

UNIVERSIDAD DE MURCIA

TRABAJO FINAL DE GRADO

Supervisión visual de normas COVID

Autor:

Juan José MORELL
FERNÁNDEZ
juanjose.morellf@um.es

Tutor:

Alberto RUIZ GARCIA
a.ruiz@um.es

21 de mayo de 2021

Those who can imagine anything, can create the impossible.

by Alan Turing

Índice general

Resumen	3
Extended abstract	4
1. Introducción	6
1.1. Historia de la Visión Artificial	6
1.2. Reconocimiento Facial y COVID-19	8
2. Estado del arte	9
2.1. Python y OpenCV	10
2.2. Aplicaciones sin Deep Learning	10
2.3. Aplicaciones con Deep Learning	11
3. Análisis de objetivos y metodología	13
3.1. Prototipo	14
4. Diseño y resolución	16
4.1. Paul Viola and Michael Jones	16
4.2. Facial Landmark	24
4.3. Mediapipe	30
4.4. Tensorflow	35
4.5. Comparación	42
5. Conclusiones y vías futuras	43
Bibliografía	46

Índice de figuras

2.1.	Soluciones de MediaPipe [15]	12
4.1.	Haar-like Features [26]	17
4.2.	Funcionamiento de una <i>Imagen Integral</i> [2].	18
4.3.	Construcción del <i>Strong Classifier</i> [22].	19
4.4.	Construcción del <i>Multi-stage Classifier</i> [22].	20
4.5.	Pruebas con Haar-like features con: <i>frontalface_default.xml</i> y <i>frontalface_alt2.xml</i>	21
4.6.	Ejemplos de clasificaciones SVM	22
4.7.	Pruebas con prototipo Haar Custom	23
4.8.	Fórmula <i>HOG</i> [25].	25
4.9.	68 coordenadas del <i>facial landmark</i> con <i>iBUG 300-W dataset</i> [28].	27
4.10.	Pruebas con Dlib's Facial Landmarks y modelo Haar-like feature <i>mcs_mouth.xml</i> .	29
4.11.	MediaPipe Graph utilizado en FaceMesh [14].	31
4.12.	Rendimiento de FaceMesh sobre dispositivos móviles [1].	32
4.13.	Pruebas con Mediapipe FaceMesh y modelo Haar-like feature <i>mcs_mouth.xml</i>	34
4.14.	Arquitectura de una CNN con uso de SSD [8].	36
4.15.	Pruebas con SSD-MobileNetV2 <i>mcs_mouth.xml</i>	40

Resumen

Extended abstract

This project faces the problem of...

aa

CAPÍTULO 1

Introducción

La visión artificial es un ámbito de la informática que surgió hace 60 años, pensado en el estudio del procesamiento digital de las imágenes. En los últimos años ha tomado mucha importancia el reconocimiento facial, algoritmo capaz de 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. 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.

Pero el avance de estas investigaciones pronto se ligaría con el del ordenador. Estos, se podrían resumir en varios acontecimientos importantes, tales como: la invención

del transistor por Bell Laboratories en 1948 y de los circuitos integrados en 1958 o la introducción por parte de IBM de los primeros ordenadores personales en 1981 [12].

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 entendimiento de como 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 empiezan 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 como las máquinas perciben objetos sólidos de tres dimensiones, uno de los avances considerados precursores de la visión artificial moderna.

Durante los años 1980s, 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 backpropagación. Mientras que en los años 1990s 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. Ambos consiguieron crear el primero que funcionase en tiempo real.

El problema de estos modelos es el uso de información para poder entrenarlos, como consecuencia se creo un proyecto llamado PASCAL VOC, que creo un dataset estándar para la clasificación de objetos. 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. 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 esta 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, aspectos, contador de personas, etc.

La detección facial se inicia con una imagen arbitraria, con el objetivo de encontrar todas las caras que hay en una imagen, y posteriormente devolver otra con 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 2000s, como *Viola and Jones face detection*, a proyectos basados en *Deep Learning*, con tecnologías como *Tensorflow* o *YOLO* [30].

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 dichas normas, terminando con un prototipo capaz de detectar cuando una personas lleva mascarilla, ya sea al entrar a un comercio, evento u trabajo.

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 varias tecnologías capaces de realizar dicha acción, de las que se pueden distinguir dos grupos: aplicaciones con y sin el uso de *Deep Learning*. Aunque, ambos se centran en las características de los rostros humanos para lograr identificarlos. Conocidas como *features*, que 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 unas de las técnicas principales en el reconocimiento facial.

