

Reconocimiento de caras con uso de mascarilla



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Samuel Arévalo Cañestro

Tutor/es:

Francisco Antonio Pujol López



Universitat d'Alacant
Universidad de Alicante

Mayo 2021

Resumen

En el momento actual, la crisis producida por el SARS-CoV-2, el uso de mascarilla está a la orden del día. Debido a la falta de información y características producida por la mascarilla, los algoritmos de reconocimiento facial ven su precisión muy reducida.

El campo del reconocimiento facial está en constante desarrollo, es por esto por lo que múltiples equipos han dedicado sus esfuerzos a desarrollar sistemas de reconocimiento facial adaptados al uso de la mascarilla. Distintas aproximaciones al mismo problema han derivado en múltiples modelos de soluciones, desde machine learning tradicional hasta redes convolucionales complejas.

En este proyecto se ha propuesto realizar un estudio del estado del arte en el campo del reconocimiento facial, y más en profundidad sobre la variante con uso de mascarilla. Además, se propone el diseño de un modelo capaz de realizar reconocimiento facial sobre imágenes de caras cuyos rasgos faciales son obstruidos por el uso de la mascarilla. Este modelo ha sido desarrollado en Python y en su implementación se emplean las tecnologías Keras, Scikit-learn y Tensorflow, entre otras.

El algoritmo propuesto está compuesto por una red neuronal convolucional como extractor de características, un elemento PCA que reduce la dimensionalidad de las características obtenidas y, por último, un clasificador basado en machine learning. Se han realizado distintas implementaciones para, posteriormente, comparar los resultados de los diferentes modelos. Entre las redes convolucionales empleadas en los modelos implementados se encuentran las redes VGG, redes ResNet y una implementación de la red Inception. En cuanto a clasificadores se ha propuesto el uso de elementos random forest, support vector machines y multilayer perceptron.

Por último, se realiza un análisis de los resultados de los modelos implementados y proyectos similares, cuyos resultados son alentadores.

Índice de contenidos

Índice de contenidos	iv
Índice de figuras	vi
1. Introducción.....	1
2. Objetivos.....	1
3. Estado del arte	2
4. Tecnologías.....	5
4.1. Python 3	5
4.1.1. Tensorflow.....	6
4.1.2. Keras	6
4.1.3. OpenCV	7
4.1.4. NumPy.....	8
4.1.5. Scikit-learn	8
4.2. Google Colaboratory.....	9
4.2.1. Conexión con Drive.....	10
4.3. Dataset ORL	11
4.3.1. Wear-Mask-To-Face. ORL Masked.....	11
5. Diseño de modelos	15
5.1. CNN Feature Extractor	15
5.1.1. ResNet50	17
5.1.2. ResNet101	20
5.1.3. VGG-16	21
5.1.4. VGG-19	23
5.1.5. Inception V3	24
5.2. PCA.....	25
5.3. Clasificadores ML.....	26
5.3.1. MLP. Perceptrón multicapa.....	26

5.3.2.	Random forest	27
5.3.3.	Support Vector Machine.....	27
6.	Implementación	28
6.1.	Preparación de los datos	28
6.1.1.	Data augmentation.....	29
6.2.	Preparación de CNN	30
6.2.1.	Resnet50	31
6.2.2.	Resnet101	31
6.2.3.	VGG16	31
6.2.4.	VGG19	32
6.2.5.	InceptionV3	32
6.3.	Preparación del clasificador.....	32
6.3.1.	MLP	33
6.3.2.	Random Forest.....	33
6.3.3.	LinearSVC	34
6.4.	Extracción de resultados	34
7.	Resultados.....	37
7.1.	MLP	37
7.2.	Random forest.....	38
7.3.	SVM. LinearSVC.....	39
7.4.	Otros proyectos	40
8.	Conclusión.....	41
9.	Bibliografía y referencias	42
10.	Anexo I. Proyecto	44

Índice de figuras

Figura 3.1 [5]Estructura del método propuesto por Hongling Chen y Chen Haoyu	3
Figura 4.1. Logo de Python 3	5
Figura 4.2. Logotipo de Tensorflow	6
Figura 4.3. Logotipo de la librería Keras.....	6
Figura 4.4. Logotipo de OpenCV	7
Figura 4.5. Logotipo de NumPy	8
Figura 4.6. Logotipo de Scikit-learn.....	8
Figura 4.7. Logotipo de Google Colaboratory	9
Figura 4.8. Ejemplo de cuaderno de Google Colaboratory	9
Figura 4.9. Logotipo de Google Drive	10
Figura 4.10. Ejemplo de conexión a Drive a Colaboratory en código Python.....	10
Figura 4.11. Imágenes de ejemplo del dataset ORL.....	11
Figura 4.12. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada. ...	12
Figura 4.13. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada. ...	12
Figura 4.14. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada. ...	13
Figura 4.15. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada. ...	14
Figura 4.16. Ejemplo del dataset ORL_Combined creado a partir de Wear-Mask-To-Face	14
Figura 5.1. Esquema de funcionamiento de una CNN indicando la inutilización de su etapa de clasificación.	16
Figura 5.2. Ejemplo de parámetros entrenables/no-entrenables al crear una red VGG16 ..	16
Figura 5.3. Logotipo de ImageNet.....	16
Figura 5.4. Esquema de un bloque de procesamiento interno de una CNN.....	17
Figura 5.5. Esquema de obtención de $a[l+2]$ en CNN.....	17
Figuras 5.6 y 5.7. Esquema de un bloque de procesamiento residual de una CNN ResNet	18
Figura 5.8. Extracto de [11], diagrama de un bloque residual de ResNet.....	18
Figura 5.9. Esquema simplificado de un bloque correspondiente a la primera etapa de una red ResNet50.	19
Figura 5.10. Fragmento del esquema de la red ResNet50 empleada.....	19
Figura 5.11. Fragmento del esquema de la red ResNet101 empleada.....	20
Figura 5.12. Esquema simplificado de un bloque de red VGG.....	21
Figura 5.13. Esquema simplificado de la red VGG16.....	21
Figura 5.14. Esquema simplificado de la red VGG16 empleada	22

Figuras 5.15, 5.16, 5.17. Estructura real de la red VGG16 empleada	22
Figura 5.18. Esquema simplificado de la red VGG19.....	23
Figuras 5.19 y 5.20. Esquema real de la red VGG19 implementada.	23
Figura 5.21. Ejemplo simplificado de estructura de un bloque de InceptionV3	24
Figura 5.22. Mitad superior de un bloque de la red Inception usada en el proyecto.....	24
Figura 5.23. Mitad inferior de un bloque de la red Inception usada en el proyecto	25
Figura 5.24. Esquema simplificado de un perceptrón multicapa.	26
Figura 5.25. Esquema simplificado de un random forest genérico [18].....	27
Figura 6.1. Extracto de código correspondiente a la extracción del dataset.....	28
Figura 6.2. Extracto de código correspondiente al uso de objetos IDG	29
Figura 6.3. Extracto de código correspondiente a la creación de los conjuntos de datos....	30
Figura 6.4. Extracto de código correspondiente a la creación de la CNN ResNet50.....	31
Figura 6.5. Extracto de código correspondiente a la creación de la CNN ResNet101	31
Figura 6.6. Extracto de código correspondiente a la creación de la CNN VGG16.....	31
Figura 6.7. Extracto de código correspondiente a la creación de la CNN VGG16.....	32
Figura 6.8. Extracto de código correspondiente a la creación de la CNN InceptionV3.....	32
Figura 6.9. Extracto de código correspondiente al uso del objeto PCA.....	32
Figura 6.10. Extracto de código correspondiente a la creación y entrenamiento de MLP..	33
Figura 6.11. Extracto de código correspondiente a la creación y entrenamiento de RF	34
Figura 6.12. Extracto de código correspondiente a la predicción del conjunto de pruebas de RF	34
Figura 6.13. Extracto de código correspondiente a la creación y entrenamiento de SVM (LinearSVC)	34
Figura 6.14. Extracto de código correspondiente a la predicción del conjunto de pruebas de RF	34
Figura 6.15. Extracto de código correspondiente a la extracción de métricas	34
Figuras 6.16, 6.17. Tabla de ejemplo obtenida al emplear los métodos de la figura 6.15 ..	35
Figura 6.18. Fragmento de código donde se prueba una imagen al azar del conjunto de pruebas.....	35
Figura 6.19. Fragmento de código donde se prueba una imagen en concreto.....	36
Figura 7.1. Gráfico comparativo de precisión con clasificador MLP	37
Figura 7.2. Curvas ROC correspondientes a los 3 mejores algoritmos empleando MLP. De izquierda a derecha: VGG16, VGG19, Inception.	37
Figura 7.3. Gráfico comparativo de precisión con clasificador Random Forest	38

Figura 7.4. Curvas ROC correspondientes a los 3 mejores algoritmos empleando Random Forest. De izquierda a derecha: VGG16, ResNet101, Inception.....	38
Figura 7.5. Gráfico comparativo de precisión con clasificador SVC.....	39
Figura 7.4. Curvas ROC correspondientes a los 2 mejores algoritmos a partir de LinearSVC. De izquierda a derecha: VGG16, VGG19.....	39
Figura 7.5. Ejemplo de reconocimiento facial con mascarilla en el proyecto RWFMD [9]40	

1. Introducción

El uso de técnicas de reconocimiento facial está a la orden del día, ya sea para desbloquear un smartphone o para acceder a un lugar de trabajo restringido.

Debido a la situación mundial actual, la crisis del COVID-19, el uso de mascarillas es obligatorio en la gran parte de los países occidentales. Al intentar aplicar la gran mayoría de técnicas existentes de reconocimiento facial sobre una cara con mascarilla, se observa una gran disminución de su rendimiento. Esto es debido a la falta de información y características producida por la oclusión que realiza la mascarilla.

Por ello, varios grupos de investigadores han intentado adaptar las técnicas ya conocidas de reconocimiento facial al uso de la mascarilla. Distintas aproximaciones al mismo problema han derivado en múltiples modelos de soluciones, desde machine learning tradicional hasta redes convolucionales complejas.

2. Objetivos

El principal objetivo de este proyecto es intentar implementar un sistema capaz de aplicar reconocimiento facial en imágenes que contienen personas haciendo uso de mascarilla.

Además de este objetivo principal, el proyecto estará orientado a:

- Realizar un estudio sobre el estado del arte de la materia.
- Hacer uso de tecnologías actuales orientadas a la implementación de redes neuronales y sistemas de machine learning, tales como Tensorflow o Keras.
- Discutir los resultados y posibles mejoras a implementar.

En este documento se tratarán los objetivos realizados, comenzando por el estudio sobre el estado del arte actual en la materia de reconocimiento facial y su variante con uso de mascarilla. Posteriormente se comentarán las tecnologías empleadas en el desarrollo del proyecto. Después se explicará el diseño de las aproximaciones realizadas, y por último se detallará la implementación de estas. Para terminar, se reflexionará sobre los resultados obtenidos.

3. Estado del arte

El reconocimiento facial con uso de mascarilla es un problema que surge del reconocimiento facial básico. El reconocimiento facial se ha abarcado de distintas maneras a lo largo de la historia, los tipos de reconocimiento facial tradicional se pueden agrupar en holísticos y geométricos.

Los métodos tradicionales holísticos analizan la imagen completa como un conjunto buscando similitudes (*Template matching*). Este método como tal no se puede implementar a tiempo real debido a la inmensidad de datos a trabajar, ya que se considera cada píxel como una característica que el algoritmo debe estudiar. Debido a esto, se emplean otros métodos auxiliares para reducir el espacio facial a un número menor de coeficientes. De manera que se puede trabajar de manera eficiente sobre la imagen completa discriminando la información irrelevante. Algunos de estos métodos son el análisis de componentes principales (PCA) que emplea eigenfaces surgido en 1991 en [1], o el análisis lineal discriminante (LDA) que emplea fisherfaces surgido en 2009 en [2].

De estas técnicas anteriormente mencionadas, PCA proporciona un buen rendimiento debido a que transforma las características de la imagen en un conjunto de características significativas llamado eigenvector, posteriormente estos eigenvectores pueden ser empleados para obtener las conocidas como eigenfaces. Estas eigenfaces básicamente son las matrices de covarianzas de cada set de imágenes. Una vez que se realiza el cálculo de las eigenfaces la complejidad y coste de la tarea de reconocimiento facial se simplifica.

Por otro lado, los métodos tradicionales geométricos se basan en comparar características en subregiones de las imágenes. Tal como se explica en [3], se extraen lo que se consideran como puntos clave que posteriormente se emplearán junto con machine learning tradicional para realizar la clasificación. Sin embargo, a pesar de sus buenos resultados en perspectiva frontal, no es el método más eficiente si se compara con otros algoritmos que han surgido posteriormente.

En 2015, un equipo de Google Inc. presentó [4] donde daban a conocer Google FaceNet, un sistema que emplea una gran red neuronal convolucional (CNN) para realizar una distribución euclidiana de las imágenes. Según [4], tras realizar la distribución según parecido de caras, las demás tareas como reconocimiento facial o verificación de usuarios se vuelven sencillas. Al contrario que los métodos surgidos hasta la fecha, FaceNet no usa una

red de clasificación ya entrenada o PCA para obtener las características de una cara, sino que usa una capa cuya salida es el vector de características de 128 bytes por cara. Según [4] Google FaceNet estableció un récord en el campo del reconocimiento facial con un porcentaje de acierto del 99.63%.

En mayo de 2019, Hongling Chen y Chen Haoyu publican [5], donde exponen el desarrollo de un sistema de reconocimiento facial que alcanzaba el nivel de estado del arte. El sistema propuesto es una combinación de red convolucional, PCA y machine learning. Emplean como extractor de características una red VGGNet-16 ya entrenada, cuyas capas son congeladas, impidiendo el entrenamiento de sus pesos. Una vez obtenida la salida de la red se emplea PCA para conseguir la reducción de dimensionalidad del vector de características hasta un conjunto de 400 elementos. Tras reducir la dimensionalidad, se utiliza un support vector machine (SVM) para realizar la clasificación. Este sistema de clasificación obtuvo, según [5], una puntuación de precisión sobre el dataset LFW [20] de reconocimiento facial de 97.47%.

