. Cybern. Fac. Electr. Eng. Czech Tech. Univ. Prague*, pp. 1–8.

[9] Chollet, Francois (2021). *Deep learning with Python*. Simon and Schuster.

[10] Costa Mari, Daniel. *Análisis de un sistema de reconocimiento facial a partir de una base de datos realizado mediante Python*. Tesis doctoral, Universidad Politécnica de Cataluña.

[11] Durán Suárez, Jaime (2017). «Redes neuronales convolucionales en R: Reconocimiento de caracteres escritos a mano».

[12] Fchollet (2017). «Classifier from little data script».
https://gist.github.com/fchollet/0830affa1f7f19fd47b06d4cf89ed44d.

[13] Fontalvo Herrera, Tomás; De la Hoz Granadillo, Efraín y Vergara, Juan Carlos (2012). «Aplicación de análisis discriminante para evaluar el mejoramiento de los indicadores financieros en las empresas del sector alimento de Barranquilla-Colombia». *Ingeniare. Revista chilena de ingeniería*, **20(3)**, pp. 320–330.

[14] Gurrea, Manuel (2000). «Análisis de componentes principales». *Proyecto e-Math Financiado por la Secretaría de Estado de Educación y Universidades (MECD)*.

[15] Jafri, Rabia y Arabnia, Hamid R (2009). «A survey of face recognition techniques». *journal of information processing systems*, **5(2)**, pp. 41–68.

[16] Jiménez Silva, Ildefonso. *Reconocimiento facial basado en redes neuronales convolucionales*. Tesis doctoral, Universidad de Sevilla.

[17] Jones, Michael y Viola, Paul (2003). «Fast multi-view face detection». *Mitsubishi Electric Research Lab TR-20003-96*, **3(14)**, p. 2.

[18] Kuhn, Max y Wickham, Hadley (2020). *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles.*.
https://www.tidymodels.org.

[19] Lerma, Joaquín Planells y Palacios, Roberto Paredes (2009). «Implementación del algoritmo de detección facial de viola-jones». *Documento*, **1**, p. 23.

[20] Luque-Calvo, Pedro L. (2017). *Escribir un Trabajo Fin de Estudios con R Markdown*.

[21] Ooms, Jeroen y Wijffels, Jan (2022). *opencv: Bindings to 'OpenCV' Computer Vision Library*. Https://docs.ropensci.org/opencv/ https://github.com/ropensci/opencv.

[22] Otero, Digna María González (2011). «BibTeX: Gestión de bibliografía en LATEX». https://www.utm.mx/.

[23] Papageorgiou, Constantine P; Oren, Michael y Poggio, Tomaso (1998). «A general framework for object detection». En: *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pp. 555–562. IEEE.

[24] PATEL, VASUKI (2020). «Face Recognition Dataset». https://www.kaggle.com/datasets/vasukipatel/face-recognition-dataset.

[25] Romero Escuntar, Karla Mariana (2006). *Reconocimiento de rostros en tiempo real utilizando una red neuronal*. B.S. thesis, QUITO/EPN/2006.

[26] RStudio Team (2020). *RStudio: Integrated Development Environment for R*. RStudio, PBC., Boston, MA.
http://www.rstudio.com/.

[27] Ruesgas, Borja Simancas (2019). *Desarrollo de un sistema de identificación mediante técnicas de reconocimiento facial*. Máster, Universidad de León.

[28] Schapire, Robert E (2013). «Explaining adaboost». En: *Empirical inference*, pp. 37–52. Springer.

[29] Smilkov, Daniel y Carter, Shan. «A Neural Network Playground - TensorFlow». https://playground.tensorflow.org/.

[30] Tapiador Mateos, M.; Sigüenza Pizarro, J.A. y Colas Pasamontes, J. (2005). *Tecnologías biométricas aplicadas a la seguridad*. Editorial Ra-Ma.

[31] Torres, Jordi (2018). *Deep Learning – Introducción práctica con Keras(PRIMERA PARTE)*. Colección WATCH THIS SPACE. ISBN 978-0-244-07895-9. https://torres.ai/deep-learning-inteligencia-artificial-keras.

[32] Viola, Paul y Jones, Michael (2001). «Rapid object detection using a boosted cascade of simple features». En: *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, tomo 1, pp. I–I. Ieee.

[33] Wayman, James L (2001). «Fundamentals of biometric authentication technologies». *International Journal of Image and Graphics*, **1(01)**, pp. 93–113. doi: 10.1142/S0219467801000086.

[34] Wickham, Hadley; Averick, Mara; Bryan, Jennifer; Chang, Winston; McGowan, Lucy D'Agostino; François, Romain; Grolemund, Garrett; Hayes, Alex; Henry, Lionel; Hester, Jim; Kuhn, Max; Pedersen, Thomas Lin; Miller, Evan; Bache, Stephan Milton; Müller, Kirill; Ooms, Jeroen; Robinson, David; Seidel, Dana Paige; Spinu, Vitalie; Takahashi, Kohske; Vaughan, Davis; Wilke, Claus; Woo, Kara y Yutani, Hiroaki (2019). «Welcome to the tidyverse». *Journal of Open Source Software*, **4(43)**, p. 1686. doi: 10.21105/joss.01686.

[35] Wikipedia (2022). «OpenCV — Wikipedia, La enciclopedia libre». https://es.wikipedia.org/w/index.php?title=OpenCV&oldid=143077911.

[36] Woodward Jr, John D; Horn, Christopher; Gatune, Julius y Thomas, Aryn (2003). «Biometrics: A look at facial recognition». *Informe técnico*, RAND CORP SANTA MONICA CA.