Este ejercicio se complica cuando las imágenes presentan errores naturales en su toma (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, en específico 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* o Visión Artificial. En esta última gracias a la existencia de una herramienta llamada *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 *Computer Vision*.

2.2. Aplicaciones sin Deep Learning

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

1. HAAR-like Features & ADABOOST.

Técnica diseñada por Paul Viola y Michael Jones en el año 2001, donde se implementa una detección mediante el uso de *Machine Learning* que es capaz de procesar imágenes a una velocidad y precisión bastante altos, 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* [34].

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. Estos servirán para el entrenamiento de un clasificador SVM (*Super Vector Machine*), que será capaz de realizar una detección en tiempo real.

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. 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* [20]. 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.

2.3. Aplicaciones con Deep Learning

A partir de 2016, el *Deep Learning* fue un ámbito que ganó mucha importancia en la visión artificial. Basado en 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. Este modelo es entrenado mediante imágenes completas [27].

2. TensorFlow.

TensorFlow es una plataforma de *Open Source* dedicada al aprendizaje automático. 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*, capaz de obtener un gran rendimiento en dispositivos móviles y ordenadores con poca potencia computacional. Esto ha creado un aumento de la popularidad en la implementaciones de detectores en dispositivos embebidos, como *Raspberry Pi*, o en chips ESP32

mediante procesamiento remoto.

3. MediaPipe.

MediaPipe [24] es un framework creado por Google centrado en el desarrollo de aplicaciones de visión artificial de forma sencilla y potente. Este se basa en un modelo propio llamado BlazeFace, inspirado en modelos como MobileNet. Entre sus soluciones (Figura 2.1) se pueden encontrar: detección facial, predicción de una malla facial (Face Landmarks), detector de iris, detector de manos, reconocer poses, segmentación de pelo, etc.

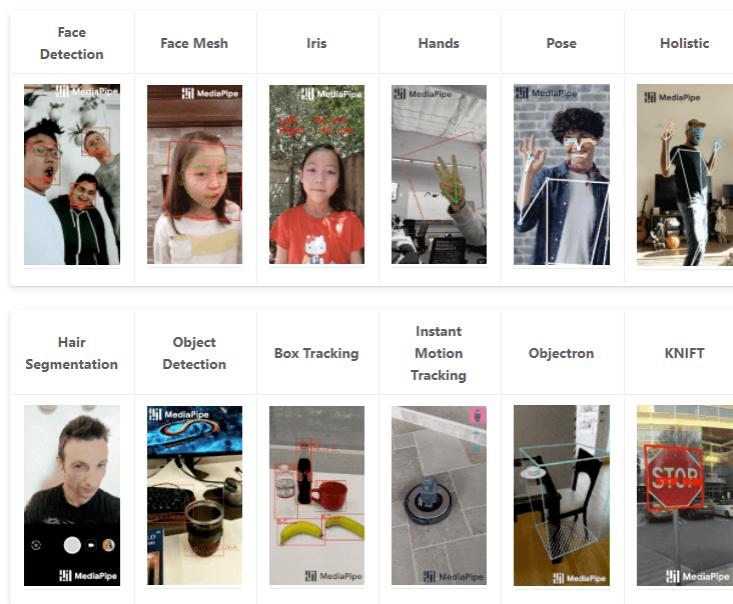


Figura 2.1: Soluciones de MediaPipe [15].

4. IBM Watson.

IBM Watson es un servicio ofrecido por IBM, 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 modulo llamado *Visual Recognition* que permite el análisis de imágenes y detección de objetos. Este tipo de servicios son de pago, pero permiten una cantidad de procesado gratuita. En este caso, IBM permite tener dos modelos personalizados y 1000 eventos de forma gratuita al mes [18].

CAPÍTULO 3

Análisis de objetivos y metodología

El COVID-19 es un virus que ha provocado en la humanidad incontables problemas, y en el mundo de la visión artificial también, por eso me voy a centrar 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, detectar cuando una persona lleva, de manera correcta, una mascarilla al entrar a un comercio, cine, restaurante, etc. Los objetivos que se plantean para llevar esto a cabo son los siguientes:

- Estudio de las tecnologías actuales, para comprobar su comportamiento con uso de mascarilla.
- Creación de un prototipo capaz de reconocer rostros y detectar si se lleva mascarilla.
- Estudiar la capacidad de que el prototipo pueda identificar si se lleva correctamente la mascarilla.
- Poder detectar la mascarilla, independientemente del tipo que se lleve.