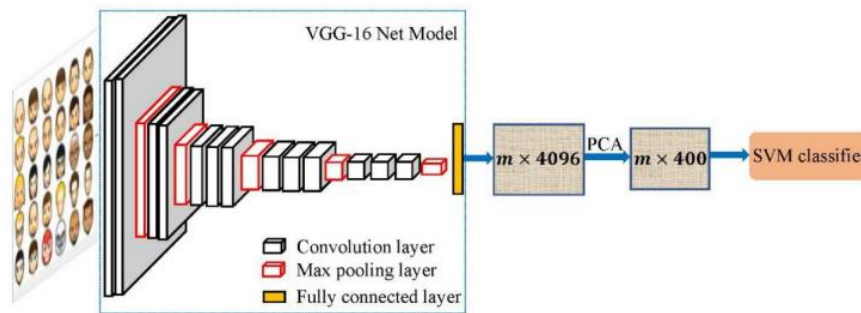


Figura 3.1 [5] Estructura del método propuesto por Hongling Chen y Chen Haoyu

También en mayo de 2019, un equipo compuesto por integrantes de distintas universidades de Bangladesh presentó a la Conferencia Internacional de Avances en Ciencia (ICASERT, en inglés) el proyecto “Implementation of Principal Component Analysis on Masked and Non-masked Face Recognition” [6]. Este trata de aplicar PCA y eigenfaces obtenidas para conseguir un sistema de reconocimiento facial que funcione correctamente teniendo en cuenta las complicaciones añadidas por el uso de mascarilla. En primer lugar, su sistema detecta la cara y la segmenta, posteriormente emplea el algoritmo PCA para obtener los eigenvalores y eigenvectores. Tras esto, se normalizan las eigenfaces obtenidas a partir de los eigenvectores de cada cara mediante las fórmulas de [6]. Una vez se consiguen las eigenfaces normalizadas, se puede calcular la predicción de manera simple realizando el cálculo entre la cara de la base de datos y la que se quiere contrastar obteniendo un peso, que será la decisión.

En [6] se expone, que el proyecto obtiene una precisión de reconocimiento con uso de mascarilla de 72%, mientras que sin uso de mascarilla su precisión aumenta hasta 95%. Aun así, en cuanto a reconocimiento facial este método sigue sin superar a Google FaceNet o el proyecto de Hongling Chen y Chen Haoyu, anteriormente comentado.

Por último, cabe destacar el proyecto de creación de un dataset con imágenes de personas con mascarilla RWMFD [9] desarrollado por miembros de la Universidad de Wuhan, entre otros. Dicho proyecto [9], no sólo ha conseguido crear un dataset extenso de imágenes con uso de mascarilla, sino que, además, han desarrollado un algoritmo de reconocimiento facial con uso de mascarilla con una precisión de, aproximadamente, un 95%. Este algoritmo está basado en discriminación de características multigranular, tomando como zonas clave los ojos y la zona visible de la cara.

Los resultados del equipo de desarrollo del RWMFD son alentadores, ya que no sólo se trata de reconocimiento facial, sino que consigue sobreponerse a las complicaciones que implica el uso de mascarilla.

4. Tecnologías

A continuación, se analizará en profundidad el conjunto de tecnologías empleadas en la realización del proyecto, desde las bibliotecas auxiliares más importantes hasta los datasets empleados.

4.1. Python 3



Figura 4.1. Logo de Python 3[23]

Como lenguaje de programación para el desarrollo del proyecto se ha escogido Python 3, la última versión de Python. Este lenguaje de programación es uno de los más versátiles que existen hoy en día.

Al ser un lenguaje interpretado, no requiere compilar el código, con lo cual podremos probar nuestro proyecto sin tener que esperar a generar un ejecutable mientras lo desarrollamos. Además, el código en Python es legible a simple vista, leer en Python es casi como leer lenguaje natural, lo que facilitará la posterior explicación de la implementación.

Debido a la inmensidad de herramientas disponibles, Python 3 es uno de los mejores lenguajes en los que desarrollar este tipo de proyectos basados en machine y deep learning. En nuestro proyecto nos apoyaremos en distintas herramientas que nos facilitarán las tareas a la hora de implementar redes neuronales o clasificadores y obtener estadísticas. Algunas de estas herramientas son Tensorflow, Keras, NumPy y Scikit-learn entre otras.

Emplearemos Python en su versión 3.7.10 para el desarrollo de nuestro proyecto.

4.1.1. Tensorflow



Figura 4.2. Logotipo de Tensorflow [24]

Tensorflow es, junto a Keras, la biblioteca dedicada al machine learning por excelencia. Presentada por Google en el año 2015, en [7] se puede obtener detalles específicos sobre su implementación. Esta librería permite al usuario medio implementar de manera sencilla algoritmos de machine learning, así como su ejecución.

Tensorflow es un sistema flexible, que permite desarrollar una gran variedad de algoritmos de machine learning. Emplearemos Tensorflow junto con Keras y Scikit-learn para implementar las redes neuronales convolucionales y clasificadores que emplearemos en nuestro sistema de reconocimiento. La versión empleada será Tensorflow 2.4.1.

4.1.2. Keras

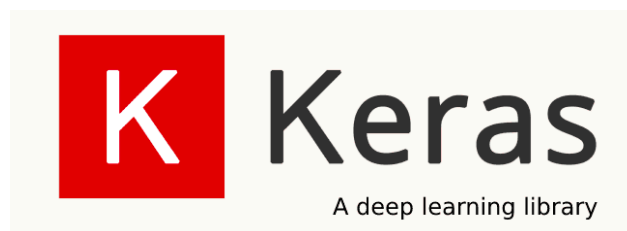


Figura 4.3. Logotipo de la librería Keras [25]

Keras es una biblioteca desarrollada en Python orientada al desarrollo de redes neuronales. Esta biblioteca contiene tanto herramientas para ayudar a los desarrolladores, como implementaciones de modelos conocidos listos para su uso. Keras permite a un usuario medio de Python aprender a usar redes neuronales en poco tiempo debido a la sencillez de su interfaz.

En 2017, el equipo de Google dedicado al desarrollo de Tensorflow comenzó a soportar Keras en la biblioteca principal Tensorflow.

En nuestro proyecto emplearemos Keras para desarrollar modelos de redes neuronales, la versión empleada es 2.4.3 junto a Tensorflow 2.4.1.

4.1.3. OpenCV



Figura 4.4. Logotipo de OpenCV[26]

OpenCV o OpenCV2, es una biblioteca de visión artificial y machine learning desarrollada por Intel que nos facilitará el tratamiento de imágenes empleadas por nuestro sistema de reconocimiento.

En nuestro caso, emplearemos algunos métodos de OpenCV para modificar las imágenes al realizar pruebas individuales.

La versión de OpenCV utilizada en el proyecto es 4.1.2.30.

4.1.4. NumPy



Figura 4.5. Logotipo de NumPy[27]

La biblioteca NumPy es una de las más usadas en el campo de la investigación. Provee a Python de arrays multidimensionales y objetos que derivan de estos, así como funciones y herramientas para su manipulación.

El objeto básico de NumPy es el `ndarray`, el array multidimensional. Este objeto encapsula arrays de n -dimensiones y permite al usuario emplear multitud de operaciones complejas sobre este tipo de objetos mediante una interfaz simple. Esto hace que esta biblioteca se casi indispensable en el ámbito de la investigación.

En el desarrollo de este proyecto se ha empleado la versión 1.19.5 de NumPy.

4.1.5. Scikit-learn



Figura 4.6. Logotipo de Scikit-learn[28]

Scikit-learn o Sklearn es una biblioteca de Python cuyo principal propósito es servir de ayuda a desarrolladores de sistemas de aprendizaje automático. Emplearemos esta librería en nuestro proyecto para obtener modelos de clasificadores de manera sencilla.

Para el desarrollo de nuestro proyecto se ha empleado la versión 0.22.2 de Scikit-learn.

4.2. Google Colaboratory



Figura 4.7. Logotipo de Google Colaboratory[29]

El desarrollo de este proyecto se ha llevado a cabo en empleando la plataforma Google Colaboratory. Esta plataforma ofrece a estudiantes, científicos e investigadores equipos remotos con gran capacidad computacional. Esto viene bien en caso de no disponer de un equipo potente, o trabajar desde distintas ubicaciones.

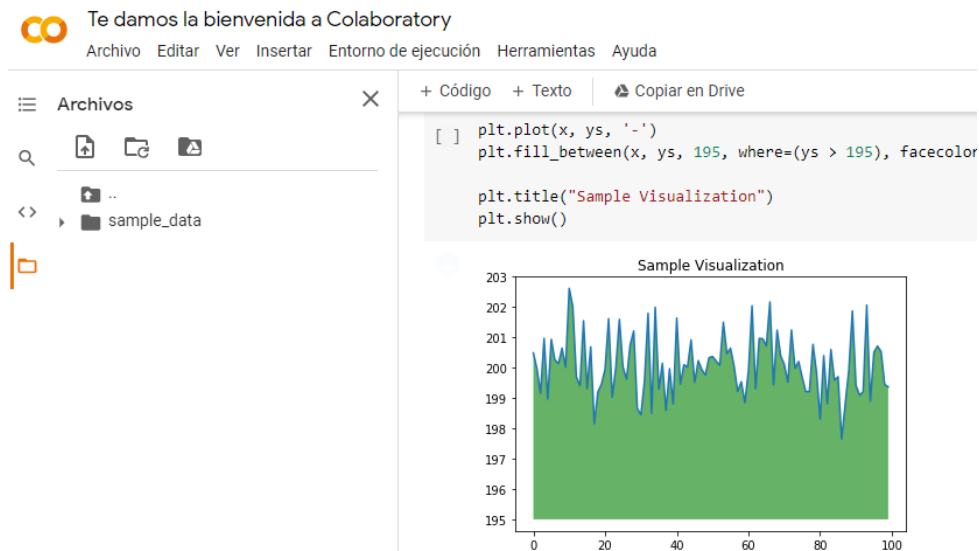


Figura 4.8. Ejemplo de cuaderno de Google Colaboratory

Google Colaboratory provee al usuario de una interfaz mediante la que implementar código en lenguaje Python, así como para supervisar su ejecución, empleando una interfaz de tipo cuaderno en el que se combina texto con código ejecutable. Se ha usado Colaboratory para realizar la implementación del proyecto en Python y Google Drive para almacenar los distintos documentos, ficheros, datasets y archivos del proyecto.

4.2.1. Conexión con Drive



Figura 4.9. Logotipo de Google Drive [30]

Colaboratory permite, mediante una biblioteca especial, conectar nuestro espacio de almacenamiento Google Drive a nuestra sesión remota en un cuaderno Colaboratory. De esta manera almacenaremos nuestros datasets en Google Drive, y posteriormente accederemos a ellos en nuestro código de Python empleando las herramientas facilitadas por Colaboratory.

El siguiente código permite conectar el almacenamiento Drive del usuario a la sesión de Google Colaboratory empleada. El almacenamiento Drive se acoplará como si se tratara de otra carpeta más dentro del sistema de archivos de la máquina de Colab.

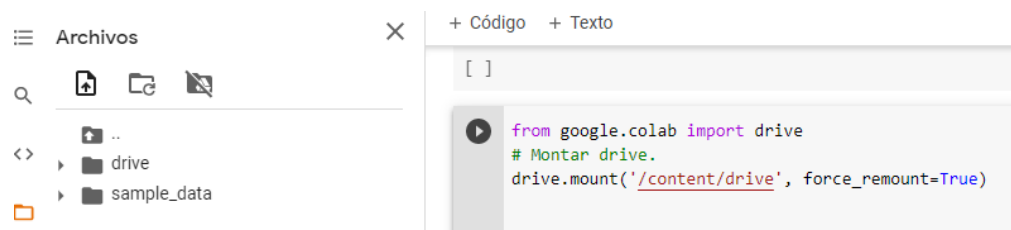


Figura 4.10. Ejemplo de conexión a Drive a Colaboratory en código Python

4.3. Dataset ORL

Trabajaremos con un dataset llamado ORL, un conjunto de imágenes tomadas entre 1992 y 1994 por la Universidad de Cambridge [8]. Este dataset contiene 400 imágenes correspondientes a 40 personas, 10 imágenes por sujeto.



Figura 4.11. Imágenes de ejemplo del dataset ORL.

Las imágenes fueron tomadas desde una perspectiva frontal, con distintas expresiones faciales, y tienen un tamaño de 92 x 112 píxeles en escala de grises.

4.3.1. Wear-Mask-To-Face. ORL Masked.

En las imágenes originales del dataset ORL los sujetos no portan mascarilla, por lo que inicialmente no servirían para realizar un sistema de reconocimiento facial con mascarilla. Debido a esto, se ha utilizado la herramienta Wear-Mask-To-Face desarrollada por el equipo de la Universidad de Wuhan creador del dataset Real-World Masked Face Dataset (RMFD) [9], podemos encontrar su implementación original en [10]. Esta herramienta permite modificar una imagen superponiendo una mascarilla a una cara en una imagen.

Se ha adaptado este programa para poder ejecutarlo en Google Colaboratory junto a los ficheros de Drive para construir nuestro dataset. Este programa recorre todo el directorio donde se encuentra el dataset y realiza un procedimiento para colocar la mascarilla artificial, además de escalar las imágenes a un tamaño de 224x224.

