Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Reconocimiento facial para la autentificación de usuarios

Trabajo Fin de Grado
**Grado en Ingeniería Informática**

**Autor**: Maria Martínez Guerrero
**Tutor**: Cèsar Ferri Ramírez
**Tutor Externo**: Carlos Asensio Pizarro
2017/2018

# Agradecimientos

A Dani, por no dejarme tirar la toalla en los últimos momentos.

A mi familia, por el apoyo incondicional durante estos años.

A mi familia universitaria, por tanto.

# Resum

El desenvolupament d'aquest projecte consisteix a crear una eina que, mitjançant el reconeixement facial, permeta l'autenticació de l'usuari a través d'imatges preses en temps real.

Amb aquesta fi, es realitzaran unes preses d'imatges de l'usuari mitjançant una camara web i del document d'identitat de l'usuari que seran les mostres a analitzar en el procés de validació. D'una banda, la presa d'imatge de l'usuari es realitzarà a través de l'algorisme Cascade Haar Classifier per detectar el rostre. D'altra banda, en les preses d'imatges del document serà necessari recórrer a diverses tecnologies com a detectors i descriptores d'imatges i el reconeixement òptic de caràcters amb la finalitat d'extreure informació del document d'identitat. Per a la realització d'aquesta primera part comentada s'utilitzarà la llibreria OpenCV.

Finalment es pretén crear i entrenar una xarxa neuronal convolucional a la recerca de la validació de l'usuari a través de la imatge presa en temps real i la imatge extreta de la mostra presa del document d'identitat. En aquesta part ens basem en la implementació de la xarxa neuronal convolucional del model VGG i per a la seva execució utilitzarem el marc de Caffe.

**Paraules clau:** OpenCv, Xarxa Neuronal Convolucional, Reconeixement Òptic de Caràcters, Classificadors en cascada Haar, Caffe.

# Resumen

El desarrollo de este proyecto consiste en crear una herramienta que, mediante el reconocimiento facial, permita la autentificación del usuario a través de imágenes tomadas en tiempo real.

Con ese fin, se realizarán unas tomas de imágenes del usuario mediante la webcam y del documento de identificación del usuario que serán las muestras a analizar en el proceso de validación. Por una parte, la toma de imagen del usuario se realizará a través del algoritmo *Cascade Haar Classifier* para detectar el rostro. Por otra parte, en las tomas de imágenes del documento será necesario recurrir a diversas tecnologías como detectores y descriptores de imágenes y el reconocimiento óptico de caracteres con el fin de extraer información del documento de identificación. Para la realización de esta primera parte comentada se utilizará la librería OpenCV.

Finalmente se pretende crear y entrenar una red neuronal convolucional en busca de la validación del usuario a través de la imagen tomada en tiempo real y la imagen extraída de la muestra tomada del documento de identificación. En esta parte nos basamos en la implementación de la red neuronal convolucional del modelo VGG y para su ejecución utilizaremos el marco de Caffe.

**Palabras clave:** OpenCv, Red Neuronal Convolucional, Reconocimiento Óptico de Caracteres, Clasificadores en cascada Haar, Caffe.

# Abstract

The development of this project consists of creating a tool that, through facial recognition, allows user authentication through images taken in real time.

To perform this, some images of the user and their user identification document will be taken through a webcam. The user identification document pictures will be the samples to be analyzed in the validation process. The user's image capture will be done through the Cascade Haar Classifier algorithm to detect the face. On the other hand, to take pictures of the document it will be necessary to use various technologies such as detectors and image descriptors and optical character recognition in order to extract information from the identification document. For the realization of this first part, the OpenCV library will be used.

Finally, it is intended to create and train a convolutional neuronal network to validate of the user through the image taken in real time and the image extracted from the sample taken from the identification document. In this part we rely on the implementation of the convolutional neuronal network of the VGG model and for this implementation we will use the Caffe framework.