3.1. Prototipo

Para cada uno de los apartados del desarrollo de este *TFG* se creará un prototipo con la finalidad de mostrar la tecnología expuesta en el mismo. Con el objetivo final de crear una aplicación que contenga todos los prototipos y se puedan ejecutar de forma sencilla.

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

- Detección de mascarillas a una distancia cercana.
- Detección de mascarillas a una distancia media.
- Detección de mascarillas desde una posición alejada, como la parte superior de una puerta.

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:

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

Cuadro 3.1: Entornos de prueba.

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.

- Dlib: 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* y *Facial Landmark*.
- 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.
- Numpy: Librería que añade funcionalidad a *arrays* multidimensionales y matrices, junto a un gran conjunto de operaciones matemáticas para trabajar con ellos.
- Mediapipe: Librería con soluciones y aplicaciones de *Machine Learning* para dispositivos móviles, en la nube o web.
- Tensorflow: Librería de *Machine Learning*, referente a el entrenamiento y utilización de redes convolucionales (*Deep Learning*).

Fases de desarrollo

El proceso de implementación del prototipo en cada una de las técnicas presentadas en este TFG, van a contar con 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 *paper*, artículo oficial donde se presenta la técnica por sus autores. Y, posteriormente buscar información extra en libros u 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. Implementacion 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.

CAPÍTULO 4

Diseño y resolución

4.1. Paul Viola and Michael Jones

En 2001, el reconocimiento facial tuvo su primera aparición en el campo de la visión artificial como aplicación en tiempo real. Este avance fue de la mano de Paul Viola y Michael Jones. Análogamente, 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 realizará una implementación del mismo mediante *Python* y *OpenCV* para comprobar como se comporta en la situación actual.

Método de estudio

El trabajo de los expertos fue presentado por parte de la Universidad de Cambridge mediante un *paper* (ensayo de la investigación). 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"[34]

Para poder lograr esta afirmación 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 prototipo 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 *grayscale*. Con el objeto de encontrar características en ellas, llamadas *haar-like features*. Nombradas así por su inventor Alfred Haar en el siglo XIX. En este trabajo se hacen uso de tres tipos de haar-like features (Figura 4.1).

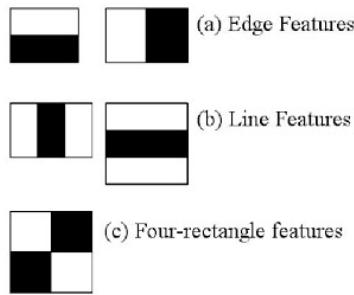


Figura 4.1: Haar-like Features [26].

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 Features* serán las características de la detección facial.

En un estudio ideal, los píxeles que forma el *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 gradiente, 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 [26].

Estas características van a ayudar al ordenador a entender lo que es la imagen estudiada. Van a ser utilizadas mediante *Machine Learning* para detectar donde 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 que la representa 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 [2]. 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):

$$\text{sum} = L4 + L1 - (L2 + L3)$$

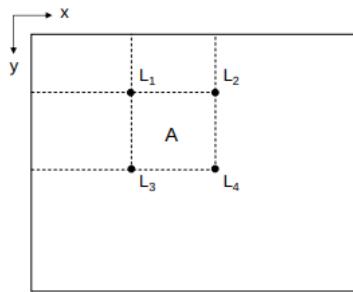


Figura 4.2: Funcionamiento de una *Imagen Integral* [2].

Viola y Jones junta esta propuesta 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 esta basado en generar una predicción muy buena a partir de la combinación de predicciones peores y más débiles, donde cada uno de estas se corresponde con el *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 [22]. (Como se muestra en la *Figura 4.3*)

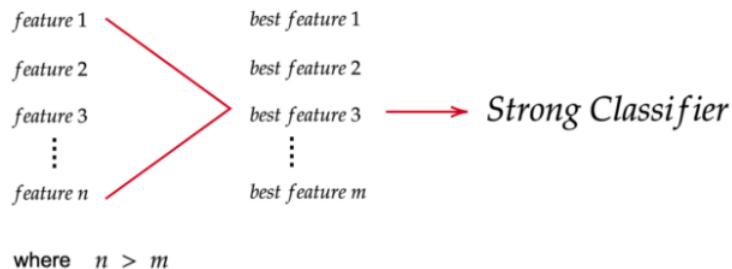


Figura 4.3: Construcción del *Strong Classifier* [22].

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 esta compuesta por varios estados de *Strong Classifiers* generados por el algoritmo *AdaBoost*. Donde 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 tal forma que, cuantos más estados atraviese 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*.