También se ha implementado una versión de este programa que no añade la mascarilla, sino que simplemente detecta las caras y realiza el escalado necesario en zonas correspondientes hasta obtener imágenes sin mascarilla de tamaño 224x224.

```

1 dataset_path = imageDirectory
2 save_dataset_path = saveDirectory
3 lastClass = "00"
4 images = 0
5
6 for root, dirs, files in os.walk(dataset_path, topdown=False):
7     for name in files:
8         new_root = root.replace(dataset_path, save_dataset_path)
9
10        imgPath = name
11        imgName, imgExt = os.path.splitext(imgPath)
12        imgSplit = imgName.split('_')
13        new_name = '{:0>3}'.format(imgSplit[0])+imgExt
14
15        lastClass = imgSplit[1]
16
17        new_root = new_root+'{:0>5}'.format(lastClass)
18
19
20        if not os.path.exists(new_root):
21            os.makedirs(new_root)
22
23
24        imgpath = os.path.join(root, name)
25        images += 1
26        save_imgpath = os.path.join(new_root, new_name)
27        cli(imgpath,save_imgpath)
28 print("Dataset construcion done. Found "+ str(images) + " files.")
29

```

Figura 4.12. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada.

La figura 4.12 realiza el acceso a las distintas imágenes de la carpeta indicada mientras que aplica el algoritmo descrito anteriormente mediante la función `cli()`, que a su vez emplea la clase *FaceMasker* y su consecuente procedimiento `.mask()`.

```

37 def cli(pic_path = './drive/MyDrive/TFG/Datasets/ORL_DB/100_10.jpg',save_pic_path
38
39     if not os.path.exists(pic_path):
40         print(f'Picture {pic_path} not exists.')
41         sys.exit(1)
42
43     mask_path = BLUE_IMAGE_PATH
44
45     FaceMasker(pic_path, mask_path, True, 'hog',save_pic_path).mask()
46

```

Figura 4.13. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada.

El procedimiento comienza por extraer puntos clave o “landmarks” de la imagen. Para realizar este primer paso, el programa hace uso de la utilidad `face_recognition` [11].

```

59     def mask(self):
60         import face_recognition
61
62         face_image_np = face_recognition.load_image_file(self.face_path)
63         face_locations = face_recognition.face_locations(face_image_np, model=self.model)
64         face_landmarks = face_recognition.face_landmarks(face_image_np, face_locations)
65         self._face_img = Image.fromarray(face_image_np)
66         self._mask_img = Image.open(self.mask_path)
67
68         found_face = False
69         for face_landmark in face_landmarks:
70             # check whether facial features meet requirement
71             skip = False
72             for facial_feature in self.KEY_FACIAL_FEATURES:
73                 if facial_feature not in face_landmark:
74                     skip = True
75                     break
76             if skip:
77                 continue
78
79             # mask face
80             found_face = True
81             self._mask_face(face_landmark)
82
83
84         if found face:

```

Figura 4.14. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada.

Tras ello, recorre las landmarks que ha encontrado buscando las que correspondan con las definidas como “face_landmarks”. En el caso de que se encuentre una que sea una cara, se emplea la función `_mask_face` de la clase para colocar una mascarilla correctamente alineada. Dicha función, calcula la posición en la que debería colocarse la mascarilla y la superpone a la imagen.

Por último, si se ha encontrado alguna cara se guardan las imágenes correspondientemente tratadas en el directorio de salida indicado, cuyo nombre será el numero identificativo del sujeto dado en el nombre de la imagen (<numerodeimagen>_<numerodesujeto>.jpg).

```

84     if found_face:
85         # align
86         src_faces = []
87         src_face_num = 0
88         with_mask_face = np.asarray(self._face_img)
89         for (i, rect) in enumerate(face_locations):
90             src_face_num = src_face_num + 1
91             (x, y, w, h) = rect_to_bbox(rect)
92             detect_face = with_mask_face[y:y + h, x:x + w]
93             src_faces.append(detect_face)
94
95         faces_aligned = face_alignment(src_faces)
96         face_num = 0
97         for faces in faces_aligned:
98             face_num = face_num + 1
99             faces = cv2.cvtColor(faces, cv2.COLOR_RGBA2BGR)
100             size = (int(128), int(128))
101             faces_after_resize = cv2.resize(faces, size, interpolation=cv2.INTER_LINEAR)
102             cv2.imwrite(self.save_path, faces_after_resize)
103
104     else:
105
106         print('Found no face.'+self.save_path)

```

Figura 4.15. Fragmento de código de la adaptación de Wear-Mask-To-Face empleada.

De esta manera, el algoritmo Wear-Mask-To-Face ahora nos produce un nuevo dataset a partir del dataset original, que llamaremos ORL_Masked. Se han juntado el dataset original, y el dataset ORL_Masked creado en un dataset llamado ORL_Combined que contiene tanto las imágenes con mascarilla como sin ella. Estos datasets han sido separados internamente en Train y Test, dos subconjuntos para entrenamiento y test respectivamente.

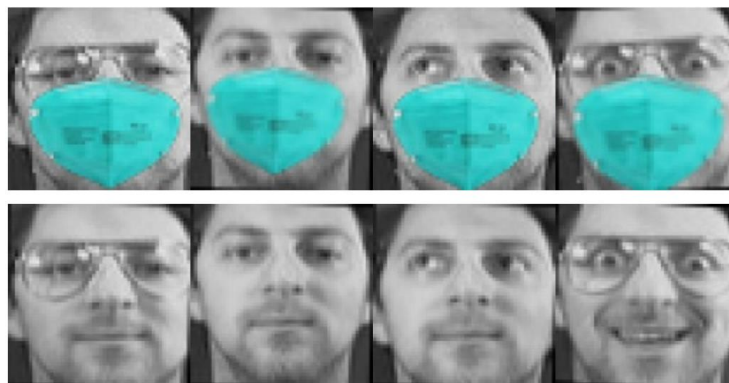
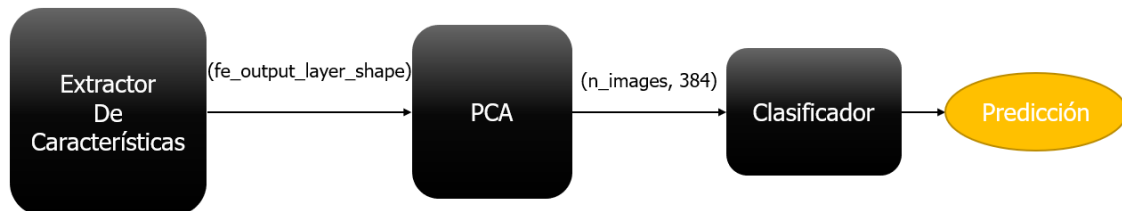


Figura 4.16. Ejemplo del dataset ORL_Combined creado a partir de Wear-Mask-To-Face

Este dataset consta aproximadamente de 560 imágenes para el conjunto de entrenamiento y 308 imágenes para el conjunto de test y se encuentra ya preparado para ser usado.

5. Diseño de modelos

En este proyecto se han implementado una serie de modelos basados cuya estructura tiene tres partes: un extractor de características, Principal Component Analysis como reductor de dimensionalidad, y un clasificador.



A continuación, se analizarán los diferentes modelos de redes neuronales empleados como extractores de características, el algoritmo Principal Component Analysis, así como los diferentes clasificadores empleados.

5.1. CNN Feature Extractor

Emplearemos una red neuronal convolucional (CNN) como extractor de características para nuestro sistema. En este proyecto, se han empleado modelos de redes ya existentes proporcionados por la biblioteca Keras, véase 4.1.2.

El diseño de estas redes neuronales se compone de dos partes principales, la extracción de características y su posterior clasificación en las últimas capas. Para nuestro propósito, emplearemos las redes únicamente como extractor de características, por lo que la sección de clasificación de la red no será utilizada.

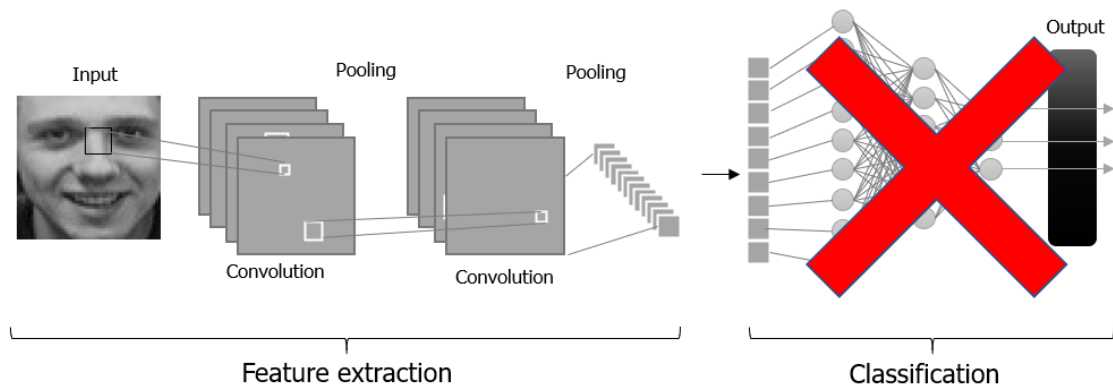


Figura 5.1. Esquema de funcionamiento de una CNN indicando la inutilización de su etapa de clasificación.

Esto es posible gracias a las herramientas que facilita Keras, que permite hacer uso de las redes sin incluir sus últimas capas de clasificación si lo indicamos al crear el modelo. La opción que permite este tipo de uso se denomina “include_top”, si se encuentra en falso no incluirá las capas finales del modelo CNN.

Estos modelos serán creados con pesos ya inicializados, ya que estos modelos de redes han sido pre-entrenados anteriormente. De esta manera no necesitamos entrenar cada una de estas redes desde cero, sino que cargamos los pesos al crear el modelo y las redes quedan listas para su uso. Además, se establecen las capas de la red como no-entrenables, dado que la intención es emplear la red como extractor de características de imágenes.

```
-----
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
-----
```

Figura 5.2. Ejemplo de parámetros entrenables/no-entrenables al crear una red VGG16

Para este proyecto se han escogido los pesos pre-entrenados con ImageNet. ImageNet es un dataset con millones de imágenes de entrenamiento [11] creado por un equipo formado por miembros de la Universidad de Stanford y miembros de la Universidad de Princeton.



Figura 5.3. Logotipo de ImageNet

Se han empleado varias redes CNN distintas para realizar, posteriormente, realizar una comparación sobre su efectividad a la hora de extraer características.

5.1.1. ResNet50

Uno de los modelos CNN empleados como feature extractor es el modelo ResNet50. Este modelo [13] está basado en bloques residuales, una estructura que permite transportar datos ya empleados a capas posteriores para su uso.

Al contrario que las redes clásicas, la función de error de este tipo de red no empeora al añadir una gran cantidad de capas, por lo que es posible crear modelos de este tipo de red con más de 150 capas.

Las capas de estas redes se pueden agrupar en bloques, estos bloques son los encargados de realizar las operaciones correspondientes a la extracción de características. La estructura básica de estos bloques puede ser, por ejemplo, varios sub-bloques concatenados compuestos por un bloque de operación linear, y un bloque de rectificación ReLU.

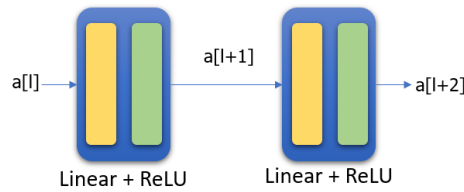


Figura 5.4. Esquema de un bloque de procesamiento interno de una CNN.

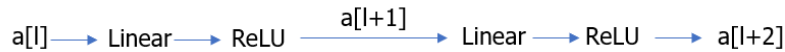


Figura 5.5. Esquema de obtención de $a[l+2]$ en CNN

De esta manera, se describe la fórmula matemática para obtener $a^{[l+2]}$:

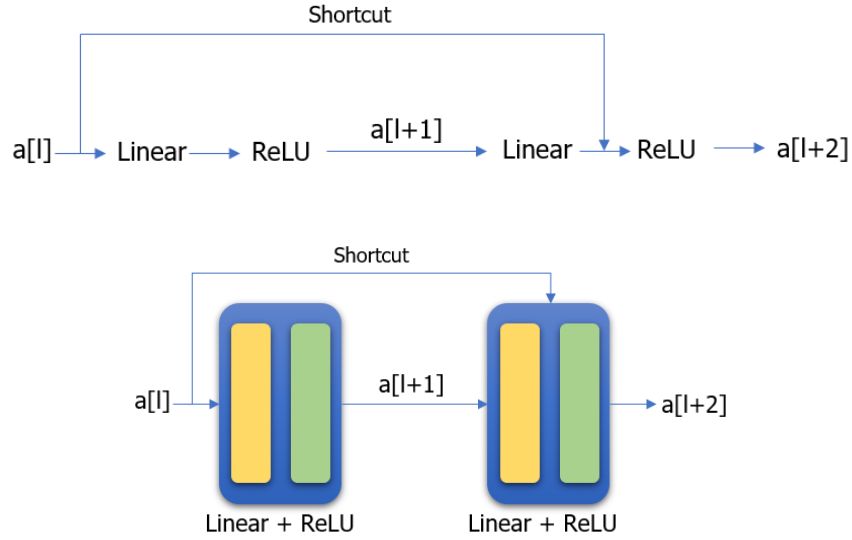
$$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}; \quad a^{[l+1]} = g(z^{[l+1]});$$

$$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]};$$

$$a^{[l+2]} = g(z^{[l+2]});$$

Donde b es el bias correspondiente, W es la matriz de pesos de la capa, a es la salida anterior que se emplea como entrada en la capa, z es la salida de la operación linear y g es la salida de la capa tras aplicar la función ReLU.

En el caso de las redes residuales, es en este procedimiento en el que se aplica un “shortcut” o “atajo” para recuperar el valor de $a^{[l]}$ y llevarlo hasta antes de la fase $\text{ReLU}^{[l+2]}$.



