

UNIVERSIDAD DE MURCIA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA

Supervisión del uso de la mascarilla mediante algoritmos de visión artificial

Autor:

Juan José MORELL
FERNÁNDEZ
juanjose.morellf@um.es

Tutores:

Alberto RUIZ GARCÍA
a.ruiz@um.es
Ginés GARCÍA MATEOS
ginesgm@um.es

14 de junio de 2021

Me gustaría expresar mi agradecimiento al tutor Alberto Ruiz García, por el apoyo y ayuda durante el desarrollo de este proyecto.

Índice general

Resumen	3
Extended abstract	5
1. Introducción	11
1.1. Historia de la Visión Artificial	11
1.2. Reconocimiento Facial y COVID-19	12
2. Estado del arte	14
2.1. Python y OpenCV	14
2.2. Aplicaciones sin Deep Learning	15
2.3. Aplicaciones con Deep Learning	16
3. Análisis de objetivos y metodología	19
3.1. Prototipos a desarrollar	19
3.2. Herramientas	20
3.3. Fases de desarrollo	21
3.4. Dataset	21
4. Diseño y resolución	23
4.1. Paul Viola and Michael Jones	23
4.2. Facial Landmark	31
4.3. Implementación con Mediapipe	37
4.4. Implementación con TensorFlow	42
4.5. Comparación entre prototipos	48
5. Conclusiones y vías futuras	51
Bibliografía	56

Índice de figuras

1.1.	Ejemplos del uso de la mascarilla	13
2.1.	Salida generada por el detector facial.	15
2.2.	HOG generada tras estudiar un rostro.	16
2.3.	Salidas de ejemplo de Facial Landmark.	16
2.4.	Detecciones de ejemplo de YOLO	17
2.5.	Ejemplo de detección con el modelo MobileNet y Tensorflow.	17
2.6.	Soluciones de MediaPipe.	18
3.1.	Ejemplos de imágenes del dataset 1 y del dataset 2	22
4.1.	Haar-like Features.	24
4.2.	Funcionamiento de una <i>Imagen Integral</i>	25
4.3.	Construcción del <i>Strong Classifier</i>	26
4.4.	Construcción del <i>Multi-stage Classifier</i>	27
4.5.	Pruebas con Haar-like features con: <i>frontalface_default</i> y <i>frontalface_alt2</i> . . .	28
4.6.	Ejemplos de clasificaciones SVM	30
4.7.	Pruebas con prototipo Haar Custom.	30
4.8.	Fórmula <i>HOG</i>	32
4.9.	68 coordenadas del <i>facial landmark</i> con <i>iBUG 300-W dataset</i>	34
4.10.	Pruebas con Dlib's Facial Landmarks y modelo Haar-like feature <i>mcs_mouth</i> . .	36
4.11.	MediaPipe Graph utilizado en FaceMesh.	38
4.12.	Rendimiento de FaceMesh sobre dispositivos móviles.	39
4.13.	Pruebas con Mediapipe FaceMesh y modelo Haar-like feature <i>mcs_mouth.xml</i> .	41
4.14.	Arquitectura de una CNN con uso de SSD.	43
4.15.	Pruebas con SSD-MobileNetV2.	47

Resumen

En 2020 comenzó una gran pandemia, provocada por el coronavirus (COVID-19), que está marcando la historia de la humanidad, y por un tiempo, paralizó todo el funcionamiento de la misma. Tal es la importancia del COVID-19, que la Organización Mundial de la Salud (OMS) presentó medidas para evitar el contagio entre personas. Entre ellas se encuentra la norma del uso de mascarillas para reducir el riesgo de la exposición de la población al virus.

Para supervisar el uso correcto de las mascarillas en imágenes de personas, se plantea el uso de técnicas de visión artificial y *machine learning* para la implementación de prototipos capaces de llevar a cabo dicho control en tiempo real. Para ello, se centra el esfuerzo en el estudio de las siguientes técnicas: *Haar-like features*, *Facial Landmarks*, *Mediapipe*, *Transfer Learning/Tensorflow*.

La primera técnica, *Haar-like feature*, proviene de una investigación dirigida por Paul Viola y Michael Jones en el año 2001, centrada en el reconocimiento facial en tiempo real mediante el uso de dichas características y un modelo de *machine learning* llamado Adaboost. Una vez detectada la cara, para identificar el uso de la mascarilla se usa un modelo SVM (*support vector machine*), capaz de realizar regresiones y clasificaciones sobre un conjunto de datos, mediante la detección facial del modelo anterior.

La siguiente técnica, *Facial Landmarks*, se apoya en la anterior (*features*) para poder predecir puntos de interés del rostro humano, como: ojos, nariz y boca. Esto se logra por medio de una técnica llamada HOG (*histogram of gradient*), que se basa en el estudio de la dirección de un punto en la imagen utilizando derivadas, logrando obtener unos datos que posteriormente serán utilizados en la creación de un clasificador SVM, capaz

de realizar una detección/predicción en tiempo real. La implementación usada para el prototipo de esta técnica fue desarrollada por *Kazemi* y *Sullivan* en 2014.

Por otro lado, el tercer prototipo se centra en el uso de una de las soluciones implementadas en *Mediapipe*, framework de Python desarrollado por Google para la creación de aplicaciones de visión artificial de forma sencilla y potente, llamada *FaceMesh*. Junto a ella, se ha hecho uso de un modelo pre-entrenado de *Haar-like features* y se ha construido un prototipo capaz de realizar detecciones en tiempo real con gran precisión. La idea de esta técnica es detectar un rostro en la imagen de entrada, obtener la zona donde se encuentra la boca del rostro y aplicar el modelo pre-entrenado para detectar si se encuentra una boca o no.

En último lugar, se implementó un prototipo con la técnica de *Transfer Learning* y *Tensorflow*, plataforma que ha permitido su desarrollo. La idea consiste en la creación de un modelo personalizado para la detección de mascarillas. Concretamente se realizan un total de tres detecciones: mascarilla, no mascarilla, mal uso de la mascarilla. Para ello, se partirá de un modelo de *Deep Learning*, llamado *SSD-MobileNetV2*, y mediante el uso de *Transfer Learning* se logra crear dicho modelo personalizado. Esta técnica se caracteriza por reusar un modelo ya entrenado para un nuevo problema.

Para terminar, se ha realizado una comparación entre todos los prototipos anteriores para estudiar cual de ellos ofrece el mejor control de la norma presentada por la OMS. Como resultado, no existe un único prototipo capaz de destacar sobre el resto en todos los ámbitos. Cada una de las técnicas empleadas pueden solucionar el problema, destacando al primer prototipo (*haar-like features*) capaz de detectar rostros con mascarilla más lejanos, el tercer prototipo (*Mediapipe*) como el más veloz y el prototipo cuatro (*Tensorflow*) como el más acertado sobre el dataset, con un porcentaje de acierto del 40.27 % para el conjunto de test y un 43.8 % para todo el dataset con un 100 % de detecciones en ambos casos.

Extended abstract

The COVID-19 pandemic, also known as the coronavirus pandemic, is an ongoing global pandemic of coronavirus 2019 (SARS-CoV-2) disease that affects all the human population. The World Health Organization (WHO) declared a Public Health Emergency of International Concern regarding COVID-19 on January 30, 2020, and later declared a pandemic on March 11, 2020. This is why the WHO published an interim guidance to provide a updated recommendation on mask use in health care and community settings, and during home care for COVID-19 cases [43]. This document contains updated evidence and guidance on mask management, virus (SARS-CoV-2) transmission, mask use by the public in areas with community and cluster transmission, mask use during vigorous intensity physical activity, etc. In addition, the WHO advises the use of masks as part of a comprehensive package of prevention and control measures to limit the spread of COVID-19 [12]. For this reason, this work will propose the use of Computer Vision and Machine Learning techniques for the implementation of prototypes capable of carrying out the visual supervision of correct mask use in real time. The effort will be focused on the study of the following techniques: Haar-like features, Facial Landmarks, Mediapipe, Transfer Learning / Tensorflow.

The first technique, Haar-like features, comes from an investigation directed by Paul Viola and Michael Jones in 2001, focused on real-time facial recognition using these type of features and a Machine Learning model called Adaboost. Moreover, paper [42] describes a machine learning approach for visual object detection which is capable of processing images fast and quite accurately. This research can be divided into three key concepts. The first is a new image representation called *Integral Image*, which allows the features used to be computed faster during the execution of the algorithm. The second is a learning algorithm, based on AdaBoost, which selects a small set of features from a larger set of features and produces efficient classifiers. The third one is a method for com-

bining those classifiers into a cascade architecture, which allows discarding background regions of the input image and spending more computation on promising object-like regions. Therefore, in real-time applications, a detector which use this technique is typicall able to run at 15 frames per seconds.

The first prototype is created by using an SVM (Support Vector Machine) model and an OpenCV face detector based on Haar-like features. This is a supervised learning model with associated learning algorithms that analyze data for classification and regresion analysis, and it is created by detections of the OpenCV's face detector itself, with images of faces with a mask and without it, obtaining a prototype that performs a detection that runs at 15 ms in real time with a good accuracy.

The second technique, Facial Landmarks, is implemented in the Dlib library for Python. It allows the recognition of points of interest on the faces that have been detected in the input image. The process that makes Dlib for this technique can be divided into two main steps: detecting faces within the image, and obtaining those points of interest. This implementation is based on the research of Kazemi and Sullivan in 2014, with the paper *One Millisecond Face Alignment with an Ensemble of Regression Trees* [24]. This technique is capable of recognizing the following points of interest in a face: mouth, eyebrows, eyes, nose and chin, thanks to the use of an ensemble of regression trees, that can be used to estimate these points (*facial landmarks*) directly from a sparse subset of pixel intensities.

The first step is to find a face within the input image, using the HOG (Histogram of Oriented Gradients) method. It follows an idea similar to the Haar-like features method, since it is based on the detection of features. The theoretical idea behind HOG is to find the appearance and shape of an object by analyzing the intensity of local gradients, thanks to the fact that these gradients have a greater magnitude in the vicinity of edges or corners. While deploying, it splits the image into small regions, called cells, and a one-dimensional gradient histogram is calculated for each of the pixels in each cell. The second step is to estimate the points of interest accurately and efficiently on the face. It is based on gradient boosting for learning an ensemble of regression trees, in charge of predicting the points of interest.

Using OpenCV and the Dlib ToolKit, both ideas are implemented using pre-trained models. For facial detection and prediction of points of interest, it will be implemented with the models proposed by Dlib, more specifically the detection based on HOG and the prediction called *shape_predictor_68_face_landmarks.dat*. The prototype created performs detections at 14.7 ms but with poor accuracy for this task.

The third prototype is built with the use of Mediapipe, an API created by Google that offers customizable machine learning solutions for live and streaming media. Google recommends Mediapipe to build prototypes by combining existing perception components, to advance them to polished cross-platform applications, and measure system performance and resource consumption on target platforms [29]. Among the solutions it presents, there is a technique called Face Mesh that estimates 468 points of interest of a face, forming a 3D mesh in real time.

MediaPipe Face Mesh employs machine learning (ML) to infer the 3D surface geometry, requiring only a simple webcam like camera input. Using lightweight model architectures together with GPU acceleration throughout the pipeline, the solution delivers real-time performance critical for live experiences.

The pipeline used in this API consists of two Deep Learning models working at the same time. Its purpose is to perform a detection from an image of the points of interest on a face, and build a 3D face landmark model that approximates its surface by regression on said points. This task is facilitated if the face, where the points of interest must be detected, is clipped, thus making the model focus only on finding the points, increasing the precision of the prediction. Likewise, the cutouts of the faces can be generated from the previous predictions made by the same model, and the prediction is only called again when the presence of the face cannot be detected [15]. All this is implemented thanks to the *MediaPipe framework*. A pipeline is defined as a directed graph of components where each component is called a Calculator. These calculators are connected by data Streams. Each one represents a time-series of data Packets. So, the calculators and streams define a data-flow graph [29].

The prototype has been developed using FaceMesh from the MediaPipe API and a detector based on Haar-like features, such as the one mentioned in the Viola and

Jones paper. Therefore, the idea behind this prototype is to obtain the area of the mouth thanks to the use of the solution provided by Mediapipe, and detect if there is a mouth in that area using a Haar-like features model implemented in OpenCV with a pre-trained model. The prototype makes detections at 12.3 ms with good precision and accuracy at short distances.

The last prototype uses the Transfer Learning technique, and for that reason the use of Tensorflow is necessary. It is an Open Source platform dedicated to machine learning, that allows us to easily compile and deploy ML technology applications. It is based on tensors, multidimensional arrays with a uniform type, like np.arrays in Numpy. In the field of computer vision, it has a large number of pre-trained models capable of performing object detection and image classification quickly and accurately. This resource is called TensorFlow Model Garden, and it is a repository with different model implementations and solutions modeled for Tensorflow. The models are pre-trained using a dataset called COCO 2017. This is a large data set on a large scale for object detection, segmentation, and capture. Among these models, MobileNet stands out, capable of obtaining great performance on mobile devices and computers with little computational power. This has created a rise in popularity for detector implementations on embedded devices, such as the Raspberry Pi, or on ESP32 chips using remote processing.