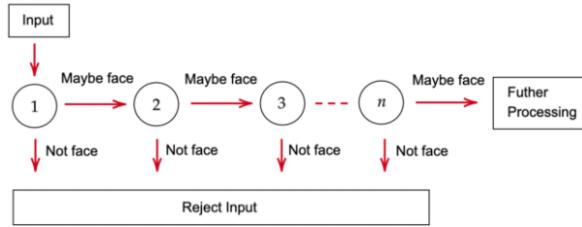


Figura 4.4: Construcción del *Multi-stage Classifier* [22].

Implementación y Experimentación

El prototipo será implementado en *Python*, con el uso de *OpenCV*. Y, el objetivo es construir dos detectores de caras, donde el primero usará dos modelos preentrenados de *OpenCV* de caras frontales, para estudiar cual 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 preentrenados y uno de *Machine learning* se pueda detectar una cara con una mascarilla o sin ella.

La **implementación básica** hace uso de un modelo preentrenado cargado mediante una clase de *OpenCV*, llamada *Cascade Classifier*. Esta representa la base de *Machine Learning* explicado en el apartado anterior. Asimismo, *OpenCV* también proporciona una serie de archivos *xml* con diferente modelos preentrenados. 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 blanco y negro, 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 esta prototipo se realizará a una distancia corta (70 cm) y media (150 cm), para comprobar su funcionamiento estandar con dos modelos pre-entrenados, llamados: *frontalface_default.xml* y *frontalface_alt2.xml*, dedicados a reconocer rostros de manera frontal. Intentando estudiar su velocidad de procesamiento y su precisión a la hora de detectar un rostro sin y con obstáculos, el resultado obtenido se ve reflejado en la Figura 4.5.

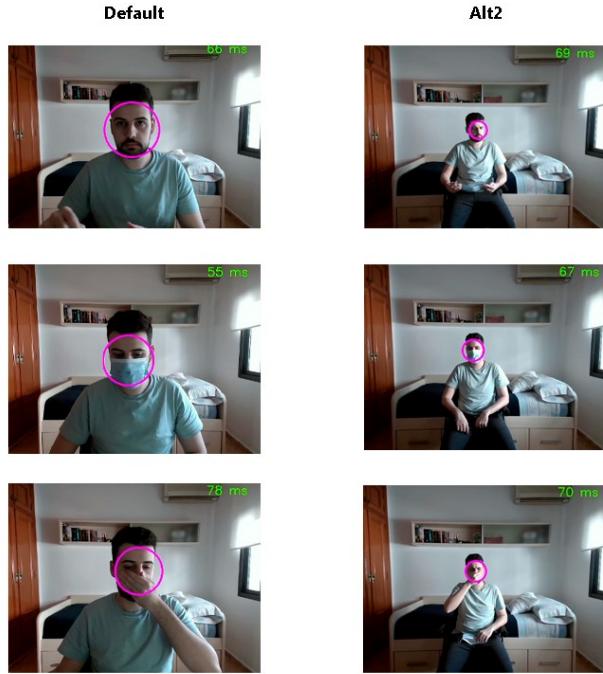


Figura 4.5: Pruebas con Haar-like features con: *frontalface_default.xml* y *frontalface_alt2.xml*

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 recogidos 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 siguiente tabla:

	default (70cm)	default (150cm)	alt2 (70cm)	alt2 (150cm)
PC1 / Time (MS)	46.15	45.09	62.04	61.75
PC2 / Time (MS)	X	X	X	X

Cuadro 4.1: Tiempos para el prototipo 1.

El **segundo prototipo** (custom) implementa un identificador de caras conjunto a un modelo de *Machine Learning* que identifica cuando una persona lleva o no mascarilla, además de si se lleva bien. 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 que caso se trata. Previo a dicha predicción y ejecución del prototipo, se necesita entrenar el modelo. Y para ello se toman referencias de rostros con y sin la mascarilla para que sirvan como entrada en el entrenamiento del modelo, usando el prototipo anterior. Exactamente se debe ejecutar dos veces el archivo Python: *trainCascade.py*, una para el caso de sin y otro para el caso de con mascarilla. Siendo la mejor opción, el uso del modelo pre-entrenado *frontalface_alt2.xml*. También es posible realizar el entrenamiento con imágenes de un dataset.

Principal Component Analysis, (**PCA**), es un algoritmo de *Machine Learning* dedicado a la reducibilidad 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 conocido como una regresión lineal [16]. La salida de PCA servirá para reducir la complejidad de la entrada al modelo **SVM** (Support Vector Machine), 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) [16].