Figuras 5.6 y 5.7. Esquema de un bloque de procesamiento residual de una CNN ResNet

De esta manera, ahora la información de $a^{[l]}$ puede avanzar en la red, modificando la ecuación representativa del bloque de la siguiente manera:

$$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}; \quad a^{[l+1]} = g(z^{[l+1]});$$

$$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]};$$

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]});$$

Esto cambia la salida $a^{[l+2]}$ y forma lo que se denomina bloque residual.

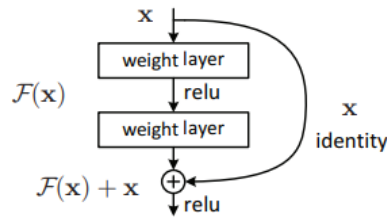


Figura 5.8. Extracto de [11], diagrama de un bloque residual de ResNet

Estos bloques residuales se pueden concatenar uno detrás de otro, formando redes residuales muy extensas.

En ResNet50, estos bloques son incluso más profundos, cuyo shortcut abarca 3 bloques computacionales en lugar de 2, véase Figura 5.9.

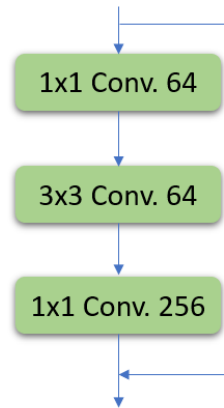


Figura 5.9. Esquema simplificado de un bloque correspondiente a la primera etapa de una red ResNet50.

Las redes ResNet están formadas por etapas que implementan este tipo de bloques con distintos parámetros.

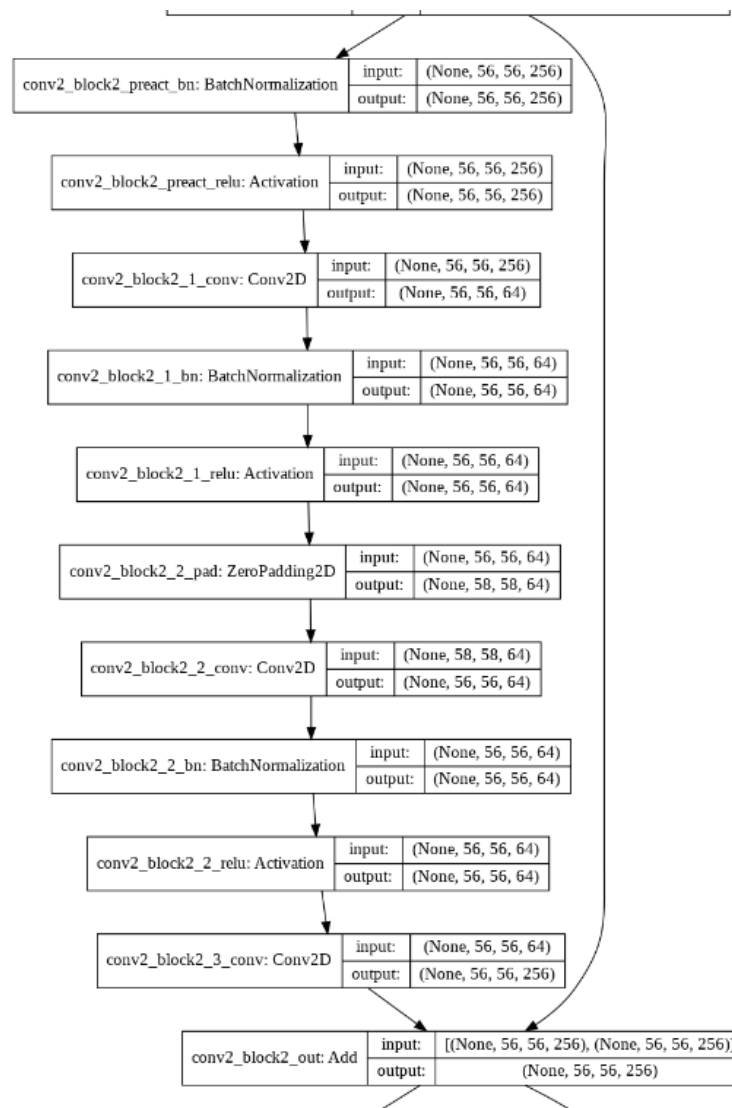


Figura 5.10. Fragmento del esquema de la red ResNet50 empleada

5.1.2. ResNet101

El modelo ResNet101 es una implementación de la red residual ResNet comentada en 5.1.1. Esta versión cuenta con 101 capas que siguen la misma filosofía de ResNet50.

Dado que la estructura de esta red es muy similar a la anteriormente comentada ResNet50 y la gran cantidad de capas ocuparía un espacio demasiado extenso, sólo se muestra un bloque residual del modelo ResNet101 empleado.

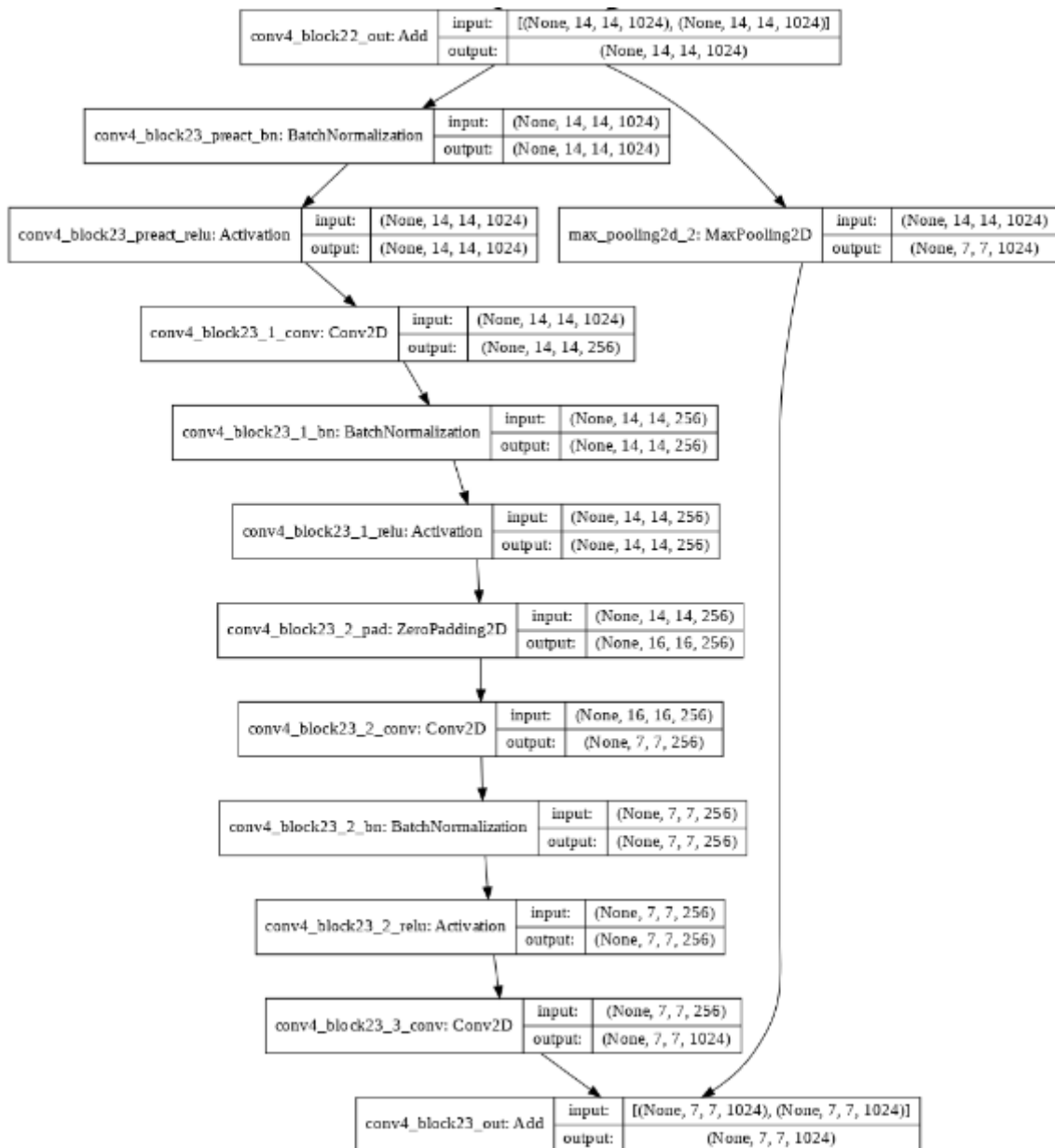


Figura 5.11. Fragmento del esquema de la red ResNet101 empleada

5.1.3. VGG-16

Las redes VGG-16, VGG-19 y sus modelos derivados fueron desarrolladas por el Grupo de Gráficos Visuales (VGG) de la Universidad de Oxford para competir en el Desafío ImageNet en 2014 [14]. Estas redes surgieron para competir con AlexNet, después de que esta red superara en rendimiento al anterior mejor modelo, LeNet.

El modelo VGG, en lugar de usar capas individuales como AlexNet, agrupa las capas en bloques, véase figura 5.12.

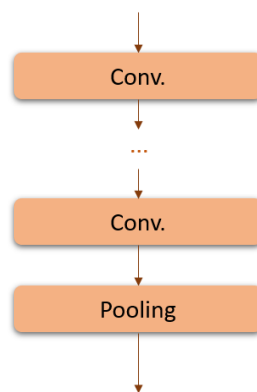


Figura 5.12. Esquema simplificado de un bloque de red VGG

El equipo de VGG hizo un análisis [14] sobre el tipo de capas a utilizar. Este análisis explicaba cómo se obtenían mejores resultados al emplear un número mayor de capas con una estructura convolucional más pequeña, frente a emplear menos capas con estructura convolucional más amplia. Esto se traduce en capas de convolución con un tamaño de 3x3, y capas de max-pooling 2x2.



Figura 5.13. Esquema simplificado de la red VGG16

Ahora bien, esta red está diseñada para clasificar, en este proyecto emplearemos la red VGG sin tener en cuenta las últimas capas de clasificación, por lo que el esquema empleado en realidad será algo parecido a la Figura 5.14.

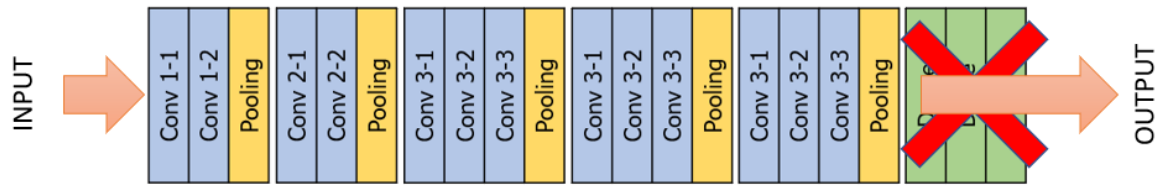
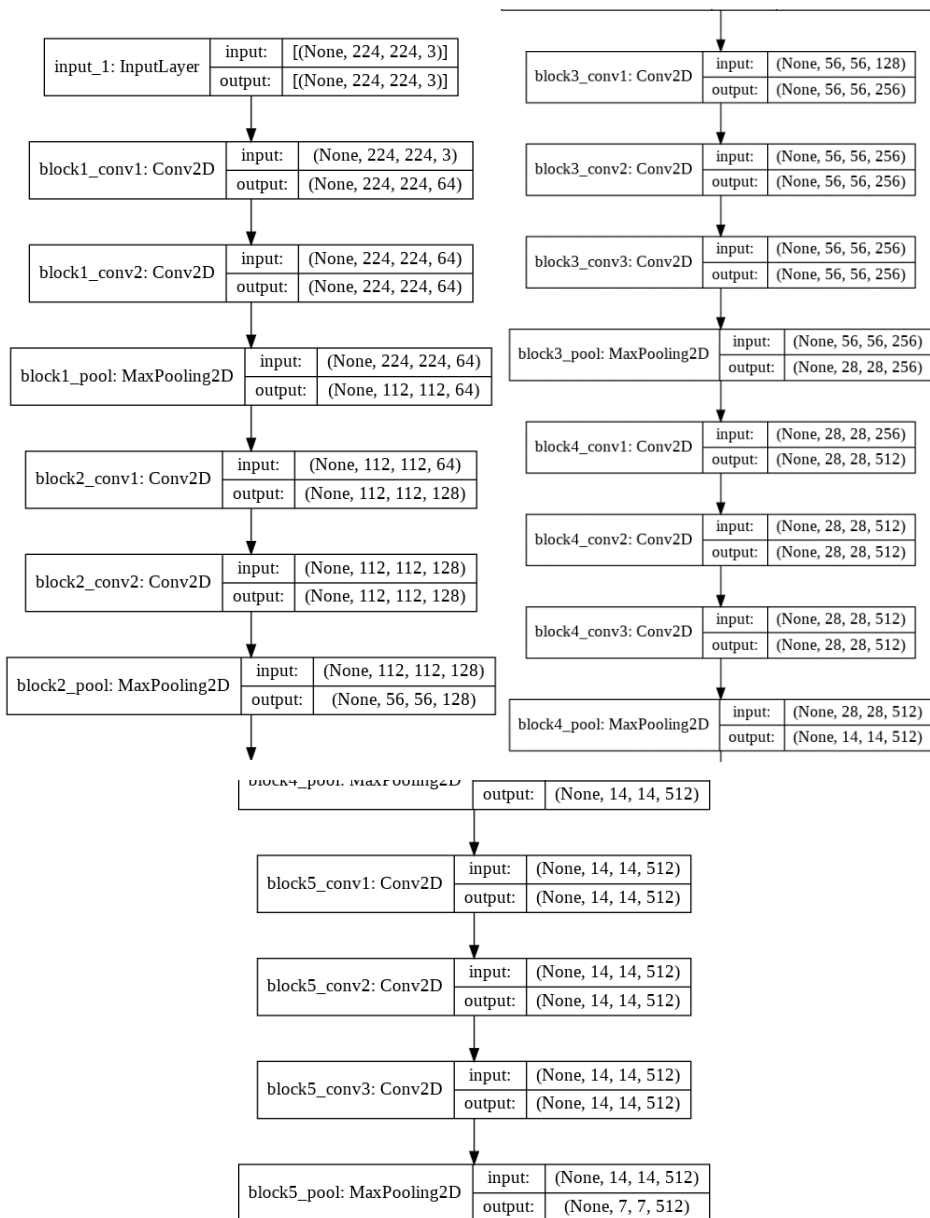


Figura 5.14. Esquema simplificado de la red VGG16 empleada

En el que no empleamos las capas de salida, sino que obtenemos la salida de la última capa de pooling para ser empleado como extractor de características.