The goal of this prototype is to use Transfer Learning on the MobileNet model to retrain it for our task. Transfer Learning is a technique in which a pre-trained model is reused for a new problem. Currently its use is becoming popular, since it allows the training of a deep neural network with a small amount of data. In this case, the aim is to use a general object detection model and, applying transfer learning, re-train it for a more specific task but without wasting its previous knowledge.

The prototype has been built with the use of an SSD-MobileNet model, a combination of an SSD (Single Shot MultiBox Detector) model with another model called MobileNet. This type of model is characterized by the use of a method to detect objects in images using a deep neural network, which generates prediction values about the presence of each object / category in each of the detections, and returns the object with the highest match. The dataset used in Transfer Learning is a combination of datasets composed of images of people with a mask, without it and with the mask incorrectly placed. It

consists of a total of 895 images, divided into two groups: train and test. The first group contains the images that are used to train the model and has 745 images. Whereas, the test set is used to validate the trained model, and it has 150 images. Once the images are selected, the labeling process is carried out, using a program created with Python called `labelImg`.

To perform Transfer Learning using Tensorflow, the following configuration files need to be created: *LabelMap*, two *TFRecord* files, and *pipeline.config*. This last file contains the main information of the model, both the number of classes that it can detect, the size that the images are resized before treating them, and checkpoints for training. For its creation, the use of Protobuf will be necessary. This is a Google tool focused on transforming the xml files generated by labeling the images in protocol buffers, to serialize the information in a structured and faster way. To carry out the training, we use the code provided after the installation of the Tensorflow Object Detection API, called *model_main_tf2*. After training, files called checkpoints are obtained, which contains the model that can be loaded for use in Tensorflow. Once the prototype is created, detections are made in a time of 83.3 ms, making the detections slower than the rest of the prototypes created but being a more scalable project.

Three metrics are used to compare the prototypes. The first comparison is based on the study of the working range of the prototypes, by measuring distance by the size of the detection that the algorithm implemented in the prototype can perform. The second comparison is based on the measure of the time the algorithm takes to process an input image. For this, the same test will be carried out on all prototypes, and the average time (ms) is calculated after an execution of one minute. Finally, the third comparison will study the percentage of correct classifications for the dataset built in this project. All the experiments have been executed in a PC with Intel i7 processor at 1.3GHz and 8 cores, Intel Iris Plus Graphics GPU, 8 Gb of RAM, and Ubuntu version 18.04.

As a result, there is no single prototype capable of standing out across the board, making it impossible to select an optimal solution to the problem in all cases. All the techniques used in this work can solve it and, depending on where the prototype will be applied, another study would have to be carried out to solve these specific objectives. For example, if we wanted to implement a detector of this type in a mobile device or

in an embedded system, it would be preferable a prototype that is fast and light, such as the third prototype (Mediapipe). Whereas, if we have a device with more computing power, it could be considered using a slower but more precise prototype such as the first one (Haar-like features) or even training a custom model such as the fourth prototype (Transfer Learning). Therefore, a preliminary study should be carried out on the device where the prototype will be applied to make a final decision.

CAPÍTULO 1

Introducción

La visión artificial es un ámbito de la informática que surgió hace 60 años, centrado en el estudio del procesamiento digital de las imágenes. En los últimos años ha tomado mucha importancia el reconocimiento facial, problema consistente en identificar rostros humanos dentro de una imagen, gracias a la investigación realizada por *Viola y Jones* en 2001, y la reciente tendencia en *smartphones* con el desbloqueo facial.

1.1. Historia de la Visión Artificial

Uno de los primeros acontecimientos que propició la visión artificial fue la creación del cable Bartlane, capaz de transmitir una imagen a través del océano Atlántico en los años 1920, con una duración de cerca a una semana [14]. Pero, dentro del entorno del procesamiento de imágenes digitales, la investigación se centró en recuperar una estructura tridimensional del mundo real a través de una imagen, para conseguir un entendimiento total de la escena que plasma la misma. Debido a esto, aparecieron varios algoritmos de reconocimiento de líneas, donde uno de ellos fue creado por parte de Huffman en 1971 [37].

Uno de los descubrimientos que inició este movimiento no fue proveniente de la informática, sino de la psicología. Esta sería una de las principales fuentes sobre el en-

tendimiento de cómo funciona la visión. Un par de psicólogos, David Hubel y Torsten Wiesel, describieron que el comportamiento de las neuronas encargadas de entender el entorno visual siempre empieza con estructuras simples como vértices. Más tarde, esta idea se convertirá en el principio central del *Deep Learning*.

Russell Kirsch, en 1959, es el primero en desarrollar un aparato que traducía las imágenes en datos que las máquinas pudiesen entender. Y, Lawrence Roberts en 1963 publica un estudio sobre cómo las máquinas pueden percibir objetos sólidos de tres dimensiones [6], uno de los avances considerados precursores de la visión artificial moderna.

Durante los años 1980, se desarrolló una red artificial capaz de reconocer patrones, mediante el uso de una red convolucional, que propició la creación de un modelo llamado LeNet-5, la primera red convolucional moderna. Este modelo se caracteriza por usar la *backpropagation* [6]. Mientras que en los años 1990 la visión artificial cambia totalmente de rumbo y los investigadores pasaron de intentar reconstruir objetos en 3D a intentar detectar objetos mediante sus características.

A partir del año 2000 se hacen muchos avances importantes y que actualmente son usados en aplicaciones reales. El primer detector facial llegaría en 2001, creado por Paul Viola y Michael Jones [42]. Ambos consiguieron crear el primero que funcionaba en tiempo real.

El problema de estos modelos es el uso de información para poder entrenarlos. Como consecuencia, se creó un proyecto llamado PASCAL VOC, que creó un dataset estándar para la clasificación de objetos [13]. Posteriormente, apareció en 2010 ImageNet, el cual contiene más de un millón de imágenes para un total de mil objetos. Junto a este apareció un modelo basado en una red convolucional llamado AlexNet [26]. Desde entonces, el *Deep Learning* se ha convertido en eje central del avance en la visión artificial, conjuntamente con todos los avances matemáticos realizados anteriormente.

1.2. Reconocimiento Facial y COVID-19

En la actualidad, la visión artificial se utiliza en muchos proyectos y está presente en investigaciones muy importantes para el futuro de la inteligencia artificial. Una de ellas se basa en el reconocimiento facial, y es usado en aplicaciones para reconocer personas,

gestos, atributos faciales, contar las personas, etc.

La detección facial se inicia con una imagen arbitraria, con el objetivo de encontrar todas las caras que hay en ella, y posteriormente devolver la localización exacta de cada una de las caras. Aunque esta tarea es natural para los humanos, es bastante complicada para los ordenadores, ya que se encuentran muchos factores que lo dificultan, tales como: la escala, localización, punto de vista, iluminación, lentes, etc. Existen centenares de investigaciones/proyectos sobre detección facial, desde uno de los más influyentes en los años 2000, como *Viola and Jones face detection*, a proyectos basados en *Deep Learning*, con tecnologías como *Tensorflow* o *YOLO* [37].

Tras el año 2020 y la aparición del COVID-19, el uso de mascarillas y el cumplimiento de las normas impuestas por la OMS (Organización Mundial de la Salud) están al orden del día. En este trabajo se tendrá como objetivo el estudio de todas estas tecnologías para poder controlar el uso correcto de la mascarilla facial, terminando con un prototipo capaz de detectar cuando una persona lleva mascarilla. En la Figura 1.1 se puede ver un ejemplo del uso correcto, incorrecto y mal uso de la mascarilla, extraídos del dataset de caras usado en la experimentación.

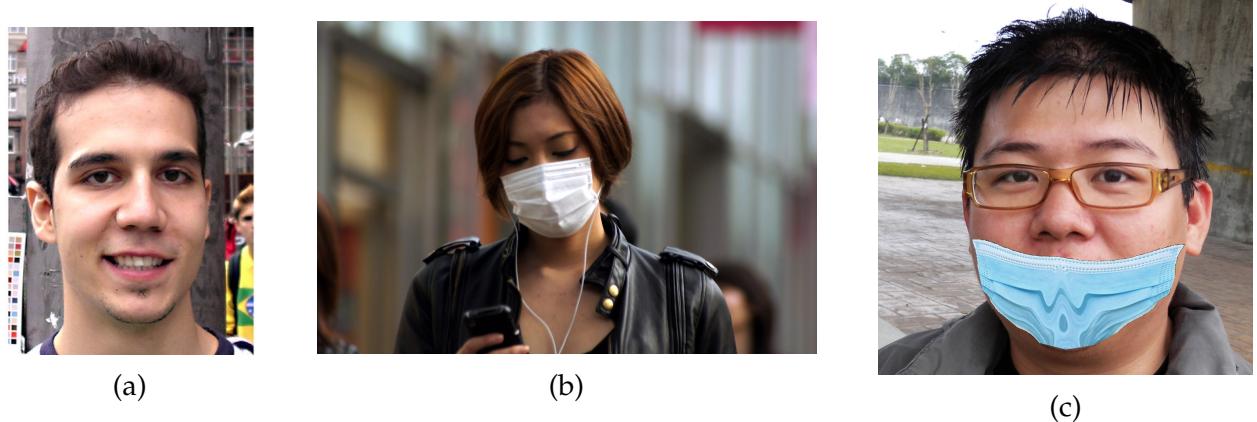


Figura 1.1: Ejemplos del uso de la mascarilla, extraídos del dataset usado en la experimentación.

La creación de un prototipo capaz de realizar esta comprobación permitiría su uso en la entradas a locales, comercios, transporte público, festivales, etc. Ofreciendo la posibilidad de la creación de aplicaciones con el uso de este, por ejemplo, la implementación de una aplicación capaz de reconocer si la persona no lleva de forma correcta la mascarilla y tiene fiebre, mediante un detector de temperatura, provocando un aviso.

CAPÍTULO 2

Estado del arte

El reconocimiento facial es un ámbito de la visión artificial que, como su nombre indica, se centra en la búsqueda de rostros humanos dentro de imágenes digitales. Durante años se han desarrollado muchas tecnologías capaces de realizar dicha acción, de las que se pueden distinguir dos grupos: aplicaciones con y sin el uso de *Deep Learning*. No obstante, ambos se centran en las características de los rostros humanos para lograr identificarlos, conocidas como *features*. Corresponden con puntos de la cara muy reconocibles, como: el mentón, ojos, cejas, nariz, etc. Esta técnica fue usada por primera vez en 2001, por Paul Viola y Michael Jones, y desde entonces se ha convertido en una de las técnicas principales en el reconocimiento facial.

Este problema se complica cuando las imágenes presentan inconvenientes naturales en su captura (como baja luz, ruido, etc.) o los individuos visten complementos que tapen sus rasgos faciales. Este es el problema que se va a plantear en este trabajo, buscar una posible solución al reconocimiento facial con complementos faciales, específicamente identificar si las personas llevan mascarillas.

2.1. Python y OpenCV

Python es un lenguaje de programación *Open Source* y de alto nivel, que permite trabajar y desarrollar aplicaciones de forma rápida y sencilla. Además, se trata de un lenguaje que ha aumentado en popularidad los últimos años gracias a su gran uso en ramas de la tecnología emergente, como *Data Science*, Inteligencia Artificial, *Big Data* y

Visión Artificial. En esta última, se ha visto potenciado gracias a la existencia de una herramienta llamada *OpenCV* [38], librería *Open Source* centrada en la creación de aplicaciones en tiempo real sobre visión artificial, que cuenta con una gran cantidad de implementaciones de algoritmos de *Computer Vision*.

2.2. Aplicaciones sin Deep Learning

Entre los años 2000 y 2015, se implementaron varias soluciones para el problema de la detección facial, centrados, su mayoría, en la extracción de características del rostro humano mediante técnicas de procesamiento de imágenes digitales. Podemos destacar los siguientes avances importantes durante estos años:

1. HAAR-like Features & ADABOOST.

Técnica diseñada por Paul Viola y Michael Jones en el año 2001, donde se implementa la detección mediante el uso de *machine learning*, que es capaz de procesar imágenes a una velocidad y precisión bastante altos (Figura 2.1), comparado con los desarrollos de la época, gracias al uso de técnicas avanzadas de procesamiento de imágenes y de un modelo de *machine learning* llamado *AdaBoost* [42].

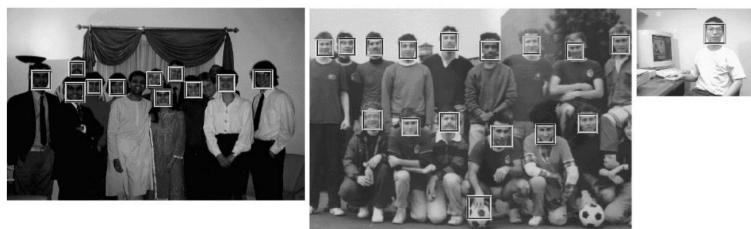


Figura 2.1: Salida generada por el detector facial [42].

2. HOG (Histogram of Gradient).

Se trata de una técnica de detección de rostros u otro tipo de objetos mediante el uso de gradientes, término dedicado al estudio de la dirección de un punto en la imagen utilizando derivadas (Figura 2.2). Estos servirán para el entrenamiento de un clasificador SVM (*support vector machine*), que será capaz de realizar una detección en tiempo real.