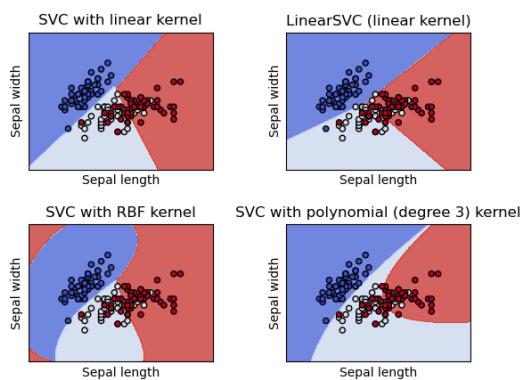


Figura 4.6: Ejemplos de clasificaciones SVM

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 (donde se encuentra dicho 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

La desventaja es que solamente funciona con los rostros/rostro que se toma como referencia para construir el modelo, igualmente pasa con el tipo de mascarilla (siendo la quirúrgica la que mejor funciona con este prototipo). Asimismo, su funcionamiento es de manera frontal y cercana, tanto a 70 cm como a 150 cm, 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. Asimismo, el tiempo de retardo que presenta este prototipo para el PC1 es de una media de 87.5 ms.

Este procedimiento se podría llegar a usar con otra implementación, específicamente con HOG, centrada también en la extracción de características pero usando 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 cara dentro de la imagen (En este caso, se usará *Haar-like features*) y obtener dichos puntos de interés. La implementación que se va a utilizar es la estudiada por *Kazemi* y *Sullivan* en 2014, con el paper *One Millisecond Face Alignment with an Ensemble of Regression Trees* [21], 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, etiquetados 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. El cual, 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. Mientras que 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 [25]:

1. *Pre-procesado*: Para una imagen de entrada de cualquier tamaño es tratada en regiones de ciertas escalas y analizadas en varias zonas de la imagen. La única restricción es que los tamaño 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 de las direcciones de dichos gradientes con el uso de la siguiente fórmula:

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

$$\theta = \arctan \frac{g_y}{g_x}$$

Figura 4.8: Fórmula HOG [25].

La fórmula esta 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 como en todos los píxeles el gradiente tomarán 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 influencia al ruido. El tamaño de la celda vendrá definida 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 ídem. 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 8x8, un bloque de 16x16 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 preentrenado.

Sullivan Paper

El *paper* 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. Esta basado en *gradient boosting* para el aprendizaje de un conjunto de arboles de regresión (*ensemble of regression trees*), que será el encargado de la predicción de los puntos de interés [20].

Este método fue uno de los primeros que mejoró el rendimiento, a diferencia con los métodos anteriores, gracias a la detección de componentes esenciales para el *face alignment* 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* [28], 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

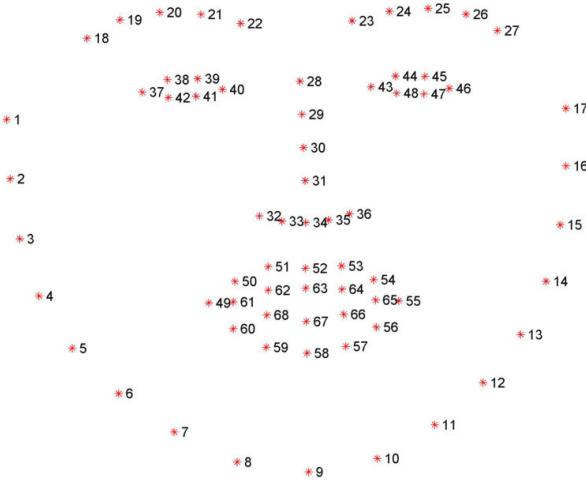


Figura 4.9: 68 coordenadas del *facial landmark* con *iBUG 300-W dataset* [28].

es que necesitamos características confiables para obtener una predicción con precisión la forma y, por otro lado, necesitamos una estimación precisa de la forma para extraer 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 las predicciones estimadas en zonas lineales del subespacio mienten, lo que concluyó siendo de ayuda para evitar este problema. Pero, posteriormente se descubrió una segunda solución, donde se asume que la función miente en zonas lineales del subespacio pero no es necesario realizar trabajo adicional. En el caso de este *paper*, 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 perdida 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 [20].

De forma más general, la investigación ofrece las siguiente 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 realizará el uso de los modelos propuesto 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 donde se encuentra el rostro en la imagen. Para realizar la detección tiene que recibir una imagen transformada en el espacio de color *GRAY*.
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 basandonos en el prototipo anterior de *Haar-like features* se construye un detector

capaz de detectar una boca. Con el objetivo de que si se detecta una boca se puede decir que la persona no lleva mascarilla, mientras que, si no es capaz de realizar una detección si que llevaría mascarilla.