Figuras 5.15, 5.16, 5.17. Estructura real de la red VGG16 empleada

En las figuras 5.15, 5.16, y 5.17, podemos observar la estructura de la red VGG16 implementada, donde la última capa nos devolverá un vector 7x7x512 por imagen.

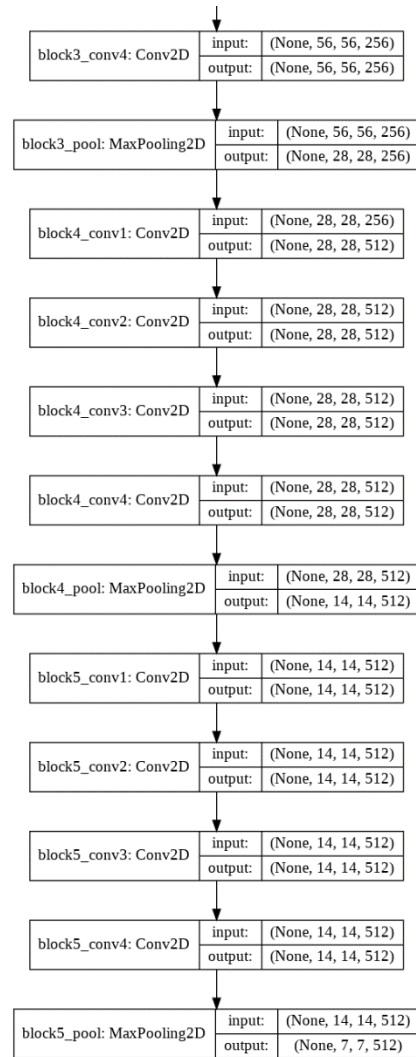
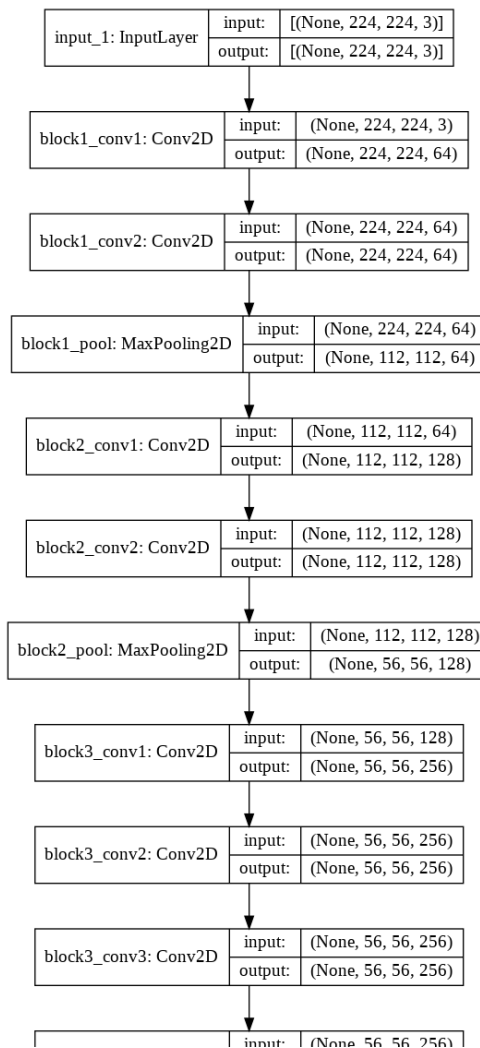
5.1.4. VGG-19

La red VGG-19 se basa en el mismo concepto que la red VGG-16, sin embargo, la estructura de la red VGG-19 contiene en las tres últimas capas una capa más de convolución. De manera que el esquema sería el siguiente:



Figura 5.18. Esquema simplificado de la red VGG19

Como se ha comentado anteriormente, no emplearemos las últimas capas para obtener la clasificación.



Figuras 5.19 y 5.20. Esquema real de la red VGG19 implementada.

5.1.5. Inception V3

El modelo Inception surgió de la necesidad de escoger unas características exactas para los filtros internos de las capas de las redes neuronales. Para solucionar esta indecisión, el equipo de Google que presentó Inception en su artículo [15] decidió implementar un modelo en el que no hubiera que tomar esa decisión.

La arquitectura de Inception realiza las opciones posibles de capas de convolución o capas de pooling en cada bloque, y concatena los resultados en una capa. De manera que es la red la que resultados y métodos son los óptimos. Un ejemplo de esta estructura es la figura 5.21.

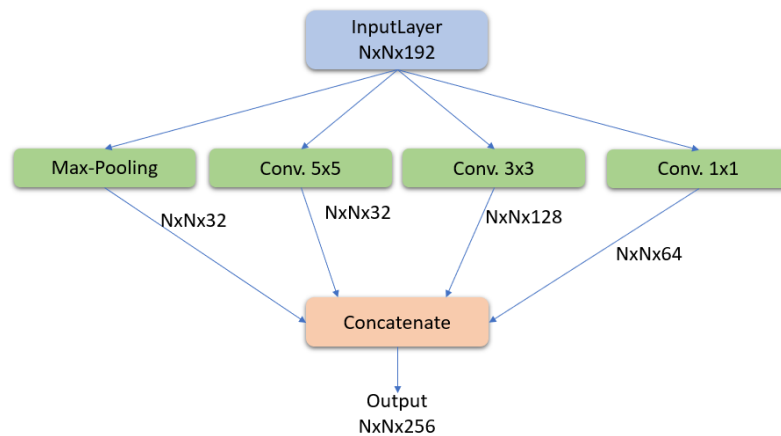


Figura 5.21. Ejemplo simplificado de estructura de un bloque de InceptionV3

A continuación, en la figura 5.22 podemos observar un bloque de la arquitectura Inception empleada en el proyecto.

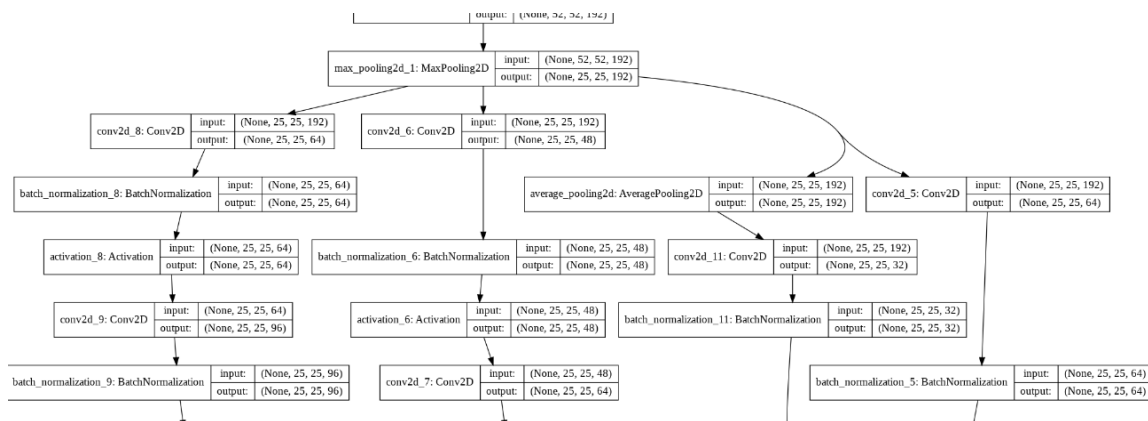


Figura 5.22. Mitad superior de un bloque de la red Inception usada en el proyecto

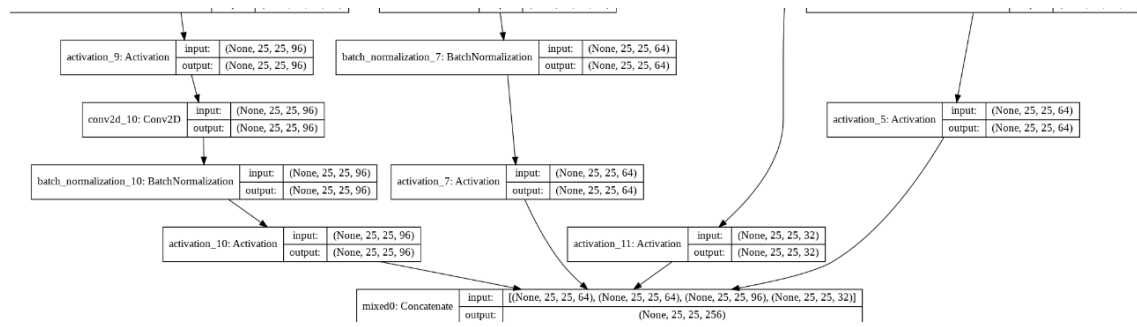


Figura 5.23. Mitad inferior de un bloque de la red Inception usada en el proyecto

5.2. PCA

El análisis de componentes principales (Principal Component Analysis por sus siglas en inglés) es una técnica ampliamente empleada en el campo de machine learning para reducir dimensionalidades y reorganizar los datos.

PCA consigue reorganizar los datos entorno a nuevos ejes, esto permite ver la información desde otra perspectiva que aporta más información.

En el proyecto se usará PCA para reducir dimensionalmente la salida de nuestro extractor de características hasta 384 componentes por cada imagen.

Este proceso será llevado a cabo por una clase importada de la biblioteca Scikit-learn, véase 4.1.5.

5.3. Clasificadores ML

Para realizar la clasificación en última instancia se han implementado diferentes versiones, con clasificadores distintos, esto permitirá hacer un pequeño análisis y comparativa de los resultados obtenidos según cada clasificador y cada extractor.

5.3.1. MLP. Perceptrón multicapa

El perceptrón multicapa es uno de los algoritmos de machine learning más antiguos. Es un tipo de red neuronal artificial simple, que consta tres partes: capa de entradas, capas ocultas y capa de salida.

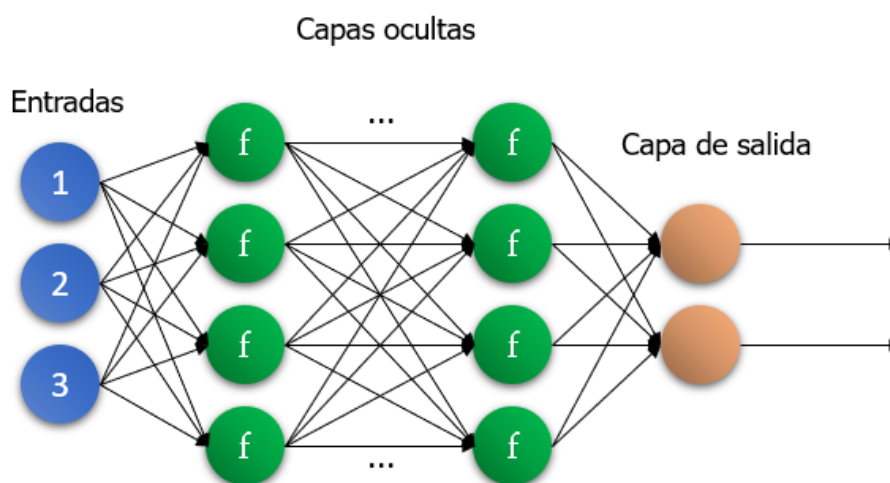


Figura 5.24. Esquema simplificado de un perceptrón multicapa.

Cada capa oculta está formada por múltiples perceptrones/neuronas que tienen una función de activación. En nuestra implementación, emplearemos el clasificador MLP que encontramos en la biblioteca Scikit-learn. Con función de activación por defecto ReLU.

El perceptrón diseñado constará de dos capas ocultas con 1536 y 768 neuronas respectivamente.

5.3.2. Random forest

Random forest es un método de aprendizaje ensemble orientado a clasificar datos en clases. Fueron presentados por Tin Kam Ho [16] en 1995.

Este tipo de clasificador está basado en la estructura de árboles. Tras ajustar el Random forest a nuestro espacio de entrenamiento se generan varios árboles de clasificación, formados por nodos de decisión y nodos hoja.

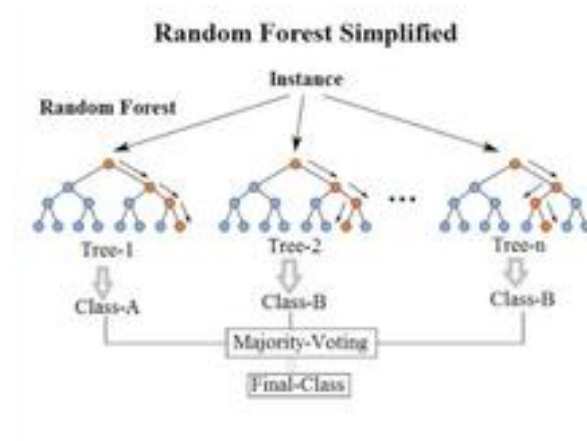


Figura 5.25. Esquema simplificado de un random forest genérico [18]

Para hacer una predicción, se calculan las predicciones de cada árbol para las características de entrada, y se combinan las predicciones mediante voto por mayoría. En el voto por mayoría, cada uno de los árboles internos realiza una predicción (voto) y la predicción final será la que reciba más de la mitad de dichos votos.

En nuestro proyecto, emplearemos el Random Forest Classifier [17] disponible en la librería Scikit-learn, véase 4.1.5.