Figura 2.2: HOG generada tras estudiar un rostro.

3. Facial Landmarks.

Con la aparición de técnicas como las anteriores, se dio el siguiente paso con la predicción de puntos de interés sobre los rostros detectados (Figura 2.3). El desarrollo de esta se realiza con el uso de técnicas llamadas: *Gradient Boosting* y *Ensemble of regression trees*, ambas del ámbito del *Machine Learning* [24]. Cabe destacar que esta técnica detecta rasgos faciales (como boca, nariz u ojos) aunque se encuentren escondidos tras algún complemento u objeto, gracias al uso de la predicción.



Figura 2.3: Salidas de ejemplo de Facial Landmark [24].

2.3. Aplicaciones con Deep Learning

A partir de 2016, el *Deep Learning* fue una tecnología que ganó mucha importancia en la visión artificial. Un concepto importante del *deep learning* son las estructuras CNN (*Convolutional Neural Network*), capaces de extraer las características de las imágenes de entrada mediante el uso de filtros. Algunas de las principales implementaciones del *Deep Learning* para la detección de objetos/rostros son:

1. YOLO.

Modelo capaz de procesar imágenes en tiempo real a una velocidad de 45 frames

por segundo (con una GPU Nvidia Titan X), con el uso de una única red convolucional (CNN) que predice simultáneamente varios tipos de clases de objetos (Figura 2.4). Este modelo es entrenado mediante imágenes completas [34].

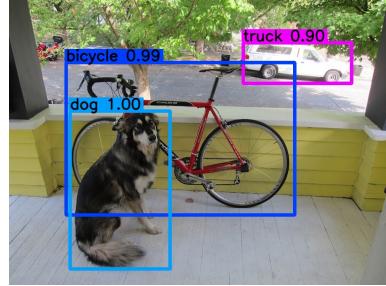


Figura 2.4: Detecciones de ejemplo de YOLO [34].

2. TensorFlow.

TensorFlow es una plataforma de *Open Source* dedicada al aprendizaje automático (AA). Permite compilar e implementar con facilidad aplicaciones con tecnología de AA. En el ámbito de la visión artificial, dispone de una gran cantidad de modelos pre-entrenados capaces de realizar detección de objetos y clasificación de imágenes de forma rápida y precisa. Entre dichos modelos destaca *MobileNet* [20], capaz de obtener un gran rendimiento en dispositivos móviles y ordenadores con poca potencia computacional (Figura 2.5). Esto ha ocasionado un aumento de la popularidad de las implementaciones de detectores en dispositivos embebidos, como *Raspberry Pi*, o en chips ESP32 mediante procesamiento remoto.



Figura 2.5: Ejemplo de detección con el modelo MobileNet y Tensorflow [20].

3. MediaPipe.

MediaPipe [29] es un framework creado por Google centrado en el desarrollo de aplicaciones de visión artificial de forma sencilla y potente. Se basa en un modelo

propio llamado *BlazeFace*, inspirado en modelos como *MobileNet*. Entre sus funcionalidades (Figura 2.6) se pueden encontrar: detección facial, predicción de una malla facial (*Face Landmarks*), detector de iris, detector de manos, reconocimiento de poses, segmentación de pelo, etc.

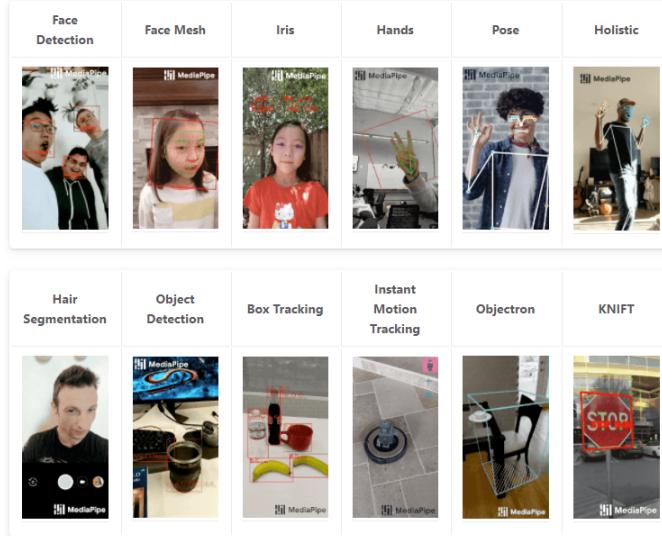


Figura 2.6: Soluciones de MediaPipe [17].

4. IBM Watson.

IBM Watson es un servicio ofrecido por IBM [21], que permite la creación de aplicaciones de inteligencia artificial en la nube. Este tipo de servicios nos permiten crear aplicaciones potentes sin necesidad de disponer de una gran potencia computacional en nuestra casa. Concretamente, IBM Watson dispone de un módulo llamado *Visual Recognition* que permite el análisis de imágenes y detección de objetos. Este tipo de servicios son de pago, pero permiten una gran cantidad de procesado gratuita. En este caso, IBM permite tener dos modelos personalizados y 1000 eventos de forma gratuita al mes [22].

CAPÍTULO 3

Análisis de objetivos y metodología

El COVID-19 es un pandemia que ha provocado en la humanidad incontables problemas, y en el mundo de la visión artificial también. Por ello, este TFG se ha centrado en la creación de un prototipo que sea capaz de revisar el cumplimiento de las normas COVID impuestas en España y en todo el mundo por la OMS. Concretamente, un prototipo que detecte cuándo una persona lleva, de manera correcta, una mascarilla al entrar a un comercio, cine, restaurante, etc. Los objetivos que se plantean para llevarlo a cabo son los siguientes:

- Estudio de las tecnologías de detección facial actuales, para comprobar su comportamiento con uso de mascarilla.
- Creación de un prototipo capaz de reconocer rostros y detectar si llevan puesta la mascarilla.
- Estudiar la capacidad de que el prototipo pueda identificar si se lleva correctamente la mascarilla, es decir, cubriendo la nariz y la boca.
- Poder detectar la mascarilla, independientemente del tipo o del dibujo que se lleve.

3.1. Prototipos a desarrollar

El desarrollo del TFG se divide en la creación de cuatro prototipos, centrados en cada una de las técnicas que se van a utilizar. Cada prototipo tendrá como objetivo final

detectar el rostro de una persona y clasificar si lleva mascarilla, bien o mal, o no la lleva.

Los prototipos serán probados en varios escenarios de prueba, todos ellos se realizarán en tiempo real. Se contará con tres escenarios distintos:

- Detección de mascarillas a una distancia cercana.
- Detección de mascarillas a una distancia media.
- Detección de mascarillas desde una posición alejada.

Asimismo, las pruebas se repetirán en dos dispositivos diferentes, el primero de ellos sin GPU y el segundo con GPU (CUDA). A continuación se muestran las especificaciones de los dispositivos en la tabla 3.1.

	PC 1	PC 2
CPU	Intel i7-1065G7 1.30GHz 8 núcleos	Intel i7-6700HQ 2.60GHz 8 núcleos
GPU	Intel Iris Plus Graphics	GTX 980M 2Gb
CUDA	NO	SI
RAM	16 Gb	8 Gb
OS	Ubuntu 18.04	Ubuntu 18.04

Tabla 3.1: Entornos de prueba.

Por último, los prototipos serán probados bajo un mismo *dataset* (mostrado en el apartado 3.4) para calcular el porcentaje de aciertos y detecciones que consiguen cada uno de ellos.

3.2. Herramientas

Para el desarrollo del prototipo se hará uso del lenguaje de programación de alto nivel Python. Junto a este se usarán las siguientes herramientas:

- *OpenCV*. Librería *Open Source* centrada en la creación de aplicaciones en tiempo real sobre visión artificial, que cuenta con una gran cantidad de implementaciones de algoritmos de visión artificial [38].
- *Dlib* [9]. Librería compuesta por implementaciones de algoritmos *Machine Learning*, centrada en la creación de aplicaciones que resuelven problemas del mundo real.

En concreto, se hará uso de su apartado de *HOG detector* [5] y *Facial Landmark* [24].

- *Scikit-learn*. Librería de *machine learning* capaz de realizar clasificación, regresión, *support vector machine* (SVM), *gradient boosting*, *k-mean*, etc. Se implementa conjuntamente con las librerías Numpy y SciPy [33].
- *Numpy*. Librería que añade funcionalidad a *arrays* multidimensionales y matrices, junto a un gran conjunto de operaciones matemáticas para trabajar con ellos [19].
- *Mediapipe*. Librería con soluciones y aplicaciones de *Machine Learning* para dispositivos móviles, en la nube o web [29].
- *Tensorflow*. Librería de *Machine Learning*, referente al entrenamiento y utilización de redes convolucionales (*Deep Learning*) [39].

3.3. Fases de desarrollo

El proceso de desarrollo del prototipo en cada una de las técnicas presentadas en este TFG, seguirá el siguiente flujo de trabajo:

1. *Investigación*. El primer paso se centra en la investigación del funcionamiento de la técnica a estudiar, priorizando el estudio del artículo oficial donde se presenta la técnica por sus autores. Y, posteriormente, se buscará información extra en libros o artículos web.
2. *Implementación básica*. El segundo paso se trata de realizar una implementación de la técnica estudiada, tal y como se presenta por los autores, con el objetivo de realizar un estudio de precisión y rapidez.
3. *Implementación propia*. Por último, se crea un prototipo de la técnica intentando resolver los objetivos establecidos, recabando datos, para una comparación final entre todas las técnicas estudiadas.

3.4. Dataset

Un *dataset* es un conjunto de imágenes utilizado para realizar el entrenamiento y validación de un modelo de *Machine Learning*. En este trabajo se hará uso de la combi-

nación de dos *dataset* existentes para representar todas las posibilidades del problema planteado, conteniendo imágenes de personas con mascarilla, sin ella y con ella pero mal colocada. El primer dataset de la combinación proviene del artículo de Kaggle llamado *Covid face mask detection dataset* [32] (Figura 3.1). Kaggle es una comunidad de *Machine Learning* y *Deep Learning* donde se comparten proyectos y dataset. El segundo dataset proviene de una investigación [3], llamada *Maskedface-net*, centrada en la creación de imágenes de personas donde se muestre un mal uso de la mascarilla (Figura 3.1). Finalmente, el dataset creado para este trabajo cuenta con un total de 895 imágenes mezcladas de ambas fuentes, donde 745 formarán el conjunto de entrenamiento y 150 el de test.



Figura 3.1: Ejemplos de imágenes del dataset 1 (3.1a, 3.1b, 3.1d, 3.1e, 3.1f, 3.1g) y del dataset 2 (3.1c, 3.1h, 3.1i). Se puede observar que en el segundo dataset la mascarilla se ha generado de forma artificial.

CAPÍTULO 4

Diseño y resolución

En este capítulo se describirá el diseño y resolución del problema planteado mediante el desarrollo de cuatro prototipos. El primer prototipo se basa en el algoritmo desarrollado por Paul Viola y Michael Jones, centrado en el uso de Haar-like features y un modelo de *Machine Learning* (Adaboost). *Facial Landmark* es la técnica usada para el segundo prototipo, estudiando concretamente el algoritmo implementado en *Dlib*, toolkit de algoritmos de *Machine Learning*. El tercer prototipo ha sido implementado mediante una solución de Mediapipe, *FaceMesh*. Mientras que el cuarto prototipo se ha construido mediante el uso de la técnica *Transfer Learning* y el framework *TensorFlow*. Por último, se prestará una comparación entre todos los prototipos mediante un estudio de su funcionamiento en una aplicación en tiempo real y bajo un dataset.

4.1. Paul Viola and Michael Jones

En 2001, el reconocimiento facial tuvo su primera importante aparición en el campo de la visión artificial como aplicación en tiempo real, como hemos visto en el capítulo 2. Este avance fue de la mano de Paul Viola y Michael Jones [42]. Análogamente, este será el punto de partida del estudio de este TFG. Durante este apartado, se estudiará el funcionamiento del algoritmo *Viola-Jones face detector*, ideado por estos dos investigadores y se presentará una implementación del mismo mediante *Python* y *OpenCV* para comprobar cómo se comporta en el problema estudiado.

Descripción del algoritmo

El trabajo de los expertos fue presentado por parte de la Universidad de Cambridge mediante un artículo de investigación muy influyente en el área. Y se introduce como:

"[...] This paper describes a machine learning approach for visual object detection which is capable of processing images extremely rapidly and achieving high detection rates" [42]

Para poder lograr este objetivo, se basan en un procedimiento de trabajo en dos fases: entrenamiento y detección. Igualmente, Paul y Michael dividen el proyecto en tres ideas principales para poder lograr un detector que se pueda ejecutar en tiempo real. Y estas son: la imagen integral, Adaboost (algoritmo de Machine Learning) y un método llamado *attentional cascade structure*.

Con todos estos puntos combinados, lograron ingeniar un algoritmo capaz de detectar caras humanas con un *frame rate* de 15 fps. Fue diseñado para la detección de caras frontales, haciéndose difícil para posiciones laterales o inclinadas.

Las imágenes que se toman para realizar la detección pasan por una transformación del espacio de color a escala de grises, con el objeto de encontrar características en ellas, llamadas *Haar-like features*. Fueron nombradas así por su inventor Alfred Haar en el siglo XIX. En este trabajo se hace uso de tres tipos de haar-like features (Figura 4.1).