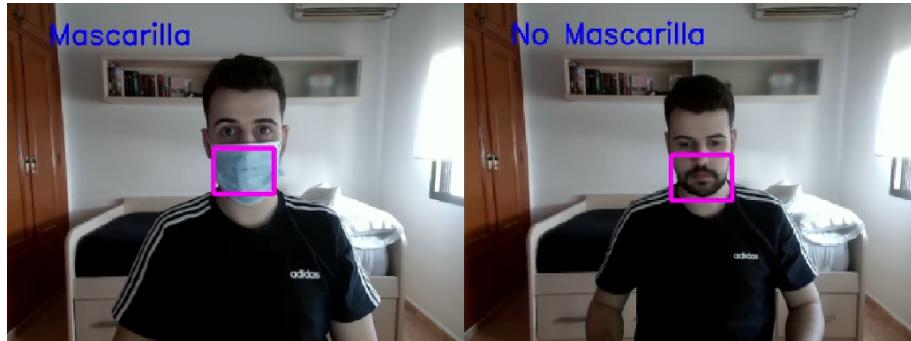


Figura 4.10: Pruebas con Dlib's Facial Landmarks y modelo Haar-like feature *mcs_mouth.xml*.

El tiempo de retardo que presenta este prototipo para el PC1 es de una media de 5 ms. Aunque estos tiempos presenten un buen resultado, el funcionamiento del prototipo 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. 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 mas usados para la creación de aplicaciones de visión artificial. [30]. Esta basado en las estructuras neuronales 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, los conocidos *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. 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 [13].

Todo esto es implementado gracias al framework *MediaPipe*, con la herramienta *MediaPipe graph*. 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* [24].

El *pipeline* puede ser definido mediante la adición/modificación de *calculators* dentro del *graph*. Específicamente, el *pipeline* que se utiliza en esta solución (*FaceMesh*) esta formada por un *graph* compuesto por un *subgraph* de *face landmark* (proveniente del módulo, ya implementado, de *face landmark* de *Mediapipe*), donde 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 [13]. En concreto el *graph* que se usa en esta implementación es el mostrado en la Figura 4.11.

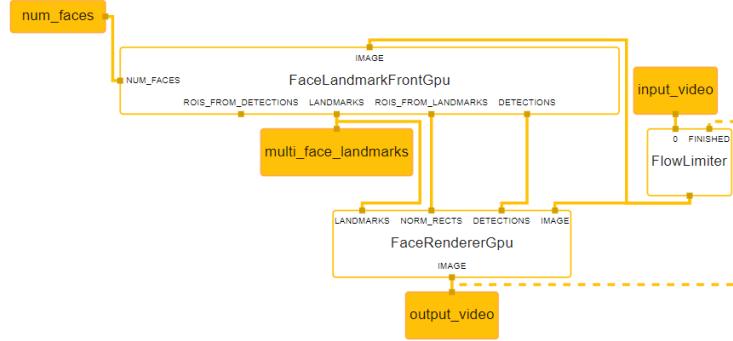


Figura 4.11: MediaPipe Graph utilizado en FaceMesh [14].

Por tanto, el procesamiento de una imagen en este modelo sigue dos pasos. El primero (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 en un modelo completo, hasta 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 [19].

Las coordenadas que se obtienen como salida están compuestas por unas coordenadas 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 masa 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). Modelo *lightweight* creado para GPU móviles, llegando a una velocidad de procesamiento entre 200 a 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. [3]

El segundo modelo, *Face Landmark Model*, generado mediante *Transfer Learning* buscando los siguientes objetivos: crear coordenadas 3D (mencionadas anteriormente) y conseguir mostrarlas en la imagen de salida de forma correcta [13]. 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 [33].

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. (Figura 4.12) [1].

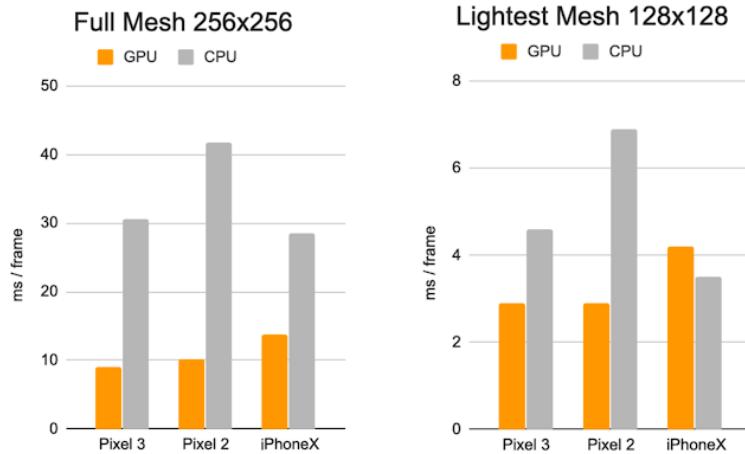


Figura 4.12: Rendimiento de FaceMesh sobre dispositivos móviles [1].