5.3.3. Support Vector Machine

Desarrollados por Vladimir Vapnik y su equipo en AT&T Bell Laboratories, 1995, los Support Vector Machine son uno modelos de aprendizaje supervisado ampliamente usados junto a algoritmos de clasificación. Este tipo de clasificador permite separar el espacio de datos entre las clases existentes.

Los SVM requieren escoger un kernel determinado para su correcto funcionamiento según los datos a los que se van a aplicar. En el proyecto se ha empleado el clasificador LinearSVC de la biblioteca Scikit-learn para clasificación multiclase, véase 4.1.5.

6. Implementación

El programa desarrollado se divide en 4 partes principales:

- Preparación de los datos
- Preparación de la red
- Preparación del clasificador
- Ejecución de las pruebas y obtención de métricas

A continuación, se explicará cada una de dichas partes, así como otros métodos auxiliares implementados.

6.1. Preparación de los datos

El programa comienza extrayendo el dataset, haciendo uso de la biblioteca zipfile. Debido a que nuestro dataset pasó por la utilidad Wear-Mask-To-Face, véase 4.3.1, y Dataset-Cleaner (una función derivada de Wear-Mask-To-Face que no superpone mascarilla, sino que simplemente detecta las caras y las delimita) nuestro dataset está listo para ser usado por el extractor de características.

```
1
2
3 drive.mount('/content/drive', force_remount=True)
4 pathToUnZip = './drive/MyDrive/TFG/Datasets/URL_Combined_Ready'
5 pathToZip1 = './drive/MyDrive/TFG/Datasets_zips/URL_Combined_Ready.zip'
6 train_Directory = './drive/MyDrive/TFG/Datasets/URL_Combined_Ready/URL_Combined'
7 test_Directory = './drive/MyDrive/TFG/Datasets/URL_Combined_Ready/URL_Combined_1'
8
9
10 with zipfile.ZipFile(pathToZip1, 'r') as zip_ref:
11     print("Extracting into"+ train_Directory)
12     zip_ref.extractall(pathToUnZip)
13 print('Dataset extracted to: '+train_Directory)
14
```

Figura 6.1. Extracto de código correspondiente a la extracción del dataset

Sin embargo, antes de pasar al apartado de la red neuronal, debemos balancear el espacio de datos ya que disponemos de aproximadamente 300 imágenes de prueba y 560 imágenes de entrenamiento. En cuanto al balance de imágenes con y sin mascarilla, el cincuenta por ciento de las imágenes corresponden a imágenes sin mascarilla y el otro cincuenta por ciento a imágenes con mascarilla.

6.1.1. Data augmentation

Realizaremos el aumentado de datos utilizando ImageDataGenerator de la biblioteca Keras, para ello se especificará a la hora de crear el IDG los parámetros específicos para el aumentado de datos. En este caso se ha empleado el parámetro rango de brillo (0.1, 0.2) y un parámetro de rotación (-30, 30).

```
3 dataAugmentTrain = 2
4 dataAugmentTest = 1
5
6 IDG_train = ImageDataGenerator(rotation_range = 10,brightness_range= (0.1, 0.2))
7 IDG_test = ImageDataGenerator(rotation_range = 10,brightness_range= (0.1, 0.2))
8 train_generator = IDG_train.flow_from_directory(
9     directory = train_Directory, class_mode = 'categorical', batch_size = ntoTrain,
10    target_size=imgDims, color_mode='rgb', shuffle = True
11 )
12
13 test_generator = IDG_test.flow_from_directory(
14    directory = test_Directory, class_mode = 'categorical', batch_size = ntoTest,
15    target_size=imgDims, color_mode='rgb', shuffle = True
16 )
```

Figura 6.2. Extracto de código correspondiente al uso de objetos IDG

Posteriormente, extraeremos los datos de los IDG correspondientes empleando los generadores obtenidos a partir del método Flow_from_directory. Se establece en las variables dataAugmentTrain y dataAugmentTest las veces que queremos aumentar el dataset, en este caso se han establecido para que el dataset de entrenamiento sea 3 veces mayor que los datos de prueba. De manera que nos quedamos con aproximadamente 1600 imágenes de entrenamiento y 600 imágenes de prueba.

```

18 # OBTENEMOS LOS arrays DE IMAGENES-LABELS
19
20 x_train,dummy_vector = train_generator.next()
21 y_train = extract_y(dummy_vector)
22 x_test,dummy_vector = test_generator.next()
23 y_test = extract_y(dummy_vector)
24
25 x = dataAugmentTrain
26 while x > 0:
27     x_train2,dummy_vector = train_generator.next()
28     y_train2 = extract_y(dummy_vector)
29     x_train = np.append(x_train, x_train2, axis=0)
30     y_train = np.append(y_train, y_train2, axis=0)
31     x -= 1
32
33 x = dataAugmentTest
34 while x > 0:
35
36     x_test2,dummy_vector = test_generator.next()
37     y_test2 = extract_y(dummy_vector)
38     x_test = np.append(x_test, x_test2, axis=0)
39     y_test = np.append(y_test, y_test2, axis=0)
40     x -= 1
41

```

Figura 6.3. Extracto de código correspondiente a la creación de los conjuntos de datos

El código en la figura 6.3 es un código muy rudimentario para extraer las imágenes y sus correspondientes etiquetas del generador creado anteriormente.

6.2. Preparación de CNN

Tras esto, se prepara la red neuronal que será nuestro extractor de características. Este paso es similar para todos los modelos empleados, ya que al emplear los modelos facilitados por la biblioteca Keras, véase 4.1.2, simplemente tenemos que recuperar el modelo indicando el set de pesos que queremos cargar.

Construiremos las redes a partir de los pesos de ImageNet, con un `input_shape` correspondiente al de las imágenes de nuestro dataset tratado (224,224,3) y estableciendo el atributo `“include_top = false”`.

Cada uno de los siguientes fragmentos de código está ubicado en su propio cuaderno Colaboratory, de manera que para cada tipo de red existe un cuaderno con cada tipo de clasificador.

6.2.1. Resnet50

```

1 # RESNET50
2 def getRESNET50Model():
3     rn50model = keras.applications.ResNet50V2(weights='imagenet', input_shape=input_shape, include_top=False)
4
5     # Establecemos las capas interiores como no-entrenables
6     for layer in rn50model.layers[:]:
7         layer.trainable = False
8
9     rn50model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
10    rn50model.summary()
11
12    return rn50model
13
14 cnnModel = getRESNET50Model()

```

Figura 6.4. Extracto de código correspondiente a la creación de la CNN ResNet50

6.2.2. Resnet101

```

1 # RESNET101v2
2 def getRESNET101Model():
3     rn101model = keras.applications.ResNet101V2(weights='imagenet', input_shape=input_shape, include_top=False)
4
5     # Establecemos las capas interiores como no-entrenables
6     for layer in rn101model.layers[:]:
7         layer.trainable = False
8
9     rn101model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
10    rn101model.summary()
11
12    return rn101model
13
14 cnnModel = getRESNET101Model()
15

```

Figura 6.5. Extracto de código correspondiente a la creación de la CNN ResNet101

6.2.3. VGG16

```

1 # VGG16 para Feature Extraction
2 def getVGG16Model(lastFourTrainable=False):
3     vgg_model = VGG16(weights='imagenet', input_shape=input_shape, include_top=False)
4
5     # Establecemos las capas interiores como no-entrenables
6     for layer in vgg_model.layers[:]:
7         layer.trainable = False
8
9     vgg_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
10    vgg_model.summary()
11
12    return vgg_model
13
14 cnnModel = getVGG16Model(lastFourTrainable=True)

```

Figura 6.6. Extracto de código correspondiente a la creación de la CNN VGG16

6.2.4. VGG19

```

1 # VGG-19
2 def getVGG19Model():
3     VGG19model = keras.applications.VGG19(weights='imagenet', input_shape=input_shape, include_top=False)
4
5     # Establecemos las capas interiores como no-entrenables
6     for layer in VGG19model.layers[:]:
7         layer.trainable = False
8
9     VGG19model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
10    VGG19model.summary()
11
12    return VGG19model
13
14 cnnModel = getVGG19Model()
15

```

Figura 6.7. Extracto de código correspondiente a la creación de la CNN VGG16

6.2.5. InceptionV3

```

1 # Inception
2 def getInceptionModel():
3     inceptionModel = keras.applications.InceptionV3(weights='imagenet', input_shape=input_shape, include_top=False)
4
5     # Establecemos las capas interiores como no-entrenables
6     for layer in inceptionModel.layers[:]:
7         layer.trainable = False
8
9     inceptionModel.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
10    inceptionModel.summary()
11
12    return inceptionModel
13
14 cnnModel = getInceptionModel()

```

Figura 6.8. Extracto de código correspondiente a la creación de la CNN InceptionV3

6.3. Preparación del clasificador

Empleamos el modelo que acabamos de crear para obtener las características de nuestro set de entrenamiento y usaremos el objeto PCA para realizar la reducción de dimensionalidad.

```

1 x_train_fExtracted = cnnModel.predict(x_train )

1 pca_components = 384 ##### 128*3
2 X_train_PCA = x_train_fExtracted.reshape(x_train_fExtracted.shape[0], -1)
3 pca = PCA(n_components = pca_components, whiten = True).fit(X_train_PCA)
4 X_for_ML = pca.transform(X_train_PCA)

```

Figura 6.9. Extracto de código correspondiente al uso del objeto PCA

Tras esto, se crean los clasificadores y comienza su entrenamiento con las características obtenidas. Como se ha comentado en 6.2., para cada CNN se dispone de un cuaderno de Colab con cada clasificador implementado.

6.3.1. MLP

El entrenamiento del MLP implementado se basa en un bucle que repetirá la creación y entrenamiento de MLP hasta que haya creado 10, o hasta que obtenga una precisión de 0.97. Si el MLP consigue un desempeño mejor que el mejor hasta ahora, se tomará como nuevo mejor.

```

1 x_test_fExtracted = cnnModel.predict(x_test)
2 x_test_for_ML = pca.transform(x_test_fExtracted.reshape(x_test_fExtracted.shape[0], -1))
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

```

```

1
2 from sklearn.neural_network import MLPClassifier
3
4 print("Fitting the classifier to the training set")
5 x = 0
6 lacc = 0
7 #dummy classifier
8 savedClass = MLPClassifier(hidden_layer_sizes=(1024,), batch_size = 28, early_stopping = True, n_iter_no_
9 niter = 10
10 nbatch = 48
11 while x <= 10 and lacc < 0.97:
12
13     classifier = MLPClassifier(hidden_layer_sizes=(1536,768,), batch_size = nbatch, early_stopping = True,
14
15     y_pred = classifier.predict(x_test_for_ML)
16     acc = sklearn.metrics.precision_score(y_true = y_test,y_pred = y_pred, average='macro', zero_division=0
17     print("Acc for "+ str(x) + " : " + str(acc))
18
19     if acc > lacc:
20         print("At "+ str(x) +" precision: " + str(acc)+ "\tBetter than "+str(lacc) )
21         savedClass = classifier
22         lacc = acc
23     else:
24         niter += 1
25
26     x += 1
27
28 clf = savedClass
29

```

Figura 6.10. Extracto de código correspondiente a la creación y entrenamiento de MLP

6.3.2. Random Forest

El clasificador random forest se crea empleando el objeto de la biblioteca Scikit-learn, véase 4.1.5. Posteriormente se entrena y se obtienen las predicciones realizadas para el conjunto de pruebas.

```

1
2 from sklearn.ensemble import RandomForestClassifier as RFC
3
4 RF_model = RFC(n_estimators = 512, random_state = 42)
5 RF_model.fit(X_for_ML, y_train)
6
7

```

Figura 6.11. Extracto de código correspondiente a la creación y entrenamiento de RF

```

1 x_test_fExtracted = modelVGG16_notTrainable.predict(x_test)
2 x_test_for_ML = pca.transform(x_test_fExtracted.reshape(x_test_fExtracted.shape[0], -1))
3 pred_RF = RF_model.predict(x_test_for_ML)

```

Figura 6.12. Extracto de código correspondiente a la predicción del conjunto de pruebas de RF

6.3.3. LinearSVC

En cuanto a la implementación de SVM, o LinearSVC en este caso, se ha empleado el objeto de Scikit-learn y su proceso es similar al de 6.3.2.

```

2 from sklearn import svm
3
4 clf = svm.LinearSVC()
5 clf.fit(X_for_ML, y_train)
6

```

Figura 6.13. Extracto de código correspondiente a la creación y entrenamiento de SVM
(LinearSVC)

```

1 x_test_fExtracted = cnnModel.predict(x_test)
2 x_test_for_ML = pca.transform(x_test_fExtracted.reshape(x_test_fExtracted.shape[0], -1))
3 pred_RF = clf.predict(x_test_for_ML)

```

Figura 6.14. Extracto de código correspondiente a la predicción del conjunto de pruebas de RF

6.4. Extracción de resultados

Independientemente del modelo, tras obtener los resultados se realiza una fase de extracción de métricas utilizando la clase metrics de la biblioteca Scikit-learn.

```

1
2 print ("Accuracy = ", metrics.accuracy_score(y_test, pred_RF))
3 print ("Classification Report:\n")
4 print (metrics.classification_report( y_test, pred_RF, zero_division = 0 ))
5

```

Figura 6.15. Extracto de código correspondiente a la extracción de métricas

Este método da como resultado una tabla con los resultados de la clasificación divididos por clases, véase 6.16. Además, nos permite extraer métricas como el accuracy global o la precisión media global, que analizaremos en 7.

```
Accuracy = 0.9495114006514658
Classification Report:
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	14
1	0.94	1.00	0.97	16
2	1.00	0.88	0.93	16
3	0.92	0.75	0.83	16
4	1.00	0.88	0.93	16
5	0.94	1.00	0.97	16
6	0.80	1.00	0.89	16
7	0.94	0.88	0.91	16
8	0.94	0.88	0.91	16
9	0.94	0.88	0.91	16
10	0.94	0.88	0.91	16
11	0.94	0.88	0.91	16
12	0.94	0.88	0.91	16
13	0.94	0.88	0.91	16
14	0.94	0.88	0.91	16
15	0.94	0.88	0.91	16
16	0.94	0.88	0.91	16
17	0.94	0.88	0.91	16
18	0.94	0.88	0.91	16
19	0.94	0.88	0.91	16
20	0.94	0.88	0.91	16
21	0.94	0.88	0.91	16
22	0.94	0.88	0.91	16
23	0.94	0.88	0.91	16
24	0.94	0.88	0.91	16
25	0.94	0.88	0.91	16
26	0.94	0.88	0.91	16
27	0.94	0.88	0.91	16
28	0.94	0.88	0.91	16
29	0.94	0.88	0.91	16
30	0.94	0.88	0.91	16
31	0.94	0.88	0.91	16
32	0.94	0.88	0.91	16
33	0.94	0.88	0.91	16
34	0.94	0.88	0.91	16
35	0.94	0.88	0.91	16
36	0.94	0.88	0.91	16
37	1.00	1.00	1.00	16
38	0.94	1.00	0.97	16
39	1.00	0.62	0.77	16
accuracy			0.95	614
macro avg	0.95	0.95	0.95	614
weighted avg	0.96	0.95	0.95	614

Figuras 6.16, 6.17. Tabla de ejemplo obtenida al emplear los métodos de la figura 6.15

Posteriormente, se extrae una imagen al azar del conjunto de imágenes de prueba, y se ejecuta una predicción. Esto es a modo de demostración.