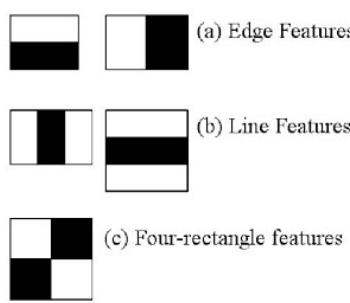


Figura 4.1: Haar-like Features [31].

Las *Haar-like features*, o también conocidas como *Haar-wavelet*, son una secuencia de funciones *rescaled square-shaped*, siendo similares a las funciones de Fourier y con un comportamiento parecido a los *Kernel* usados en las *Redes Convolucionales* (matrices que consiguen extraer ciertas *features* de la imagen de entrada), de manera que las *Haar-like features* serán las características de la detección facial.

En un estudio ideal, los píxeles que forma la *feature* tendrá una división clara entre píxeles de color blanco con los de color negro (Figura 4.1), pero en la realidad eso casi nunca se va a dar.

Más específicamente, las *Haar-like features* están compuestas por valores escalares que representan la media de intensidades entre dos regiones rectangulares de la imagen. Estas capturan la intensidad del valor de gris, la frecuencia espacial y las direcciones, mediante el cambio del tamaño, posición y forma de las regiones rectangulares basándose en la resolución que se define en el detector [31].

Estas características van a ayudar al ordenador a entender lo que hay en la imagen analizada. Van a ser utilizadas mediante *Machine Learning* para detectar dónde hay una cara o no, mediante un recorrido sobre toda la imagen. Esto conlleva una potencia de computación elevada. Para paliar este problema idearon el método de la *Imagen Integral*.

La *Imagen Integral* permite calcular sumatorios sobre subregiones de la imagen, de una forma casi instantánea. Además de ser muy útiles para las *HAAR-like features*, también lo son en muchas otras aplicaciones.

Si se supone una imagen con unas dimensiones de $\langle w, h \rangle$ (ancho y alto, respectivamente), la imagen integral correspondiente tendrá unas dimensiones de $\langle w + 1, h + 1 \rangle$. La primera fila y columna de esta son ceros, mientras que el resto tendrán el valor de la suma de todos los píxeles que le preceden [?]. Ahora, para calcular la suma de los píxeles en una región específica de la imagen, se toma la correspondiente en la imagen integral y se suma según la siguiente fórmula (siguiendo la numeración de la Figura 4.2):

$$\text{sum} = L_4 + L_1 - (L_2 + L_3)$$

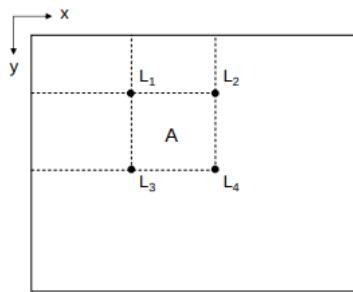


Figura 4.2: Funcionamiento de una *Imagen Integral* [?].

Viola y Jones juntaron esta idea con los filtros *Haar-like features*, y consiguen computar dichas características de manera constante y eficaz [7].

Una vez estudiada la obtención de características y con un set de entrenamiento, solo queda seleccionar un método de *Machine Learning* que permita crear una función de clasificación. Concretamente, se plantea el uso de una variante de *AdaBoost*, que permite seleccionar un pequeño conjunto de características y poder entrenar un clasificador.

Este algoritmo de aprendizaje está basado en generar una predicción muy buena a partir de la combinación de predicciones peores y más débiles, donde cada una de ellas se corresponde con la umbralización, o *threshold*, de una de las características *Haar-like*. La primera vez que aparece este algoritmo, de forma práctica, fue de la mano de *Freund y Schapire* [11]. Sin embargo, el usado por *Viola y Jones* es una modificación de este. La salida que genera el algoritmo *AdaBoost* es un clasificador llamado *Strong Classifier*, como se ha mencionado anteriormente, compuesto por combinaciones lineales de *Weak Classifiers*.

El procedimiento para encontrar *Weak Classifiers* es ejecutar el algoritmo T iteraciones, donde T es el número de clasificadores a encontrar. En cada iteración, el algoritmo busca el porcentaje de error entre todas las características y escoge la que menos porcentaje de error presente en dicha iteración [27], como se muestra en la *Figura 4.3*.

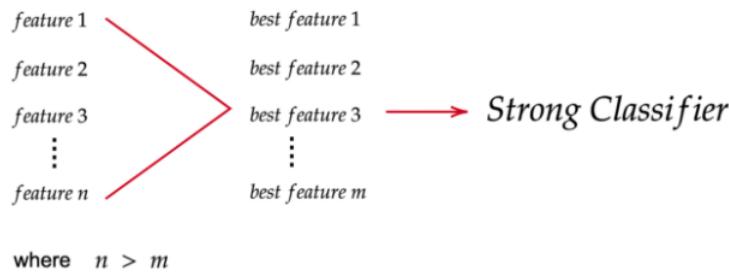


Figura 4.3: Construcción del *Strong Classifier* [27].

Con estos clasificadores, se procede a la construcción de una estructura en cascada para crear un *Multi-stage Classifier*, que podrá realizar una detección rápida y buena. Por tanto, la estructura de cascada está compuesta por varios estados de *Strong Classifiers* generados por el algoritmo *AdaBoost*. El trabajo de cada estado será identificar si, dada una región de la imagen, no hay una cara o si hay la posibilidad de que la haya [11]. Si el resultado de uno de los estados es que no existe una cara en dicha región, esta

se descarta directamente. Mientras que, si hay la posibilidad de que exista una, pasa al siguiente estado de la estructura. De esta forma, cuantos más estados atravesie una región de la imagen, con más seguridad se podrá afirmar que existe una cara en ella. La estructura completa se refleja en la *Figura 4.4*.

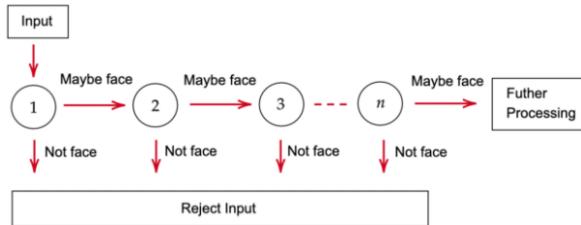


Figura 4.4: Construcción del *Multi-stage Classifier* [27].

Implementación y Experimentación

El prototipo será implementado en *Python*, con el uso de *OpenCV*. El objetivo es construir dos detectores de caras, donde el primero usará dos modelos pre-entrenados de *OpenCV* de caras frontales, para estudiar cuál es mejor para el siguiente prototipo. Mientras que, en el segundo, se intentará modificar el programa para que mediante el uso de uno de estos modelos pre-entrenados y uno de *Machine learning* se pueda detectar una cara con una mascarilla o sin ella.

La **implementación básica** hace uso de un modelo pre-entrenado cargado mediante una clase de *OpenCV*, llamada *Cascade Classifier*. Esta representa la base de *Machine Learning* explicada en el apartado anterior. Asimismo, *OpenCV* también proporciona una serie de archivos *xml* con diferentes modelos pre-entrenados. En concreto, para este prototipo se hace uso del modelo por defecto, detector de caras frontal, como se muestra en la investigación de *Viola y Jones*, y una variante del mismo denominada *frontalface_alt2.xml*. Finalmente, la detección se realiza, tras hacer una transformación del espacio de color a escala de grises, mediante la función *detectMultiScale* de la clase, creada anteriormente, *Cascade Classifier*. Concretamente, su funcionalidad será encontrar caras dentro de las imágenes que vaya procesando.

La prueba de este prototipo se realizará a una distancia corta (70 cm) y media (150 cm), para comprobar su funcionamiento estándar con dos modelos pre-entrenados, llamados: *frontalface_default.xml* y *frontalface_alt2.xml*, dedicados a reconocer rostros de ma-

nera frontal, intentando estudiar su velocidad de procesamiento y su precisión a la hora de detectar un rostro sin y con obstáculos. Algunos ejemplos del resultado obtenido se ven reflejados en la *Figura 4.5*.

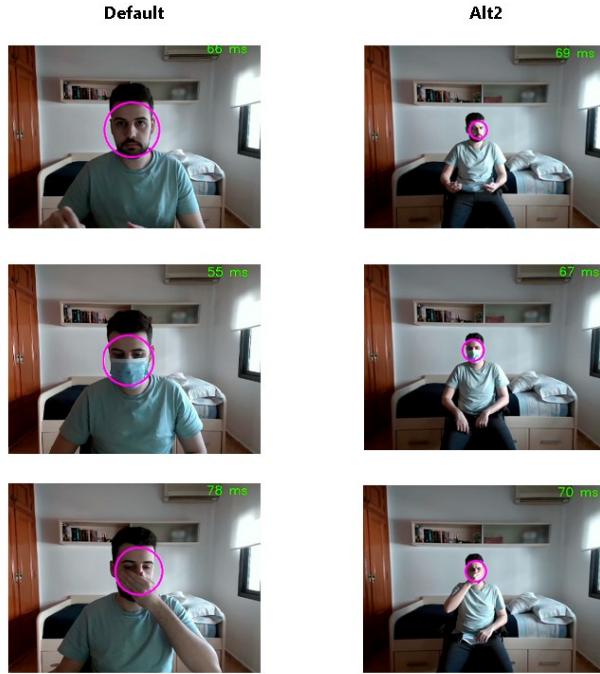


Figura 4.5: Pruebas con Haar-like features con: *frontalface_default.xml* y *frontalface_alt2.xml*

Tras la experimentación, se puede comprobar que ambos modelos funcionan bastante bien, tanto para un reconocimiento de rostro sin obstáculos como con ellos. Aunque es importante destacar que el modelo pre-entrenado *frontalface_default.xml* no es tan preciso en distancias superiores a los 150 cm. En cuanto a los tiempos medidos para estas pruebas, el segundo modelo pre-entrenado *frontalface_alt2.xml* obtiene un tiempo medio más lento pero funciona de forma más precisa. Se refleja en la tabla 4.1, tras realizar una media de 10 ejecuciones de una duración de un minuto:

	default (70cm)	default (150cm)	alt2 (70cm)	alt2 (150cm)
PC1 / Time (ms)	46.15	45.09	62.04	61.75
PC2 / Time (ms)	50.02	47.02	55.46	58.22

Tabla 4.1: Tiempos de ejecución para el prototipo 1.

El **segundo prototipo** (custom) implementa un detector de caras junto a un modelo de *Machine Learning* que identifica cuándo una persona lleva o no mascarilla, además de si la lleva puesta correctamente. Gracias a los modelos PCA y SVM, se puede crear un modelo para su identificación con el uso de muestras de los modelos pre-entrenados

Haar-like features probados antes. Para ello se realizará el siguiente esquema de procedimiento:

1. Haar-like features

Sigue el mismo procedimiento que el prototipo anterior. Mediante la creación de una clase de OpenCV, llamada *CascadeClassifier*, y el uso de un modelo *Haar-like features* pre-entrenado, se obtiene el ROI donde se localiza un rostro. En este caso, se hará un estudio con el modelo pre-entrenado llamado *frontalface_alt2.xml*, capaz de reconocer de forma más fiable rostros con mascarilla.

2. Modelo PCA/SVM

Una vez obtenido el ROI de la localización de la cara con o sin mascarilla, se procede a predecir de qué caso se trata. Previo a dicha predicción y ejecución del prototipo, se necesita entrenar el modelo. Y para ello se toman como referencias las imágenes de nuestro dataset, concretamente el conjunto de entrenamiento. El entrenamiento se realiza con la ejecución del archivo Python: *trainDataset.py*. El modelo final está formado por un algoritmo PCA (*Principal Component Analysis*) seguido de un modelo SVM (*Support Vector Machine*).

PCA es un algoritmo de *Machine Learning* dedicado a la reducción de dimensionalidad. Identifica un hiperplano común que conecta a todos los datos y son proyectados sobre él. *Scikit-Learn* ofrece una implementación de este, mediante descomposición SVD (Singular Value Decomposition), también conocida como una regresión lineal [18]. La salida de PCA servirá para reducir la complejidad de la entrada al modelo **SVM**, capaz de realizar regresiones o clasificaciones, tanto lineales como no lineales. La idea tras este modelo es separar los distintos conjuntos de datos existentes mediante líneas rectas en el espacio (Figura 4.6) [18].

Para la implementación de esta parte se hace uso de *Scikit-Learn*, librería de Python que implementa una gran variedad de clasificadores, regresores y cluster de modelos *Machine Learning*.

En resumen, el procedimiento que sigue este prototipo se basa en cargar el modelo que se ha creado, detectar un rostro en la imagen de entrada y a partir de ese ROI

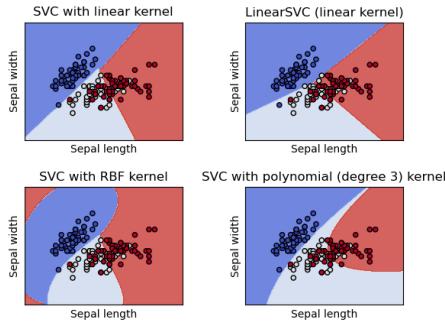


Figura 4.6: Ejemplos de clasificaciones SVM

(donde se encuentra un rostro), hacer un estudio con el modelo y mostrar por pantalla el resultado. Para la detección de los rostros antes de tratarlos con el modelo, se realizará con el modelo pre-entrenado *frontalface_default.xml*, para conseguir un funcionamiento más fluido. El funcionamiento del prototipo se muestra en la Figura 4.7.