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 (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 a ..., mientras que el segundo se entiende como

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 desventaja 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 si porta una mascarilla, cuando en realidad no es así. Lo bueno es que no hace falta distinguir entre tipos mascarillas.

El tiempo de retardo que presenta este prototipo para el PC1 es de una media de 15.5 ms. Tras la realización de las pruebas, se puede concluir en que el prototipo funciona



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

de una manera bastante eficaz y precisa en distancias cercanas, mientras que en una posición elevada, como podría ser una puerta, tarda en detectar a la persona, hasta que esta no se encuentre a una distancia cercana de la misma.

4.4. Tensorflow

TensorFlow es una plataforma de *Open Source* dedicada al aprendizaje automático. Permite compilar e implementar con facilidad aplicaciones con tecnología de AA. Esta se basa en tensores, matrices multidimensionales con un tipo uniforme, que 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 [31].

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* [32], 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 de lo que estamos buscando, comprobar si la mascarilla esta bien vestida.

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. Algo bastante revolucionario, puesto que todos los modelos hasta ahora necesitaban millones de datos, clasificados a mano y posteriormente necesitar una gran capacidad computacional para ser entrenados [33]. 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 extracto de características. Normalmente, se implementa con modelos como *ResNet*. Mientras que, la cabeza *SSD* está formada por uno o mas capas convolucionales que interpretan la salida del backbone como *bounding boxes* y clases de objetos [8].

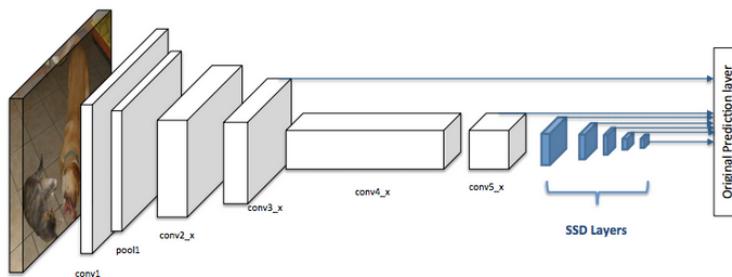


Figura 4.14: Arquitectura de una CNN con uso de SSD [8].

Otras características destacables de *SSD* es 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 paper con su mismo nombre, "SSD: Single Shot MultiBox Detector" [23]. 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. Asimismo, 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 conjunto 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:

- Coco mAP (*main Average Precision*): indica la precisión media del modelo.
- Speed: 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, he realizado una serie de fotos con varias mascarillas y sin (todas estas imágenes son tomadas con la webcam que se va a realizar el trabajo) mezcladas con imágenes de ejemplo encontradas por internet. Consta de un total de X imágenes, divididas en dos grupos train y test. En el primero se encuentran las imágenes que se utilizarán para el entrenamiento del modelo por *Transfer Learning* y cuenta con X imágenes. Mientras que, el conjunto de test se utilizará para validar el modelo entrenado con un total de X imágenes.

Para construir un dataset más completo se utiliza una técnica llamada *Data Augmentation*, que genera modificaciones de las imágenes del modelo, como blur, rotaciones, etc.

Antes de realizar el modelo es necesario preparar las imágenes conjuntamente a un archivo donde se plasmen las etiquetas del modelo. 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 dos labels del trabajo: *mask* o *noMask*. Una vez creados todos los archivos de etiquetado *xml* con los labels, se tiene que crear un archivo *pbtxt* donde se refleje 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. Depende 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 manera que, 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*.