```
1 def getPred(model, img):
2     featureVector = cnnModel.predict(x_test[arn].reshape(1,224,224,3))
3
4     return model.predict(pca.transform(featureVector.reshape(featureVector.shape[0], -1)))
5
6 ## Seleccionamos un elemento aleatorio de test
7
8 import random
9 arn = random.randint(0, len(x_test))
10
11 print ("Prediction for x_test["+str(arn)+"] from "+ str(y_test[arn]) )
12
13 pred = getPred(clf, x_test[arn])
14 print("Prediction from RF: " + str(pred))

>prediction for x_test[148] from 31
>prediction from RF: [31]
```

Figura 6.18. Fragmento de código donde se prueba una imagen al azar del conjunto de pruebas

Hay que tener en cuenta, que, aunque los nombres de los directorios son [1...40], en el programa las clases van en un rango [0...39], por lo que en este ejemplo estaríamos prediciendo una imagen de la clase 32.

De igual modo, en la figura 6.19 se muestra un bloque de código donde se prueba una imagen en concreto preseleccionada.

En 6.18, se define la ubicación de la imagen, en este caso una imagen de un individuo a la cual se le ha superpuesto una mascarilla.

Tras esto, se define un método para obtener una predicción dado un clasificador y una ruta a una imagen, el código lee la imagen desde la ruta, y la prepara para el clasificador.

Por último, la carga en el clasificador y devuelve el resultado de la predicción. También se muestra por pantalla la imagen que con la que se ha trabajado y su predicción.