Figura 4.7: Pruebas con prototipo Haar Custom.

El funcionamiento del prototipo en tiempo real es de manera frontal y cercana, tanto a 70 cm como a 150 cm, y se ha observado que funciona de forma correcta. Sin embargo, para una posición lejana y con más información a tratar, el detector se pierde y crea identificaciones falsas o no llega a reconocer nada. Por último, el tiempo de retardo que presenta este prototipo en tiempo real para el PC1 es de una media de 87.5 ms, y para el PC2 una media de 116.14 ms. Esto supone un aumento de unos 42 ms en relación a aplicar únicamente el detector de caras para el PC1 y de unos 67 ms para el PC2.

Este procedimiento se podría llegar a usar con otra implementación, específicamente con HOG, centrada también en la extracción de características pero usando otro tipo de descriptores basados en gradientes.

4.2. Facial Landmark

Con el objetivo de ampliar la idea anterior, se plantea el uso de Facial Landmark, una tecnología que nos permite el reconocimiento de puntos de interés en las caras que se han detectado en la imagen. Sus pasos de ejecución son: detectar caras dentro de la imagen y obtener dichos puntos de interés. La implementación que se va a utilizar es la propuesta por *Kazemi y Sullivan* en 2014, con el paper *One Millisecond Face Alignment with an Ensemble of Regression Trees* [25], y usado en el *toolkit Dlib*. Este método se centra en localizar las siguientes zonas faciales: boca, cejas, ojos, nariz y mentón, gracias al uso de un conjunto de árboles de regresión. Estos son entrenados mediante un modelo formado por puntos de interés de un grupo de imágenes, etiquetadas a mano y especificadas como coordenadas (x,y).

Dlib será el *toolkit (Open Source)* utilizado para la implementación de dicho método. Este contiene algoritmos de *Machine Learning* y herramientas capaces de crear software complejo en *C++* y *Python* para resolver problemas reales, sobre todo centrado en robótica, dispositivos embebidos, móviles y ordenadores de gran capacidad [9].

HOG - Histogram of Oriented Gradients

El primer paso en esta solución es encontrar una cara dentro de la imagen de entrada, y el encargado será el método HOG [5]. Este algoritmo sigue una idea similar al método de *Haar-like*, ya que se basa en la detección de *features* (características).

La idea teórica tras *HOG* es encontrar la apariencia y forma de un objeto mediante la distribución de la intensidad de gradientes locales, gracias a que estos obtienen una magnitud mayor en las cercanías de bordes o esquinas. La implementación divide la imagen en pequeñas regiones (llamadas celdas) y se calcula un histograma de gradientes de una dimensión para cada uno de los píxeles de cada celda. Para un mejor estudio, se normaliza el contraste de la imagen de entrada [5]. Con este procedimiento se obtiene un *feature vector* a partir de la imagen de entrada, y la distribución resultante de los gradientes serán usados como las características. La implementación de HOG se podría dividir en 5 pasos generales [30]:

1. *Pre-procesado.* Dada una imagen de entrada de cualquier tamaño es tratada en re-

giones de ciertas escalas y analizadas en varias zonas de la imagen. La única restricción es que los tamaños de las regiones analizadas tienen una relación de aspecto fija.

2. *Cálculo de las imágenes de gradiente.* Para el cálculo del histograma de gradientes es necesario realizar el cálculo de los gradientes, tanto verticales como horizontales. Esto se puede obtener fácilmente gracias al uso de *Kernels* (filtros). Posteriormente, se busca la magnitud del gradiente (g) y las direcciones de dichos gradientes (θ) con el uso de la siguiente fórmula (Figura 4.8):

$$g = \sqrt{g_x^2 + g_y^2}$$
$$\theta = \arctan \frac{g_y}{g_x}$$

Figura 4.8: Fórmula HOG [30].

La fórmula está implementada en *OpenCV* mediante la función *cartToPolar*. En este punto, se puede obtener la imagen de los gradientes, eliminando toda la información no relevante de la imagen original, observando cómo en todos los píxeles el gradiente tomará una magnitud y una dirección. Si la imagen es RGB, los gradientes se evalúan sobre los tres canales de color, siendo la magnitud final el valor mayor entre los tres canales.

3. *Cálculo de los histogramas de gradientes en celdas (8x8).* La imagen se divide en celdas y se calculan los histogramas para cada una de ellas. Por ejemplo, si la celda es de tamaño 8x8, esta contendrá 192 píxeles (8x8x3), donde el gradiente tiene dos valores, descritos anteriormente (magnitud y dirección), por cada uno de los píxeles, lo que añade 128 valores (8x8x2). Esto facilita la representación de las regiones mediante el uso de un histograma, además mejora la robustez frente al ruido. El tamaño de la celda vendrá definido dependiendo de la escala de características (*features*) que se estén buscando.
4. *Normalización de bloques (16x16).* Una vez creado el histograma, hay que tener en cuenta que la imagen es sensible al brillo que tenga. Por ejemplo, si el brillo de la imagen se divide en dos, la magnitud del gradiente también lo hará. De igual forma, si el brillo se multiplica por dos, el gradiente también. Pero, se busca un

descriptor que sea independiente a esto, por lo que se normaliza el histograma para que no se vea afectado por los cambios de luz/brillo. Disponiendo de celdas de 8×8 , un bloque de 16×16 posee cuatro histogramas que pueden ser condensados en un único vector normalizado.

5. *Cálculo del vector de HOG.* El último paso es concatenar los vectores normalizados obtenidos en uno global.

Este procedimiento se repite varias veces sobre imágenes distintas y se introduce en un modelo *Machine Learning* del tipo SVM Linear, con el objetivo de obtener un detector de caras. En el caso de *Dlib*, posee un modelo pre-entrenado.

Implementación de Kazemi y Sullivan

El artículo de Kazemi y Sullivan presenta la implementación usada en el *toolkit Dlib* del algoritmo que estima de forma precisa y eficiente los puntos de interés faciales. Está basado en *gradient boosting* para el aprendizaje de un conjunto de árboles de regresión (*ensemble of regression trees*), que será el encargado de la predicción de los puntos de interés [24].

Este método fue uno de los primeros que mejoró el rendimiento, a diferencia de los métodos anteriores, gracias a la detección de componentes esenciales para el alineamiento de las caras y procesarlos para introducirlos en funciones de regresión en cascada. Cada una de estas funciones estima, de forma eficiente, la forma facial desde una estimación inicial y obtiene un conjunto de píxeles indexados a dicha estimación. En concreto, *Dlib* estimará un total de 68 píxeles indexados (Figura 4.9), ya que sigue las anotaciones del *dataset iBUG 300-W* [35], usado en los modelos pre-entrenados ofrecidos por *Dlib*.

Además, se introdujeron dos elementos clave a las funciones de aprendizaje de regresión. El primero gira en torno a la intensidad de los píxeles indexados con respecto a la estimación actual de la forma facial, ya que estos son muy influenciables por la deformación de la estimación de la forma y a los cambios de iluminación. El dilema que aparece es que necesitamos características confiables para obtener una predicción con precisión de la forma y, por otro lado, necesitamos una estimación precisa de la forma para extraer

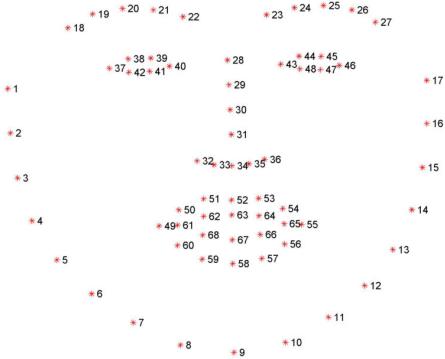


Figura 4.9: 68 coordenadas del *facial landmark* con *iBUG 300-W dataset* [35].

características fiables. Para resolver este dilema, se plantea el uso de un enfoque iterativo. La idea se basa en obtener una imagen transformada en un sistema de coordenadas normalizado basado en una estimación actual de la forma, y posteriormente extraer las características para predecir un vector de actualización para los parámetros de la forma. Este proceso suele repetirse varias veces.

El segundo elemento clave se trata de reducir la dificultad del proceso de inferencia/-predicción. El objetivo es obtener una función capaz de estimar la forma que concuerde con la información de la imagen y el modelo. Para resolver esto, se plantearon dos soluciones a lo largo del tiempo. El primero afirma que algunas de las predicciones estimadas durante el proceso mienten, no obteniendo un resultado deseado. Lo que concluyó siendo de ayuda para evitar este problema. Pero, posteriormente, se descubrió una segunda solución, donde se asume que las algunas predicciones estimadas mienten pero no es necesario realizar trabajo adicional. En el caso de este artículo, se implementa una solución donde se utilizan una combinación de ambas. Por lo que cada regresor aprende mediante *gradient boosting* conjuntamente a una función de pérdida de error al cuadrado. Esto permite realizar un estudio sobre un mayor número de características relevantes de forma eficiente. El resultado es una cascada de regresores que pueden localizar los puntos de referencia faciales cuando es inicializada con la media de la pose facial [24].

De forma más general, la investigación ofrece las siguientes contribuciones a las investigaciones de *Face Landmark* anteriores:

1. Un método de alineación basado en un conjunto de árboles de regresión que devuelve la forma facial mientras minimiza la función de error.

2. Método que maneja la predicción de puntos que faltan o no están presentes en la imagen. Por ejemplo, rostros que estén medio ocultos. (Esto solamente se realizará si el modelo creado con HOG detecta dicha cara oculta).
3. Resultados donde se demuestran que el método produce predicciones de alta calidad.
4. El efecto de la cantidad de datos de entrenamiento.

Prototipo

Mediante *OpenCV* y el *ToolKit Dlib* se obtiene la implementación de ambas ideas, mediante modelos pre-entrenados. Para la detección facial y predicción de puntos de interés, se usarán los modelos propuestos por Dlib, siendo el de detección basado en HOG y el de predicción llamado *shape_predictor_68_face_landmarks.dat*. El procedimiento que seguirá este prototipo será el siguiente:

1. Detección facial. Dlib dispone de una función donde inicializa un detector basado en HOG con el que se podrá determinar dónde se encuentra el rostro en la imagen. Para realizar la detección tiene que recibir una imagen transformada en escala de grises.
2. Predicción de *Facial Landmarks*. Una vez detectados los rostros de la imagen, se utiliza un predictor creado con la función *shape_predictor* de Dlib. Este es capaz de determinar los puntos de interés de un ROI con el rostro detectado anteriormente. En este punto es necesario el uso de un modelo pre-entrenado, exactamente el de 68 puntos de interés (nombrado anteriormente).
3. Se obtiene el ROI de la zona donde se encuentra la boca, a través de los puntos de interés obtenidos en el paso anterior.
4. Detección de una boca dentro del ROI. Mediante un modelo pre-entrenado y basándose en el prototipo anterior de *Haar-like features*, se construye un detector capaz de detectar una boca. Si se detecta una boca, se puede decir que la persona no lleva mascarilla, mientras que, si no es capaz de realizar una detección sí que llevaría mascarilla.

El tiempo de ejecución que presenta este prototipo en tiempo real para el PC1 es

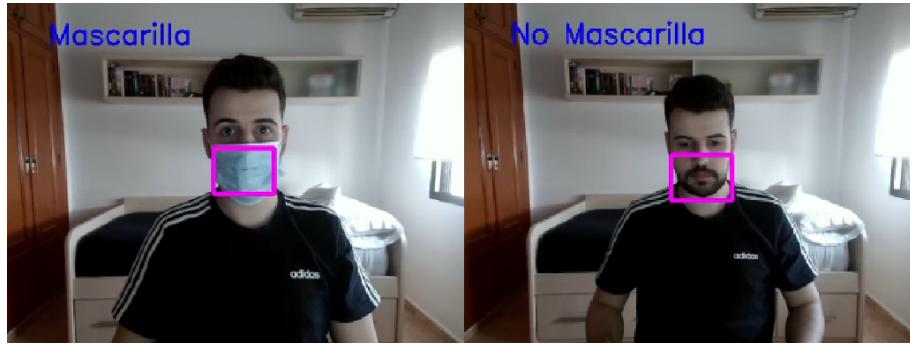


Figura 4.10: Pruebas con Dlib's Facial Landmarks y modelo Haar-like feature *mcs_mouth.xml*.

de una media de 14,7 ms y para el PC2 una media de 8,79 ms. Aunque estos tiempos presenten un buen resultado, el funcionamiento del prototipo en tiempo real no es el esperado. Por un lado, le cuesta reconocer un rostro con mascarilla, incluso en muchas ocasiones sin lograrlo, ya en el primer tipo de prueba (distancia de 75 cm). Por otro lado, el detector facial sin mascarilla funciona correctamente y de manera eficaz (Figura 4.10). Por lo que otro planteamiento que se podría desarrollar sería un prototipo donde si aparece una detección de una persona y su boca, significa que no viste una mascarilla. Por tanto, no cumpliría con la normativa y saltaría la alarma.