**Keywords:** OpenCv, Neuronal Convolutional Network, Optical Character Recognition, Cascade Classifiers Haar, Caffe.

# Tabla de contenidos

# Tabla de figuras

# 1. Introducción

## 1.1 Motivación

La revolución digital sufrida en la sociedad en los últimos años ha desencadenado en una proliferación de plataformas digitales de índoles diversas. Esto ha provocado que la sociedad haya dejado de ser reacia a la tecnología, y demande cada vez más soluciones que implican la digitalización de acciones cotidianas, siendo ésta una necesidad para las empresas del siglo XXI.

Este desembarco tecnológico ha llegado a sectores tradicionales y altamente regulados, como el sector bancario, generando múltiples retos. En este sector, la necesidad de autentificar la identidad de los clientes, sin necesidad de forzarlos a acudir físicamente a las oficinas, es uno de los muchos retos que han de ser superados para que las compañías puedan ofrecer sus servicios de forma cien por cien digitalizada.

Los procesos convencionales de autenticación del usuario y su documentación usados por la mayoría de estas plataformas bancarias suelen resultar en la no adquisición de potenciales clientes. Esto es debido a que tienden a ser insatisfactorios para el usuario digital, muy acostumbrado a la instantaneidad, y que se ve obligado a esperar incluso semanas para abrir una cuenta, provocando su desinterés. Además, dichos procesos suelen tener unos costes elevados para las plataformas.

Por ello, debido a esta problemática real del mundo digital, se propone la producción de un sistema de validación de usuarios para plataformas digitales que deban cumplir con elevados niveles de verificación. De esta forma dispondrán de un sistema automático que les permita dinamizar dicha validación al permitir a los usuarios la integración inmediata en la plataforma, reduciendo los tiempos de respuesta y costes a nivel de empresa.

## 1.2 Objetivos

El principal objetivo de este Trabajo de Final de Grado es el desarrollo de un sistema de verificación de identidad por medio de la utilización de patrones faciales obtenidos de imágenes tomadas a través de una webcam y el propio documento de identificación del usuario.

Para poder llevar a cabo este objetivo final serán necesarios llevar a cabo los siguientes subjetivos:

1. Extracción de la imagen del usuario: Se obtendrá la imagen del rostro del usuario a partir de la imagen de la webcam y de clasificadores en cascada de *Haar*.
2. Detección y normalización del documento de identificación: La detección se realizará mediante el uso de puntos característicos. Estos servirán localizar el documento de identidad dentro de la imagen obtenida por una webcam y para proyectar las imágenes sobre un espacio normalizado del que extraer la información.

3. Extracción de los datos del documento de identificación: Una vez normalizado se aplicarán diversos filtros de convolución y programas de reconocimiento de caracteres para obtener la información del documento. Así mismo, se obtendrá la fotografía del documento de identificación.
4. Validación del rostro del usuario con la fotografía del documento de identificación: Una vez obtenidas ambas imágenes estas serán comparadas mediante sistemas de clasificación basados en redes convolucionales profundas, proporcionando el resultado de la validación del usuario.

# 2. Estado del arte

## 2.1 OpenCV

OpenCV es una librería software de visión artificial y sistemas de aprendizaje automático siendo diseñado para conseguir una eficiencia computacional elevada y enfocada en aplicaciones de tiempo real desarrollada nativamente en C/C++ y bajo una licencia BSD (*Berkeley Software Distribution*). Esta licencia permite el uso y modificación del código tanto para fines comerciales como académicos. Actualmente dispone de interfaces de desarrollo o API (*Application Program Interface*) para C++, C, Python Java y MATLAB siendo compatible con Linux, Mac OS X y Windows, así como también con dispositivos móviles Android y iOS.


*Figura 1: Logotipo OpenCV*

La librería ofrece más de 2500 algoritmos que han sido optimizados llegando a abarcar un conjunto completo de algoritmos en el campo de la visión artificial y del aprendizaje automático. Estos algoritmos que nos proporciona pueden ser usados para detectar y reconocer rostros, identificación de objetos, detectar y clasificar las acciones humanas en videos, el seguimiento de objetos en movimiento, crear composiciones de imágenes de 360 grados de alta revolución, eliminación de los rojos ojos en imágenes o seguir los movimientos oculares, entre un largo etc.