```

1 # Probamos una imagen en concreto
2 pathToImg = "./drive/MyDrive/TFG/Datasets/URL_DB_Masked/00021/201.jpg"
3
4 def getClfPred(classifier, img_path):
5     img = cv2.imread(img_path)
6     img = cv2.resize(img, (imgL, imgL))
7     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
8     kernel = np.ones((5,5),np.float32)/25
9     img = cv2.filter2D(img,-1,kernel)
10    plt.imshow(img)
11
12    featureVector = cnnModel.predict(img.reshape(1, 224, 224, 3))
13
14    return classifier.predict(pca.transform(featureVector.reshape(featureVector.shape[0]
15 clfpred = getClfPred(clf, pathToImg)
16
17 print(clfpred)

```

[20]

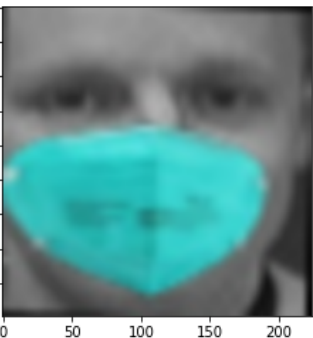


Figura 6.19. Fragmento de código donde se prueba una imagen en concreto

En este caso, como se puede ver en la figura 6.1, se estaba prediciendo una imagen proveniente del fichero 21. En el código la carpeta 21 corresponde a la clase 20 por el mismo motivo que se ha comentado anteriormente en este mismo apartado.

Todo el código de extracción de resultados comentado hasta ahora se ejecuta para cada uno de los modelos implementados en sus respectivos cuadernos de Google Colaboratory.

7. Resultados

A continuación, estudiaremos los resultados obtenidos con los distintos clasificadores y extractores de características.

7.1. MLP

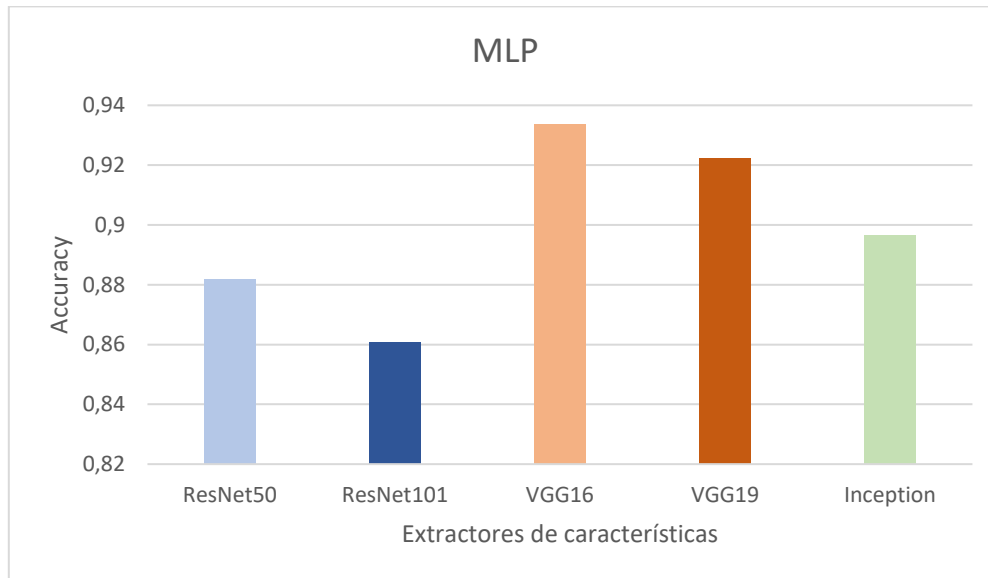


Figura 7.1. Gráfico comparativo de precisión con clasificador MLP

En cuanto a los resultados con el clasificador MLP se hace notar el buen desempeño del feature extractor realizado a partir de VGG16. En ese caso se obtiene una precisión en el conjunto de pruebas de 0.9336, es decir, un 93,36% de acierto.

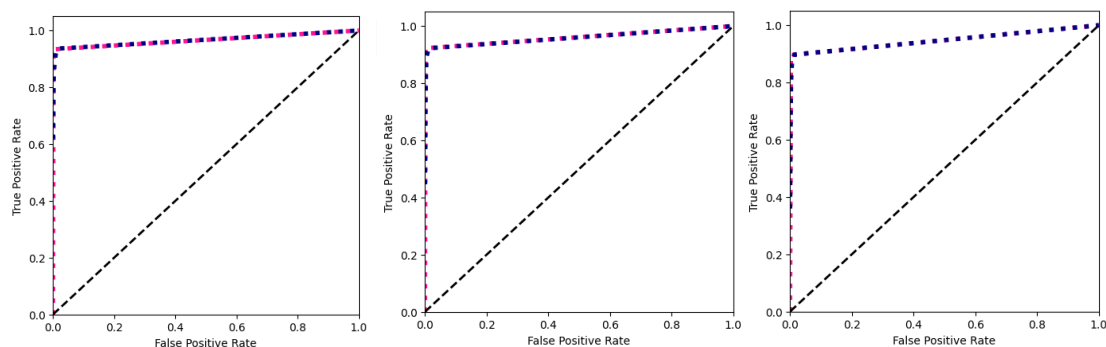


Figura 7.2. Curvas ROC correspondientes a los 3 mejores algoritmos empleando MLP. De izquierda a derecha: VGG16, VGG19, Inception.

En cuanto a la curva ROC, observamos que en los algoritmos que emplean VGG16, VGG19 e Inception junto a MLP el área bajo la curva es de 0.97, 0.96 y 0.95 respectivamente.

7.2. Random forest

En el caso de los resultados obtenidos empleando random forest, es destacable el mal rendimiento que se obtiene si lo comparamos con las métricas obtenidas en el caso del MLP o LinearSVC.

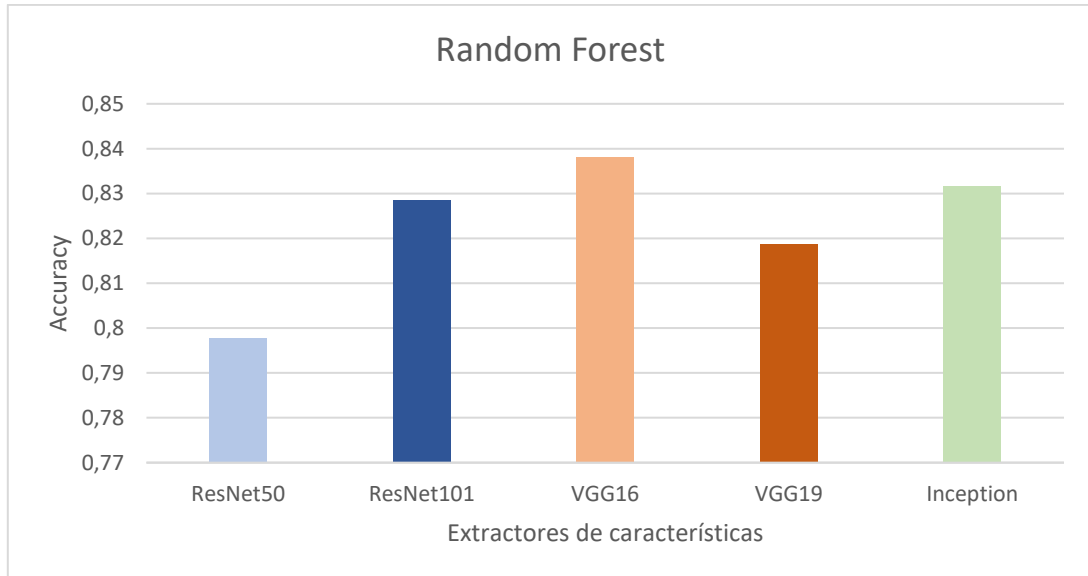


Figura 7.3. Gráfico comparativo de precisión con clasificador Random Forest

Mientras que, en el caso anterior empleando MLP, el mejor modelo obtenía una precisión de aproximadamente 93%, ahora con random forest, el mejor modelo usando el extractor VGG16 sólo logra un 83.8% de aciertos.

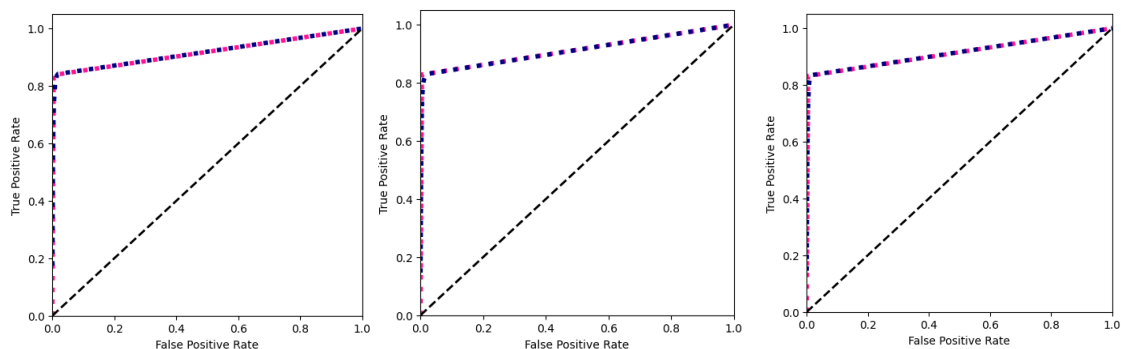


Figura 7.4. Curvas ROC correspondientes a los 3 mejores algoritmos empleando Random Forest.

De izquierda a derecha: VGG16, ResNet101, Inception.

Respecto a las curvas ROC de los algoritmos con Random Forest destaca la similitud de las áreas bajo la curva. En la figura 7.4 encontramos las curvas ROC de VGG16, ResNet101 e Inception empleando RF, con un área bajo la curva de 0.92, 0.91 y 0.91 respectivamente.

7.3. SVM. LinearSVC

Por último, cabe destacar los buenos resultados obtenidos al emplear como clasificador el objeto LinearSVC de Scikit-learn, muy similares a los obtenidos con MLP.

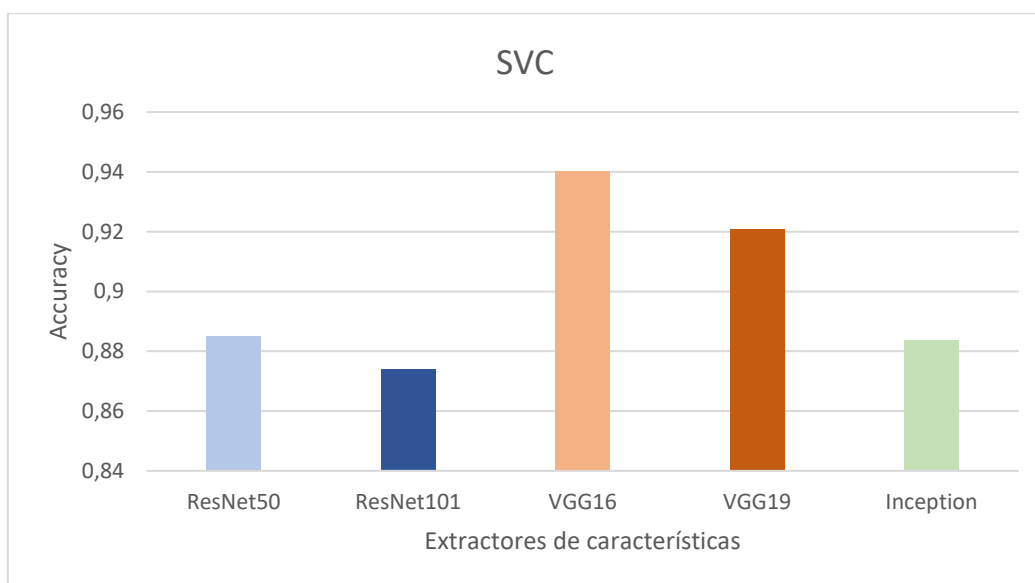


Figura 7.5. Gráfico comparativo de precisión con clasificador SVC

En esta ocasión, el clasificador empleando como feature extractor VGG16 vuelve a dominar con un 94% de accuracy. A VGG16 le sigue muy de cerca VGG19, que mejora los resultados obtenidos anteriormente en RF llegando a un 92% de accuracy.

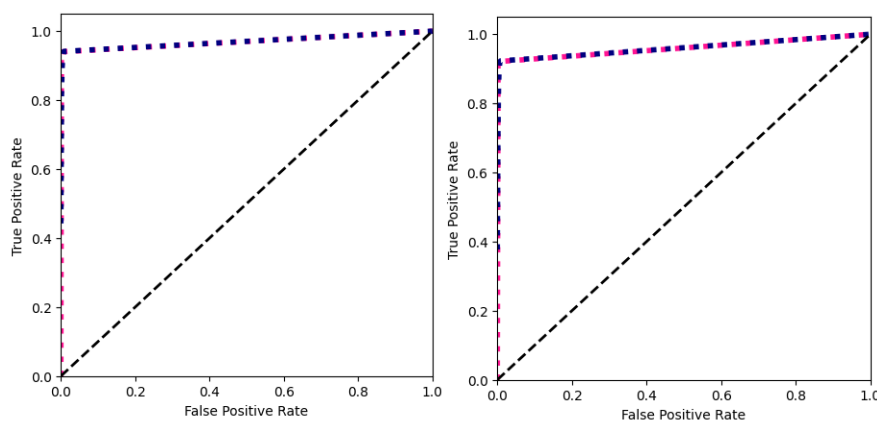


Figura 7.4. Curvas ROC correspondientes a los 2 mejores algoritmos a partir de LinearSVC. De izquierda a derecha: VGG16, VGG19

En esta última implementación, los mejores resultados son los obtenidos mediante el algoritmo que emplea VGG16 y VGG19, con un área bajo la curva de 0.97 y 0.96 respectivamente, así como las mejores precisiones generales.

7.4. Otros proyectos

En este apartado compararemos los resultados obtenidos de nuestra implementación con otros proyectos similares que emplean el dataset ORL.

Uno de estos proyectos es [21], que emplea una red CNN en Keras para realizar reconocimiento facial sobre el dataset ORL. En [21] se obtiene un modelo con 95% de precisión. Nuestro mejor modelo en comparación es el implementado usando VGG16 junto con LinearSVC, que obtiene una precisión de aproximadamente un 94%. Hay que tener en cuenta, que la tarea del proyecto [21] es reconocimiento facial, mientras que en este proyecto el objetivo es conseguir aplicar reconocimiento facial a imágenes con mascarilla.

Otro proyecto que emplea el dataset ORL es [6]. Como se ha comentado en el apartado 3 de este documento, este estudio aplica principal component analysis para conseguir aplicar reconocimiento facial con mascarilla sobre imágenes del dataset ORL y otras imágenes tomadas por el equipo del proyecto. Los resultados de dicho proyecto en cuanto a reconocimiento facial estándar oscilan entre 95% y 96% de precisión, mientras que en reconocimiento facial con mascarilla sus resultados oscilan entre 73% y 68%. En comparación, nuestro mejor modelo que emplea VGG16 como extractor de características, PCA para reducción de dimensionalidad y LinearSVC como clasificador, obtiene una precisión de 94% aproximadamente.

Por último, cabe destacar el rendimiento del algoritmo desarrollado por el equipo del proyecto RWMFD [9], anteriormente comentado en este documento, véase apartado 3. Además de la creación del dataset, implementaron un algoritmo dedicado a reconocimiento facial con mascarilla en tiempo real, dicho algoritmo está basado en discriminación de características por multigranularidad. El proyecto del RWMFD emplea el dataset con dicho nombre, y consigue unos resultados de precisión de hasta un 95%.



Figura 7.5. Ejemplo de reconocimiento facial con mascarilla en el proyecto RWFMD [9]

8. Conclusión

El reconocimiento facial es uno de los campos dentro del deep learning que más uso hace de las nuevas técnicas, modelos y algoritmos emergentes. Prueba de ello es la continua aparición de nuevos artículos y proyectos sobre la materia, tales como [5] o [9].

En este proyecto se ha intentado desarrollar un sistema capaz de reconocer caras cuyos rasgos faciales son obstruidos por el uso de la mascarilla. Tras analizar los resultados se puede decir que se ha logrado crear un algoritmo que, aunque no está al nivel del estado del arte, logra obtener buenas métricas en las pruebas realizadas.

A pesar de no haber conseguido un algoritmo al nivel de Google FaceNet, o el algoritmo descrito en [5], que hace uso de una estructura similar a la que se ha intentado implementar, pero mucho mejor optimizada, se ha hecho un estudio profundo del estado de la materia. En este estudio se ha podido ver la complejidad de este problema, que no solo se ve afectado por el uso de mascarilla, sino también por la capacidad de los diseñadores de modelos a la hora de describir e implementar nuevas técnicas para sobreponerse a estos impedimentos.

En cuanto a los resultados obtenidos, podemos decir que los mejores algoritmos son los creados a partir de multilayer perceptron y LinearSVC. Entre estos los que más destacan son los que emplean las redes VGG como extractor de características, VGG16 y VGG19. Dichos algoritmos obtienen puntuaciones de precisión y área bajo la curva ROC más que aceptables.

El reconocimiento facial con uso de mascarilla es un campo en auge, ya que debido al establecimiento de la mascarilla como un accesorio más, serán necesarias nuevas y mejores técnicas para poder realizar dicha tarea a tiempo real y de manera precisa.

9. Bibliografía y referencias

- [1] M. A. TURK AND A. P. PENTLAND, "FACE RECOGNITION USING EIGENFACES," PROCEEDINGS. 1991 IEEE COMPUTER SOCIETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 1991, PP. 586-591, DOI: 10.1109/cvpr.1991.139758. (20/05/2021)
- [2] F. Z. CHELALI, A. DJERADI AND R. DJERADI, "LINEAR DISCRIMINANT ANALYSIS FOR FACE RECOGNITION," 2009 INTERNATIONAL CONFERENCE ON MULTIMEDIA COMPUTING AND SYSTEMS, 2009, PP. 1-10, DOI: 10.1109/mmcs.2009.5256630. (20/05/2021)
- [3] STAROVOITOV, VALERY & SAMAL, DMITRY. (1999). A GEOMETRIC APPROACH TO FACE RECOGNITION.. 210-213. 10.13140/2.1.3200.8644. (20/05/2021)
- [4] F. SCHROFF, D. KALENICHENKO AND J. PHILBIN, "FACENET: A UNIFIED EMBEDDING FOR FACE RECOGNITION AND CLUSTERING," 2015 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), 2015, PP. 815-823, DOI: 10.1109/cvpr.2015.7298682. (20/05/2021)
- [5] CHEN, HONGLING & HAoyu, CHEN. (2019). FACE RECOGNITION ALGORITHM BASED ON VGG NETWORK MODEL AND SVM. JOURNAL OF PHYSICS: CONFERENCE SERIES. 1229. 012015. 10.1088/1742-6596/1229/1/012015. (20/05/2021)
- [6] EJAZ, MD & ISLAM, MD & SIFATULLAH, MD & SARKER, ANANYA. (2019). IMPLEMENTATION OF PRINCIPAL COMPONENT ANALYSIS ON MASKED AND NON-MASKED FACE RECOGNITION. 1-5. 10.1109/ICASERT.2019.8934543. (21/05/2021)
- [7] ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G.S.; DAVIS, A.; DEAN, J.; DEVIN, M.; ET AL. TENSORFLOW: LARGE-SCALE MACHINE LEARNING ON HETEROGENEOUS DISTRIBUTED SYSTEMS. 2015. AVAILABLE ONLINE: [HTTPS://WWW.TENSORFLOW.ORG](https://www.tensorflow.org) (22/05/2021)
- [8] THE DATABASE OF FACES. AT&T LABORATORIES CAMBRIDGE. [HTTPS://CAM-ORL.CO.UK/FACEDATABASE.HTML](https://cam-orl.co.uk/facedatabase.html) (22/05/2021)
- [9] WANG, ZHONGYUAN & WANG, GUANGCHENG & HUANG, BAOJIN & XIONG, ZHANGYANG & HONG, QI & WU, HAO & YI, PENG & JIANG, KUI & WANG, NANXI & PEI, YINGJIAO & CHEN, HELING & YU, MIAO & HUANG, ZHIBING & LIANG, JINBI. (2020). MASKED FACE RECOGNITION DATASET AND APPLICATION. (22/05/2021)
- [10] [HTTPS://GITHUB.COM/AMOSWISH/WEAR_MASK_TO_FACE](https://github.com/amoswish/WEAR_MASK_TO_FACE) (23/05/2021)
- [11] [HTTPS://PYPI.ORG/PROJECT/FACE-RECOGNITION/](https://pypi.org/project/face-recognition/) (23/05/2021)
- [12] J. DENG, W. DONG, R. SOCHER, L. LI, KAI LI AND LI FEI-FEI, "IMAGENET: A LARGE-SCALE HIERARCHICAL IMAGE DATABASE," 2009 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 2009, PP. 248-255, DOI: 10.1109/cvpr.2009.5206848. (23/05/2021)

- [13] K. HE, X. ZHANG, S. REN AND J. SUN, "DEEP RESIDUAL LEARNING FOR IMAGE RECOGNITION," 2016 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), 2016, PP. 770-778, DOI: 10.1109/CVPR.2016.90. (23/05/2021)
- [14] ZISSERMAN, ANDREW. (2014). VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. ARXIV 1409.1556. (23/05/2021)
- [15] C. SZEGEDY ET AL., "GOING DEEPER WITH CONVOLUTIONS," 2015 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), 2015, PP. 1-9, DOI: 10.1109/CVPR.2015.7298594. (24/05/2021)
- [16] TIN KAM HO, "RANDOM DECISION FORESTS," PROCEEDINGS OF 3RD INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION, 1995, PP. 278-282 VOL.1, DOI: 10.1109/ICDAR.1995.598994. (24/05/2021)
- [17] [HTTPS://SCIKIT-LEARN.ORG/STABLE/MODULES/GENERATED/SKLEARN.ENSEMBLE.RANDOMFORESTCLASSIFIER.HTML](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.randomforestclassifier.html) (24/05/2021)
- [18] [HTTPS://EN.WIKIPEDIA.ORG/WIKI/RANDOM_FOREST#/MEDIA/FILE:RANDOM_FOREST_DIAGRAM_COMPLETE.PNG](https://en.wikipedia.org/wiki/Random_Forest#/media/File:Random_Forest_Diagram_Complete.png) (24/05/2021)
- [19] BALABAN, STEPHEN. (2015). DEEP LEARNING AND FACE RECOGNITION: THE STATE OF THE ART. 94570B. 10.1117/12.2181526. (24/05/2021)
- [20] [HTTP://VIS-WWW.CS.UMASS.EDU/LFW/](http://vis-www.cs.umass.edu/lfw/) (25/05/2021)
- [21] [HTTPS://GITHUB.COM/SORINDRAGAN/FACE-RECOGNITION](https://github.com/sorindragan/face-recognition) (30/05/2021)
- [22] [HTTPS://GITHUB.COM/SAC52/TFG-MFR-CNNFE-ML](https://github.com/Sac52/TFG-MFR-CNNFE-ML) (01/06/2021)
- [23] [HTTPS://WWW.PYTHON.ORG/](https://www.python.org/) (01/08/2021)
- [24] TENSORFLOW, THE TENSORFLOW LOGO AND ANY RELATED MARKS ARE TRADEMARKS OF GOOGLE INC. [HTTPS://WWW.TENSORFLOW.ORG/](https://www.tensorflow.org/) (01/06/2021)
- [25] [HTTPS://KERAS.IO/](https://keras.io/) (01/06/2021)
- [26] [HTTPS://OPENCV.ORG/RESOURCES/MEDIA-KIT/](https://opencv.org/resources/media-kit/) (01/06/2021)
- [27] [HTTPS://NUMPY.ORG/](https://numpy.org/) (01/06/2021)
- [28] [HTTPS://SCIKIT-LEARN.ORG/STABLE/](https://scikit-learn.org/stable/) (01/06/2021)
- [29] [HTTPS://RESEARCH.GOOGLE.COM/COLABORATORY/](https://research.google.com/colaboratory/) (01/06/2021)
- [30] [HTTPS://WWW.GOOGLE.COM/INTL/ES_ES/DRIVE/](https://www.google.com/intl/es_es/drive/) (01/06/2021)

10. Anexo I. Proyecto

El material empleado en el proyecto y su implementación estará disponible a partir de la fecha de publicación en [22].