Próximos pasos

En los siguientes apartados se probará implementar prototipos con el uso de Deep Learning. En las últimas décadas, se ha convertido en uno de los métodos más usados para la creación de aplicaciones de visión artificial [37]. Está basado en las estructuras neuronales denominadas CNN (*Convolutional Neural Network*), principales responsables de que un ordenador pueda procesar de forma sencilla una imagen. CNN es una combinación entre capas neuronales y convolucionales, siendo estas últimas filtros con los que se obtendrán características de la imagen de entrada, las conocidas *features*. La principal función del Deep Learning es clasificar, ya sea una imagen, un objeto o incluso varios objetos a la vez. Esto es gracias al uso de modelos, entrenados con miles de imágenes, que consiguen clasificar imágenes en varias categorías, por ejemplo *MobileNet* [10].

4.3. Implementación con Mediapipe

Mediapipe es una API *open-source* creada por *Google*, que ofrece servicios de *Machine Learning* para vídeos y fuentes multimedia. Entre ellas, se encuentra un servicio llamado *Face Mesh* que ofrece una solución que estima 468 puntos de interés de un rostro, conformando una malla 3D en tiempo real. Este usa aceleración GPU conjuntamente con un modelo y el uso de una *pipeline*.

La *pipeline* que se utiliza en esta API consiste en dos modelos de *Deep Learning* que trabajan al mismo tiempo. Su funcionalidad es realizar una detección a partir de una imagen de los puntos de interés sobre una cara y construir un modelo *face landmark* 3D que aproxima la superficie de esta mediante regresión sobre dichos puntos. Esta tarea es facilitada si la cara, donde se tienen que detectar los puntos de interés, se encuentra recortada, haciendo así que el modelo se centre solamente en buscar los puntos, aumentando la precisión de la predicción. Asimismo, los recortes de las caras se puede generar a partir de las predicciones anteriores realizadas por el mismo modelo, y solamente es llamada la predicción nuevamente cuando no se consigue detectar la presencia de la cara [15].

Todo esto es implementado gracias al framework *MediaPipe*, con la herramienta *MediaPipe graph* [29], arquitectura caracterizada por estar formada por componentes llamados *Calculator*, nodos del grafo que tras la entrada de cero o más inputs generan cero o más salidas. Todos estos nodos están conectados mediante datos en forma de *Streams*, donde cada uno representa un conjunto de datos-tiempo en *Packets*. Por tanto, los *calculators* y *streams* definen el flujo de datos del *Graph* [29].

El *pipeline* puede ser definido mediante la adición/modificación de *calculators* dentro del *graph*. Específicamente, el *pipeline* que se utiliza en esta solución (*FaceMesh*) está formado por un *graph* compuesto por un *subgraph* de *face landmark* (proveniente del módulo, ya implementado, de *face landmark* de *Mediapipe*), que a su vez usa otro *subgraph* proveniente de *face detection module* para la detección de caras, y un *face renderer subgraph* para mostrar el resultado [15]. En concreto, el *graph* que se usa en esta implementación es el mostrado en la Figura 4.11.

Por tanto, el procesamiento de una imagen en este modelo sigue dos pasos. El pri-

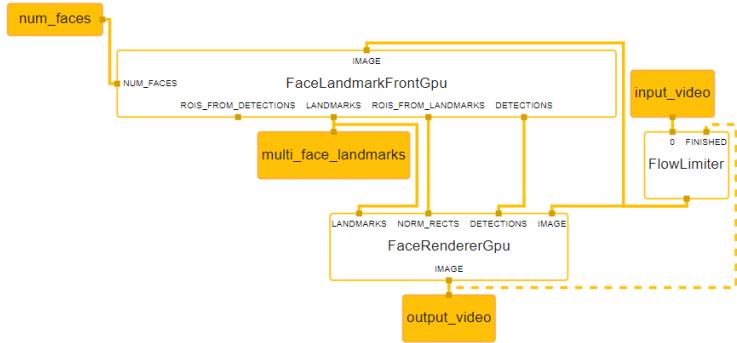


Figura 4.11: MediaPipe Graph utilizado en FaceMesh [16].

mero (1), toma la imagen de entrada, capturada por la cámara, es procesada por un detector de caras *lightweigth*, llamado *BlazeFace*, y produce unos rectángulos que definen el perímetro donde se encuentra la cara, conjuntamente con un par de puntos de interés superficiales (ojos, boca y nariz). Estos puntos se utilizarán para alinear la cara para el siguiente paso. Y el segundo (2), mediante el rectángulo obtenido en el paso anterior, se recorta la cara de la imagen inicial y es re-escalado para utilizarse como entrada de la red neuronal que realiza la predicción de la malla. El tamaño de re-escalado será entre 256x256 píxeles en un modelo completo, y 128x128 en el modelo más pequeño. Tras la predicción, se obtiene como salida un vector de coordenadas *landmark 3D*, que serán mapeadas y dibujadas en la imagen original [23].

Las coordenadas que se obtienen como salida están compuestas por los valores x e y provenientes de localizaciones del plano 2D propio de la imagen. Mientras que la coordenada z es interpretada como una profundidad relativa a un centro de masas que compone la malla de la cara.

Se utilizan dos modelos para el funcionamiento de *FaceMesh*. El primero de ellos dedicado a la detección facial, llamado *BlazeFace* (mencionado anteriormente), es un modelo *lightweight* creado para GPU móviles, llegando a una velocidad de procesamiento entre 200 y 1000 fps en dispositivos móviles punteros. Es inspirado en los modelos *MobileNet*, tanto la primera versión como la segunda, provenientes del *framework SSD* (*Single Shot Multibox Detector*). Este modelo produce una salida compuesta por un rectángulo perimetral y 6 puntos de interés faciales [2].

El segundo modelo, *Face Landmark Model*, es generado mediante *Transfer Learning*

buscando los siguientes objetivos: crear coordenadas 3D (mencionadas anteriormente) y conseguir mostrarlas en la imagen de salida de forma correcta [15]. El *Transfer Learning* es una técnica de *Machine Learning* donde se puede hacer uso de un modelo pre-entrenado para personalizarlo y usarlo en una tarea determinada. Conviene destacar que un modelo entrenado es una red almacenada, entrenada previamente con un conjunto de datos con el objetivo de realizar una tarea de clasificación de imágenes a gran escala [41].

FaceMesh propone una implementación mediante *TensorFlow Lite* que dispone de dos formatos: CPU y GPU, que presentan un rendimiento en dispositivos móviles (*Pixel3*, *Pixel2* y *iPhoneX*) muy fluido, impresionando sobre todo su funcionamiento sobre CPU, como se puede ver en la Figura 4.12 [1].

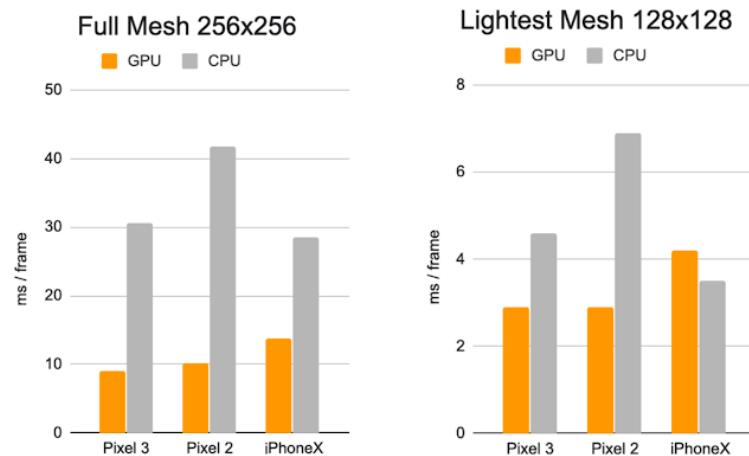


Figura 4.12: Rendimiento de FaceMesh sobre dispositivos móviles [1].

Prototipo

El prototipo de este apartado se realizará mediante el uso de *FaceMesh* del framework *MediaPipe* y un detector basado en *Haar-like features*, como el mencionado en el *paper* de Viola & Jones (descrito en la sección 4.1). Los pasos que lo definen son los siguientes:

1. Detección FaceMesh

Para la detección del rostro en la imagen de entrada se utiliza la solución de MediaPipe llamada FaceMesh, implementada en el framework de Python que recibe el mismo nombre, *mediapipe*. Esta solución dispone de dos parámetros de configuración: *min_detection_confidence* y *min_tracking_confidence*. El primero hace referencia al valor mínimo de *confidence* para considerar una detección como buena, mientras

que el segundo se entiende como el valor mínimo para considerar una predicción de *facial landmark* buena.

Una vez configurado el FaceMesh, se procede a tratar la imagen de entrada para facilitar su tratamiento. Se convierte el espacio de color de la imagen de BGR, espacio de color con el que trabaja *OpenCV*, a RGB. Y, para mejorar el rendimiento, se elimina la opción de la imagen para que sea escribible (*writeable*). Este proceso será invertido después de realizar la detección.

2. Obtención de zona ROI

Tras la detección de Landmarks, el resultado se almacena en una variable llamada *multi_face_landmarks*, donde se encuentran todas las marcas faciales de todos los rostros detectados. A su vez, este contiene cada una de las marcas, con una totalidad de 468 puntos.

Estas marcas contienen parámetros para medir la precisión de la detección del mismo, llamados: *visibility* y *presence*. Si estos valores son muy bajos, no se tienen en cuenta. Tras esto, se normalizan sus valores a coordenadas con respecto a los píxeles de la imagen de entrada, se reservan únicamente las coordenadas relacionadas con la boca y se obtiene el ROI donde se localiza esta.

3. Detección de una boca en el ROI

Para el último paso se hace uso de un modelo pre-entrenado de *Haar-like features* llamado *mcs_mouth.xml*, capaz de detectar una boca. Si dentro del ROI no se detecta nada, significa que la persona tiene una mascarilla puesta.

Por tanto, la idea tras este prototipo es obtener la zona de la boca gracias al uso de la solución proporcionada por Mediapipe, *FaceMesh*. Y, posteriormente, detectar mediante un modelo Haar-like features si existe una boca en dicha zona. Se utiliza la misma idea planteada en el prototipo anterior, pero intentando mejorar su funcionamiento con el uso del framework *Mediapipe*. La desventajas de este prototipo es que si la persona obstaculiza la zona de la boca a la hora de la detección, podría ser capaz de engañar al algoritmo haciendo pensar que sí porta una mascarilla, cuando en realidad no es así. Lo bueno es que no hace falta distinguir entre tipos de mascarillas.

4.3. Implementación con Mediapipe



Figura 4.13: Pruebas con Mediapipe FaceMesh y modelo Haar-like feature *mcs_mouth.xml*

El tiempo de retardo que presenta este prototipo en tiempo real para el PC1 es de una media de 12,3 ms y para el PC2 una media de 15.1 ms. Tras la realización de las pruebas (Figura 4.13), se puede concluir que el prototipo funciona de una manera bastante eficaz y precisa en distancias cercanas, mientras que en una posición lejana tarda en detectar a la persona, hasta que esta no se encuentre a una distancia cercana. Destaca un funcionamiento más veloz en el PC1 sin GPU, semejante al resultado mostrado en la Figura 4.12 para el caso de iPhoneX (*Lightest Mesh 128x128*).

4.4. Implementación con TensorFlow

TensorFlow es una plataforma de *Open Source* dedicada al aprendizaje automático. Permite compilar e implementar con facilidad aplicaciones con tecnología de aprendizaje automático (AA). Esta se basa en tensores, matrices multidimensionales con un tipo uniforme. Si se está familiarizado con NumPy, los tensores son como *np.arrays*, con la característica de que nunca se puede actualizar el contenido de un tensor, solo crear uno nuevo [39].

Concretamente, se usarán los modelos y ejemplos de aprendizaje automático ofrecidos y entrenados mediante la API de alto nivel de TensorFlow para la implementación del último prototipo. Este recurso recibe el nombre de *TensorFlow Model Garden* [40], y se trata de un repositorio con diferentes implementaciones de modelos y soluciones modeladas para TensorFlow. Los modelos están pre-entrenados mediante un dataset llamado COCO 2017 [4], siendo este un gran conjunto de datos a grande escala para *object detection*, segmentación y captación.

Model Garden dispone de un apartado dedicado a la visión artificial, donde se encuentran los ámbitos de clasificación de imágenes, como *MNIST*, *ResNet* o *EfficientNet*, y detección de objetos y segmentación, donde se destacan *RetinaNet*, *Mask R-CNN*, etc. Sin embargo, este prototipo se centrará en la implementación de uno de los modelos que ofrece Model Garden, llamado *SSD-MobileNet*, y aplicar *Transfer Learning* sobre él, para obtener un detector sobre una tarea específica, comprobar si la mascarilla está bien colocada.

Transfer Learning es una técnica en la que se reusa un modelo pre-entrenado para un nuevo problema. Últimamente su uso se está popularizando, ya que permite el entrenamiento de una *deep neural network* con una pequeña cantidad de datos. Esto es algo bastante revolucionario, puesto que todos los modelos hasta ahora necesitaban millones de datos, clasificados a mano y posteriormente necesitan una gran capacidad computacional para ser entrenados [41]. En este caso, se busca utilizar un modelo general de detección de objetos y aplicando transfer learning, re-entrenarlo para una tarea más específica pero sin desperdiciar sus conocimientos previos.