El gran elevado número de algoritmos de los cuales dispone distribuidos de forma modular han conseguido presentar un entorno de desarrollo fácil de utilizar y de alta eficiencia, facilitando al usuario la construcción de aplicaciones sofisticadas con rapidez. Hoy en día es empleado por miles de usuarios entre los cuales podemos encontrar grupos de investigación, instituciones gubernamentales y grandes compañías. Algunas de estas, son empresas como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda o Toyota. La robustez que nos presenta OpenCV da origen a potentes aplicativos en diversos campos, ya sea evitar accidentes por ahogamiento en

piscinas europeas, ser capaz de ejecutar arte interactivo en España y Nueva York, la composición de las imágenes de *Street Views* (servicios en internet desde los que se pueden ver recorridos virtuales de calles o carreteras) o la detección de intrusos en canales de video vigilancia en Israel.

A continuación, procedemos a describir algunos de los módulos que están integrados en OpenCV, así como también describiremos algunas de sus funciones y estructuras de datos, que han sido utilizados en la realización de este proyecto.

### 2.1.1 Módulo Core

Este módulo define las estructuras de datos básicos, entre ellos los arreglos multidimensionales Mat, así como las funciones básicas utilizadas por el resto de módulos

#### *2.1.1.1 Class Mat*

La clase Mat representa una matriz n-dimensional densa de un solo canal o multicanal. Se puede usar para almacenar vectores y matrices con valores reales o complejos, imágenes en escala de grises o colores, volúmenes de píxeles, campos de vectores, nubes de puntos e histogramas. La existencia de este tipo de datos en OpenCV facilita el trabajo con imágenes, así como la facilidad de recordar secciones de interés de estas imágenes.

### 2.1.2 Features 2D

Este módulo permite el uso de detectores de características, descriptores y comparadores descriptores.

### 2.1.3 Módulo Highgui

Este módulo se desarrolla por la necesidad de poder visualizar y operar sobre imágenes de forma rápida y sencilla sin la necesidad de disponer de un marco de interfaz de usuario.

#### *2.1.3.1 Función imshow*

Permite visualizar una imagen en una ventana y su declaración es:

```
void imshow( const string& winname, InputArray Mat )
```

donde:

- **winname**, nombre de la ventana.
- **Mat**, imagen a mostrar.

#### *2.1.3.2 Función imread*

Permite realizar la carga de una imagen desde archivo siendo su declaración:

```
Mat imread ( cadena const & filename, int flags = 1 )
```

donde:

- **filename**, ruta del archivo.
- **flag**, especifica el tipo de color en que se cargará la imagen. Puede tomar los siguientes valores:

- CV_LOAD_IMAGE_ANYDEPTH, carga la imagen en 16-bit/32-bit cuando la entrada tenga la profundidad correspondiente, de lo contrario la convierte en 8-bit.
- CV_LOAD_IMAGE_COLOR, carga la imagen a color.
- CV_LOAD_IMAGE_GRAYSCALE, carga la imagen en escala de grises.
- >0, carga la imagen en color de 3 canales.
- =0, carga imagen en escala de grises.
- <0, carga imagen en su estado original.

#### *2.1.3.3 Función imwrite*

Permite guardar un fichero especificado y su función viene dada por:

```
bool imwrite( const string & filename , InputArray img ,
    vector const <int> & params = vector <int> () )
```

donde:
- **filename**, nombre o ruta del fichero a guardar.
- **img**, imagen a guardar.
- **params**, codificados a pares y pueden tomar los siguientes valores:
  - CV_IMWRITE_JPEG_QUALITY, puede ser una calidad y por defecto tiene el valor 95. Puede tomar un valor entre 0 y 100 (cuando más grande mejor).
  - CV_IMWRITE_PNG_COMPRESSION, puede ser el nivel de compresión y por defecto tiene el valor de 3. Puede tomar un valor entre 0 y 9 (a valor más alto, tamaño más pequeño y mayor nivel de compresión).
  - CV_IMWRITE_PXM_BINARY, puede ser un indicador de formato binario y su valor por defecto es 1. Puede tomar el valor de 0 o 1.