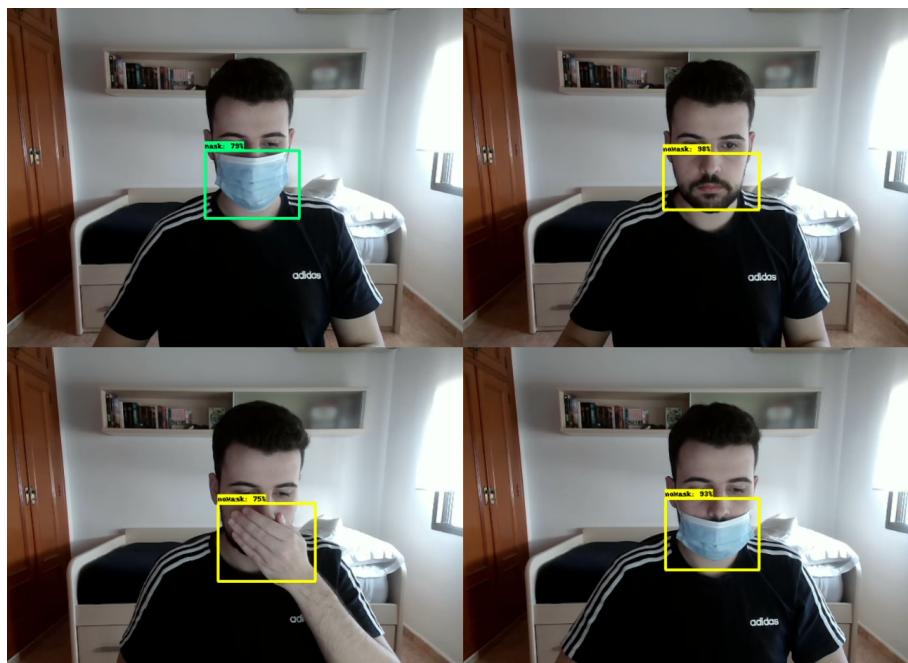


Figura 4.15: Pruebas con SSD-MobileNetV2 *mcs_mouth.xml*

El tiempo de retardo que presenta este prototipo para el PC1 es de una media de 83.2 ms. Aunque el resultado obtenido sea bastante bueno, 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, al igual que los anteriores, funciona de forma correcta y precisa en pruebas cercanas a la cámara, mientras que en distancias mayores se dificulta la detección. De igual forma, si se encuentra en una situación donde la iluminación sea escasa o la cámara no esté bien colocada el rendimiento del prototipo disminuye.

4.5. Comparación

CAPÍTULO 5

Conclusiones y vías futuras

Finalizamos el trabajo estableciendo las conclusiones y vías futuras...

Bibliografía

- [1] Arsiom 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] AIShak. Integral images in opencv. <https://aishack.in/tutorials/integral-images-opencv/>, Jun 2010. Acceso: 07-04-2021.
- [3] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. Blazeface: Sub-millisecond neural face detection on mobile gpus. 07 2019.
- [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] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Pearson, 2018.
- [13] Google. Mediapipe face mesh. 2020. Acceso: 21-04-2021. URL: https://github.com/google/mediapipe/solutions/face_mesh.
- [14] Google. Mediapipe visualizer, 2021. Acceso: 20-05-2021. URL: <https://viz.mediapipe.dev/>.
- [15] Google. ML solutions in mediapipe, 2021. Acceso: 19-05-2021. URL: <https://github.com/google/mediapipe/>.
- [16] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly, 2020.
- [17] 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.
- [18] IBM. Visual recognition, 2021. Acceso: 07-05-2021. URL: <https://cloud.ibm.com/catalog/services/visual-recognition>.
- [19] Yury Kartynnik, Arsiom Ablavatski, Ivan Grishchenko, and Matthias Grundmann. Real-time facial surface geometry from monocular video on mobile gpus. 07 2019.
- [20] 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.
- [21] Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. 06 2014. doi:10.13140/2.1.1212.2243.
- [22] 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-c55cc>
- [23] 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.
- [24] Camillo Lugaresi, 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.
- [25] Satya Mallick. Histogram of oriented gradients explained using opencv. 2016. Acceso: 02-05-2021. URL: <https://learnopencv.com/histogram-of-oriented-gradients/>.

- [26] 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.
- [27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016. arXiv:1506.02640.
- [28] 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.
- [29] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. arXiv: 1801.04381.
- [30] R. Szeliski. *COMPUTER VISION: Algorithms and applications*. SPRINGER NATURE, 2021.
- [31] TensorFlow. Tensorflow core, 2021. Acceso: 02-05-2021. URL: https://www.tensorflow.org/guide?hl=es_419.
- [32] Tensorflow. Tensorflow official models, 2021. Acceso: 02-05-2021. URL: <https://github.com/tensorflow/models/tree/master/official>.
- [33] TensorFlow. Transferir el aprendizaje y la puesta a punto, 2021. Acceso: 25-04-2021. URL: https://www.tensorflow.org/tutorials/images/transfer_learning.
- [34] 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.