SSD-MobileNet

El prototipo de TensorFlow que se va a construir usará un modelo *SSD-MobileNet*, combinación de un modelo SSD, *Single Shot MultiBox Detector*, con otro llamado MobileNet. Este tipo de modelos se caracterizan por el uso de un método para detectar objetos en imágenes mediante una *deep neural network*, que genera valores de predicción sobre la presencia de cada objeto/categoría en cada una de las detecciones y devuelve el objeto con el que más coincide.

El modelo *SSD* se caracteriza por estar formado por dos componentes principales: un modelo *backbone* y una cabeza *SSD* (Figura 4.14). El *backbone* es un modelo de clasificación de imágenes, que está pre-entrenado para ser utilizado como un extractor de características. Normalmente, se implementa con modelos como *ResNet*. Mientras que la cabeza *SSD* está formada por una o más capas convolucionales que interpretan la salida del backbone como *bounding boxes* y clases de objetos [8].

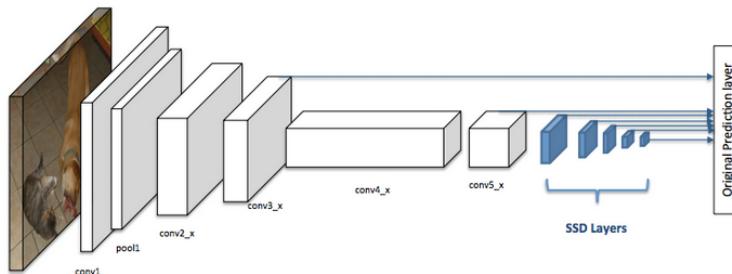


Figura 4.14: Arquitectura de una CNN con uso de SSD [8].

Otras características destacables de *SSD* son la división de la imagen de entrada mediante un *grid* y realizar una predicción de la clase y la localización del objeto en cada una de las celdas de dicho *grid*. Si en dicho *grid* no existe ningún objeto, se considera como *background* y la localización es ignorada. A cada celda se le puede asignar varias *anchor boxes* (*Bounding boxes* con una altura y anchura predefinida) con un tamaño igual al de las celdas. Pero no todos los objetos que están contenidos en la imagen tienen dicho tamaño. Por eso se añade un parámetro (*ratio*) dedicado a especificar los diferentes valores que pueden tomar las *anchor boxes*. Asimismo, existe otro parámetro llamado *zoom* para especificar cuánto pueden escalar dichos *boxes*, tanto aumentando como reduciendo [8].

El comportamiento y aportaciones específicas de este modelo, vienen planteadas en el artículo con su mismo nombre, "SSD: Single Shot MultiBox Detector" [28]. Y son las siguientes:

1. Se introduce un modelo para la detección de múltiples categorías que es más eficaz y preciso que los modelos posteriores como YOLO.
2. El núcleo de SSD predice la categoría y localización de la *bounding box* mediante el uso de pequeños filtros convolucionales aplicados sobre mapas de características.
3. Para obtener una mayor precisión, se realizan predicciones sobre mapas de características con diferente escalado.
4. Experimentación con análisis de tiempo y precisión sobre el modelo evaluado sobre PASCAL VOC, COCO y ILSVRC. De la misma forma, es comparado con modelos recientes.

MobileNet es un modelo con arquitectura CNN para la clasificación de imágenes y visión artificial en móviles. Lo que hace especial este modelo es la potencia computacional necesaria para ejecutarlo o aplicar *transfer learning* sobre él. Esto lo convierte en un perfecto candidato para dispositivos móviles, sistemas integrados y ordenadores sin GPU o baja eficiencia computacional, sin llegar a comprometer significativamente la precisión de los resultados.

Este modelo se basa en el uso de una arquitectura optimizada que utiliza convoluciones separables en profundidad para construir *deep neural networks* ligeros. Además, proporciona dos hiperparámetros globales que permiten ajustar de forma eficiente entre la latencia y precisión. Por consiguiente, con la combinación de ambos, se logra la creación de un modelo apto para dispositivos y ordenadores con poca potencia de GPU, capaz de obtener una detección de forma precisa y veloz.

Prototipo

1. Instalación del framework

Para realizar este prototipo se hace uso del API *object_detection* proporcionado por TensorFlow para Python. Tras seguir las instrucciones ofrecidas, se consigue un entorno de trabajo con las siguientes herramientas: Anaconda junto a una versión de Python 3.7, TensorFlow, CUDA, TensorFlow Model Garden, Object Detection API, Protobuf y COCO API.

2. Elección del modelo

A la hora de elegir un modelo se tienen en cuenta dos parámetros principales. El primero, Coco mAP (*main Average Precision*), indica la precisión media del modelo. Mientras que, el segundo, Speed, mide la velocidad de refresco.

SSD-MobileNetV2 (320x320) será el modelo elegido, ya que dispone de las mejores medidas en cuanto a velocidad. Exactamente cuenta con una medida *Coco mAP* de 20.2 y 19 ms de *Speed*. Aunque la precisión sea más baja que el resto de modelos, se suple con la gran velocidad que logra este modelo.

3. Dataset de imágenes y labeling

Para el dataset, se ha realizado una combinación de datasets donde aparecen imágenes de personas con mascarilla, sin ella y con ella pero mal puesta [3] [32]. Consta de un total de 895 imágenes, divididas en dos grupos, entrenamiento y test. En el primero se encuentran las imágenes que se utilizarán para el entrenamiento del modelo por *Transfer Learning* y cuenta con 745 imágenes. Mientras que el conjunto de test se utilizará para validar el modelo entrenado con un total de 150 imágenes.

Antes de realizar el modelo es necesario preparar las imágenes junto con un archivo donde se plasmen las etiquetas del mismo. Para ello, se hace uso de un programa (hecho en Python) llamado *labelImg*. Al tratar las imágenes, se crean unos archivos *xml* donde se encuentra la información del etiquetado que se ha realizado. En este caso se tiene uno de los tres *labels* del trabajo: *mask*, *noMask* o *badMask*. Una vez creados todos los archivos de etiquetado *xml* con los labels, se tiene que crear un archivo *pbtxt* donde se reflejen todas las labels posibles que tendrá el modelo.

En resumen, un *LabelMap* es una archivo referido a enumerar todas las clases posibles en la identificación de objetos. Además, cada una de las imágenes de entrenamiento dispondrá de un archivo adicional donde aparezcan cada *label* de la misma y su localización. Dependiendo del modelo usado, los archivos de *LabelMap* serán de un tipo u otro. En el caso de TensorFlow se utiliza un *TFRecord*, mientras que en YOLO se usa un *txt*.

Por último, se usa un script proporcionado por TensorFlow para transformar el etiquetado que hemos realizado al formato correcto para el entrenamiento del modelo (*TFRecord*).

4. Transfer Learning

Antes de hacer el entrenamiento, se tiene que preparar un archivo de configuración sobre el modelo seleccionado (SSD-MobileNetV2). Este archivo tiene el nombre de *pipeline.config* y contiene la información principal del modelo, tanto el número de clases que puede detectar, el tamaño que reajusta las imágenes antes de tratarlas y *checkpoints* para el entrenamiento. Para su creación será necesario el uso de las siguientes herramientas: TensorFlow, Object Detection API y Protobuf. Este último es una herramienta de Google centrada en transformar los *xml* de etiquetado que hemos creado antes en *protocol buffers*, para poder serializar la información de forma estructurada y más rápida.

Para realizar el entrenamiento se hace uso del código proporcionado tras la instalación de la API Object Detection de TensorFlow, llamado *model_main_tf2*. A este hay que cederle la siguiente información: dirección de salida, archivo de configuración y número de pasos de entrenamiento. Tras el entrenamiento, se obtienen unos archivos llamados *checkpoints* con los que se podrá cargar el modelo para su uso mediante la siguientes clases de TensorFlow: *Model_builder* y *Checkpoint*.

5. Realizar detección

Para la detección de la mascarilla se crea una función, *detect_fn*, que tomando una imagen se pre-procesa con el objetivo de reducir su tamaño al de la entrada del modelo, siendo en este caso 320x320. Tras esto, se realiza la detección usando el modelo que hemos creado y se realiza un post-procesado para obtener una imagen de salida del mismo tamaño que la de entrada.

El último paso será realizar esta detección en tiempo real, mediante el uso de OpenCV.

Una vez capturado un *frame* de la cámara de entrada, se debe transformar en una entrada compatible con el modelo, en este caso un tensor. Con este ya se puede detectar el objeto con la función que se ha definido anteriormente, *detect_fn*.

De esta manera, cuando se obtiene el *label* que se ha detectado, se utiliza la herramienta de visualización de la API Object Detection, *visualization_utils*, para mostrar el resultado por pantalla. Un ejemplo del funcionamiento se puede observar en la Figura 4.15.

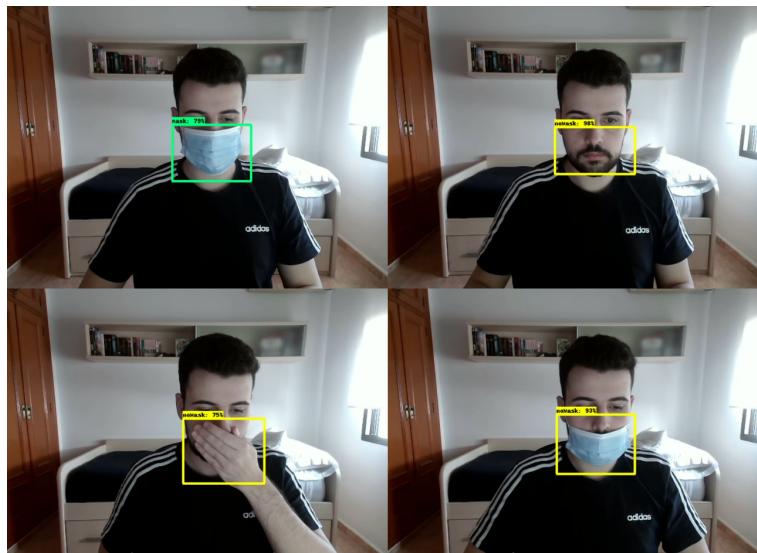


Figura 4.15: Pruebas con SSD-MobileNetV2.

El tiempo de retardo que presenta este prototipo en una ejecución en tiempo real para el PC1 es de una media de 83.2 ms y para el PC2 de 67.49 ms. El resultado obtenido es bastante bueno cuando solamente funciona en un ámbito cerrado. Esto se debe a que el entrenamiento del modelo usado se ha realizado con imágenes más. Aun así, este prototipo tras un entrenamiento con un dataset más amplio, funciona de forma correcta pero con más errores, destacando que su funcionamiento correcto sucede en una distancia a cámara muy cercana. Cuanto más lejos se sitúe la persona de la cámara, más detecciones falsas se crearán, sobre todo en el caso de encontrar una mascarilla cuando no la hay. Sería posible mejorar su funcionamiento si se utiliza un dataset más amplio aún y un entrenamiento más largo. Por ejemplo, el prototipo realiza detecciones de *badMask* cuando debería ser *noMask* o viceversa, haciendo difícil la diferenciación entre ambas. De igual forma, si se encuentra en una situación donde la iluminación es escasa o la cámara no está bien colocada, el rendimiento del prototipo disminuye.

4.5. Comparación entre prototipos

Este apartado está dedicado a la comparación de los prototipos implementados anteriormente, con el objetivo de contrastar el funcionamiento de todos ellos y las técnicas implementadas. Para ello, se hará uso de tres medidas: distancia de funcionamiento, el tiempo que tarda en procesar una imagen y porcentaje de acierto/detección bajo el dataset. Las dos primeras serán sobre una ejecución en tiempo real.

Distancia

La primera comparación se basa en el estudio del rango de funcionamiento de los prototipos, pudiendo medir su distancia mediante el tamaño de la imagen que se le pasa como entrada al algoritmo. En la siguiente tabla (Tabla 4.2) se presenta esto mediante una medida en píxeles (x,y) que corresponde con la medida de la detección más pequeña que el prototipo puede procesar. Además, aparecen dos medidas: la primera cuando no se lleva mascarilla y la segunda cuando sí.

Prototipo 1	Prototipo 2	Prototipo 3	Prototipo 4
(25,25)	(32,32)	(18,26)	(320, 320)*
(25,25)	(68,68)	(38,44)	(320, 320)*

Tabla 4.2: Rango píxeles de los prototipos (pruebas realizadas en el PC1).

En el caso del último prototipo, todas las imágenes que procesa son re-escaladas a unas dimensiones de 320x320, debido a las características que presenta el mismo, obteniendo como resultado detecciones con unos píxeles cercanos a estos valores.

En conclusión, como resultado final se obtiene que el prototipo 1, a pesar de no utilizar técnicas avanzadas de *deep learning*, obtiene el mejor resultado de los cuatro prototipos, pudiendo detectar rostros de hasta 25x25 píxeles dentro de la imagen de entrada.

Tiempo de ejecución

La segunda comparación se basa en la medida del tiempo que tarda el algoritmo en tratar una imagen de entrada. Para ello se realizará la misma prueba en todos los prototipos, mostrando en la siguiente tabla (Tabla 4.3) el tiempo medio (ms) de todas las imágenes analizadas tras una ejecución de un minuto.

	Prototipo 1	Prototipo 2	Prototipo 3	Prototipo 4
PC1	87.5 ms	14.7 ms	12.3 ms	83.2 ms
PC2	116.14 ms	8.79 ms	15.1 ms	67.49 ms

Tabla 4.3: Tiempo de ejecución de los prototipos.

El resultado del estudio, sobre el PC1, presenta al prototipo 3 como el más veloz, mientras que el prototipo 1 es el más lento. Esto se debe a la tecnología implementada en el tercer prototipo, que presenta un modelo *deep learning* centrado en realizar un trabajo rápido en cualquier dispositivo, incluyendo móviles. Mientras que en el PC2, se puede observar una mejora de rendimiento en el prototipo 4, gracias al uso de la GPU y CUDA. Pero el resto de prototipos obtienen un resultado semejante al del PC1.

Dataset

La última comparación será observar el porcentaje de aciertos/detecciones de cada uno de los prototipos para el *dataset* creado para este trabajo. Para ello, el experimento se centrará en el porcentaje de aciertos tanto del conjunto test como del dataset al completo. Además, se mostrará el total de imágenes con detecciones, ya que si no se consigue realizar una detección, tampoco se podrá saber si la persona lleva o no mascarilla. El resultado de los experimentos aparece en las siguientes tablas 4.4 (conjunto de test) y 4.5 (conjunto completo):

	Prototipo 1	Prototipo 2	Prototipo 3	Prototipo 4
Acierto (%)	29.53 / 52.35	45.64	55.03	40.27 / 65.77
Detecciones (%)	82.55 / 82.55	69.8	79.87	100 / 100

Tabla 4.4: Porcentaje de acierto / resultados para el conjunto test del dataset.

	Prototipo 1	Prototipo 2	Prototipo 3	Prototipo 4
Acierto (%)	51.35 / 65.77	51.46	53.04	43.8 / 80.07
Detecciones (%)	83.33 / 83.33	68.36	72.97	100 / 100

Tabla 4.5: Porcentaje de acierto / resultados para el dataset completo.

Para el primer y cuarto prototipo, se presentan dos resultados. El primero hace referencia al porcentaje de aciertos conseguidos cuando hay tres clases para clasificar: *Mask*, *NoMask* y *BadMask*. Mientras que el segundo resultado es el caso cuando la clase *BadMask* se trata como *NoMask*. Asimismo, los experimentos en el segundo y tercer prototipo se realizan solamente con la clasificación de dos clases: *Mask* y *NoMask*.

Como resultado de esta comparación, se obtiene que el cuarto prototipo consigue un porcentaje alto de aciertos, teniendo en cuenta que consigue detectar el 100 % de las imágenes que trata. Sin embargo, es importante destacar el funcionamiento de los demás prototipos, consiguiendo detectar/clasificar las imágenes de entrada con un buen resultado.

En consecuencia, se puede decir que no hay ningún prototipo capaz de destacar en todos los ámbitos, haciendo imposible la selección de una solución general para el problema. Todas las técnicas usadas en este trabajo son capaces de resolverlo, y dependiendo de dónde se tenga que aplicar el prototipo, se tendría que realizar otro estudio para resolver los objetivos del cliente. Por ejemplo, si se desea implementar un detector de este tipo en un dispositivo móvil o en un sistema embebido, se buscaría un prototipo que sea rápido y poco pesado, como el tercer prototipo (*Mediapipe*). Mientras que, si se dispone de un dispositivo con más potencia de computo, se podría plantear usar un prototipo más lento pero preciso como el primero (*Haar-like features*) o incluso entrenar un modelo personalizado como el cuarto prototipo (*Transfer Learning*). Aun así, se debería de realizar un estudio previo en el dispositivo donde se implementará el prototipo para poder realizar una decisión definitiva.

CAPÍTULO 5

Conclusiones y vías futuras

Durante el desarrollo de este trabajo se ha estudiado y probado la utilización de cuatro técnicas sobre la tarea de detección facial con mascarilla, para asegurar el cumplimiento de las normas COVID-19 impuestas por la OMS (Organización Mundial de la Salud). Estas técnicas son las siguientes: Haar-like features, Facial Landmarks, Mediapipe y Transfer Learning.

Según los resultados obtenidos durante el estudio, se concluye en que las técnicas presentadas consiguen realizar detecciones de rostros con mascarillas, cumpliendo así el primer objetivo impuesto. Los mejores resultados se logran con el primer prototipo, centrado en el uso de *Haar-like features* combinado con *Machine Learning* para la creación de un modelo apto para medir el cumplimiento de las normas COVID-19, durante una ejecución en tiempo real. No obstante, el cuarto prototipo es el que mejor resultado obtiene bajo las pruebas realizadas con el *dataset*. De igual forma, el resto de técnicas son capaces de realizar dicha tarea, destacando el buen rendimiento del prototipo con Mediapipe y los malos resultados de *Facial Landmark*, incapaz de detectar el rostro con mascarilla en muchas ocasiones.

De forma análoga, un prototipo de los desarrollados es capaz de reconocer cuándo la persona detectada viste mal la mascarilla, siendo el caso de Transfer Learning y Tensorflow, gracias a la creación de un modelo adaptado a estas necesidades, pero uni-

camente logrando un buen funcionamiento en situaciones ideales: buena luz y cercanía a la cámara. Finalmente, es importante destacar el último de los objetivos: tratar los falsos positivos. Los prototipos creados a partir de Haar-like features, Facial Landmarks y Mediapipe presentan dicha característica, ya que realizan detecciones de 'llevar mascarillas' en situaciones donde la persona no la lleva, por ejemplo: cuando la persona obstaculiza la visibilidad de la boca. El motivo proviene de la naturaleza de estos prototipos, dado que basan su funcionamiento en el uso del reconocimiento de características, y al no encontrar la boca se supone el uso de mascarilla. El único caso apto para este objetivo es el prototipo de Tensorflow, comentado anteriormente.

A lo largo del desarrollo de este trabajo se ha aprendido a emplear herramientas como Python, Tensorflow, frameworks de Python (Mediapipe, Dlib, Numpy) y OpenCV, además de profundizar en términos e ideas del *Machine Learning*, visión artificial y *Deep Learning*.

A modo de cierre, se pueden destacar las siguientes vías futuras para continuar este trabajo:

- Realizar más pruebas en distintos dispositivos (ordenadores, móviles, sistemas embebidos).
- Ampliar el funcionamiento de los prototipos, creando modelos más completos.
- Utilizar servicios en la nube para implementar los prototipos.
- Realizar el estudio de más técnicas.
- Añadir más normas COVID-19 en los prototipos, como por ejemplo el respeto de la distancia social.

Bibliografía

- [1] Artsiom Ablavatski and Google AI Ivan Grishchenko, Research Engineers. Real-time ar self-expression with machine learning. 03 2019. Acceso: 25-04-2021. URL: <https://ai.googleblog.com/2019/03/real-time-ar-self-expression-with.html>.
- [2] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. Blazeface: Sub-millisecond neural face detection on mobile gpus. 07 2019.
- [3] Adnane Cabani, Karim Hammoudi, Halim Benhabiles, and Mahmoud Melkemi. Maskedface-net – a dataset of correctly/incorrectly masked face images in the context of covid-19. *Smart Health*, 19:100144, Mar 2021. URL: <http://dx.doi.org/10.1016/j.smhl.2020.100144>, doi:10.1016/j.smhl.2020.100144.
- [4] COCO. Coco, common objects in context, 2021. Acceso: 02-05-2021. URL: <https://cocodataset.org/#home>.
- [5] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, 2005. doi:10.1109/CVPR.2005.177.
- [6] Rostyslav Demush. A brief history of computer vision (and convolutional neural networks). 02 2019. URL: <https://hackernoon.com/a-brief-history-of-computer-vision-and-convolutional-neural-networks-8fe8aacc79f3>.
- [7] Konstantinos Derpanis. Integral image-based representations. 1, 01 2007.
- [8] ArcGIS Developers. How single-shot detector (ssd) works?, 2021. Acceso: 05-05-2021. URL: <https://developers.arcgis.com/python/guide/how-ssd-works/>.
- [9] Dlib. Dlib library, 2021. Acceso: 25-04-2021. URL: <http://dlib.net>.
- [10] Govinda Dumane. Introduction to convolutional neural network (cnn) using tensorflow. 2020. Acceso: 28-04-2021. URL: <https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83>.
- [11] Robert E. Schapire. Explaining adaboost. 2020. Acceso: 11-04-2021. URL: <https://www.math.arizona.edu/~hzhang/math574m/>.

- [12] WHO Headquarters (HQ) Emergencies Preparedness. Mask use in the context of covid-19. page 22, December 2020. URL: [https://www.who.int/publications/item/advice-on-the-use-of-masks-in-the-community-during-home-care-and-in-healthcare-settings-in-the-context-of-the-novel-coronavirus-\(2019-ncov\)-outbreak](https://www.who.int/publications/item/advice-on-the-use-of-masks-in-the-community-during-home-care-and-in-healthcare-settings-in-the-context-of-the-novel-coronavirus-(2019-ncov)-outbreak), doi:WHO/2019-ncov/IPC_Masks/2020.5.
 - [13] M. Everingham, L. Gool, C. K. Williams, J. Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88:303–338, 2009.
 - [14] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Pearson, 2018.
 - [15] Google. Mediapipe face mesh. 2020. Acceso: 21-04-2021. URL: https://github.com/google/mediapipe/tree/master/solutions/face_mesh.
 - [16] Google. Mediapipe visualizer, 2021. Acceso: 20-05-2021. URL: <https://viz.mediapipe.dev/>.
 - [17] Google. ML solutions in mediapipe, 2021. Acceso: 19-05-2021. URL: <https://github.com/google/mediapipe/tree/main/solutions/ml>.
 - [18] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly, 2020.
 - [19] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi:10.1038/s41586-020-2649-2.
 - [20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. arXiv:1704.04861.
 - [21] IBM. Ibm watson. 2021. URL: <https://www.ibm.com/es-es/watson>.
 - [22] IBM. Visual recognition, 2021. Acceso: 07-05-2021. URL: <https://cloud.ibm.com/catalog/services/visual-recognition>.
 - [23] Yury Kartynnik, Artsiom Ablavatski, Ivan Grishchenko, and Matthias Grundmann. Real-time facial surface geometry from monocular video on mobile gpus. 07 2019.
 - [24] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1867–1874, 2014. doi:10.1109/CVPR.2014.241.
 - [25] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. 06 2014. doi:10.13140/2.1.1212.2243.
-

- [26] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014. arXiv:1404.5997.
- [27] Socret Lee. Understanding face detection with the viola-jones object detection framework. 2020. Acceso: 11-04-2021. URL: <https://towardsdatascience.com/understanding-face-detection-with-the-viola-jones-object-detection-framework-c55cc2a9da14>.
- [28] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2, doi:10.1007/978-3-319-46448-0_2.
- [29] Camillo Lugaressi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Ubweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for building perception pipelines. 06 2019.
- [30] Satya Mallick. Histogram of oriented gradients explained using opencv. 2016. Acceso: 02-05-2021. URL: <https://learnopencv.com/histogram-of-oriented-gradients/>.
- [31] Takeshi Mita, Toshimitsu Kaneko, and Osamu Hori. Joint haar-like features for face detection. *IEEE Int Conf Comp Vis*, 2:1619 – 1626 Vol. 2, 11 2005. doi:10.1109/ICCV.2005.129.
- [32] Prithwiraj Mitra. Covid face mask detection dataset, 2020. Acceso: 23-05-2021. URL: <https://www.kaggle.com/prithwirajmitra/covid-face-mask-detection-dataset>.
- [33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [34] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016. arXiv:1506.02640.
- [35] Christos Sagonas, Georgios Tzimiropoulos, Stefanos Zafeiriou, and Maja Pantic. 300 faces in-the-wild challenge: The first facial landmark localization challenge. In *2013 IEEE International Conference on Computer Vision Workshops*, pages 397–403, 2013. doi:10.1109/ICCVW.2013.59.
- [36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. arXiv: 1801.04381.
- [37] R. Szeliski. *COMPUTER VISION: Algorithms and applications*. SPRINGER NATURE, 2021.
- [38] OpenCV Team. Opencv. 2021. URL: <https://opencv.org/>.

- [39] TensorFlow. Tensorflow core, 2021. Acceso: 02-05-2021. URL: https://www.tensorflow.org/guide?hl=es_419.
- [40] Tensorflow. Tensorflow official models, 2021. Acceso: 02-05-2021. URL: <https://github.com/tensorflow/models/tree/master/official>.
- [41] TensorFlow. Transferir el aprendizaje y la puesta a punto, 2021. Acceso: 25-04-2021. URL: https://www.tensorflow.org/tutorials/images/transfer_learning.
- [42] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *IEEE Conf Comput Vis Pattern Recognit*, 1:I–511, 02 2001. doi: 10.1109/CVPR.2001.990517.
- [43] Wikipedia. Covid-19 pandemic, 2021. Acceso: 27-05-2021. URL: https://en.wikipedia.org/wiki/COVID-19_pandemic.