#### *2.1.3.4 Clase VideoCapture*

Esta clase proporciona la API para capturar video de cámaras o para leer archivos de video y secuencias de imágenes.

##### 2.1.3.4.1 Función open

Esta función abre un archivo de video o un dispositivo de captura para capturar video. En el proyecto se utilizará la implementación que abre un dispositivo que viene dada por:

```
bool VideoCapture::open(int device)
```

donde **device** toma el valor de identificación del dispositivo de captura de video, es decir, un índice de cámara.

##### 2.1.3.4.2 Funcion isOpened

Esta función comprueba que la cámara se ha inicializado de forma correcta y devuelve true en caso positivo siendo su función:

```
bool VideoCapture::isOpened()
```

#### 2.1.3.4.3 Funcion grab

Esta función toma el siguiente fotograma (*frame*) del archivo de video o del dispositivo. En caso de realizar la operación con éxito devuelve true. La función es:

```
bool VideoCapture::grab()
```

#### 2.1.3.4.4 Función retrieve

Esta función decodifica y devuelve el fotograma de video capturado. siendo la función:

```
bool VideoCapture::retrieve(Mat& image, int channel=0)
```

donde **image** es el objeto donde guardará dicho fotograma.

### 2.1.4 Módulo Calib3d

Este módulo contiene algoritmos básicos para múltiples vistas, calibración de cámara, estimación de la postura de un objeto, algoritmos de correspondencia y elementos de reconstrucción 3D.

### 2.1.5 Módulo Nonfree

El módulo contiene algoritmos que pueden estar patentados en algunos países o que tienen otras limitaciones de uso.

### 2.1.6 Módulo Imgproc

Este es un módulo de procesamiento de imagen que incluye filtrado de imagen lineal y no lineal, transformaciones geométricas de imagen, la conversión de espacio de colores, histogramas entre otros.

### 2.1.7 Módulo Objdetect

Este módulo realiza la detección de objetos. En nuestro caso será utilizado para la detección de rostros con la implementación de clasificadores en cascada de *Haar*.

#### *2.1.7.1 Clase CascadeClassifier*

Esta clase proporciona la API para el clasificador en cascada.

#### 2.1.7.1 Función load

Esta función carga un clasificador de un archivo y su función es:

```
bool CascadeClassifier::load(const string& filename )
```

donde **filename** es el nombre del archivo desde el cual se realiza la carga el clasificador. Este archivo puede contener un viejo clasificador de *Haar* ya entrenado y solo puede ser cargado desde un archivo XML o YAML.

#### 2.1.7.2 Función detectMultiScale

Esta función de encarga de detectar objetos de diferentes tamaños en la imagen de entrada que se devuelven como una lista de rectángulos y se define como:

```
void CascadeClassifier::detectMultiScale(const Mat& image,
vector<Rect>& objects, double scaleFactor=1.1, int minNeighbors=
3, int flags=0, Size minSize=Size(), Size maxSize=Size())
```

donde:

- **image**, matriz del tipo CV_8U que contiene la imagen donde se van a detectar los objetos.
- **objects**, vector de rectángulos donde cada rectángulo contiene el objeto detectado.
- **scaleFactor**, especifica cuanto se reduce el tamaño de la imagen en cada escala de imagen.
- **minNeighbors**, especifica cuantos vecinos debe tener cada rectángulo candidato para conservarlo.
- **minSize**, tamaño mínimo posible del objeto.
- **maxSize**,, tamaño máximo posible del objeto.

## 2.2 Algoritmo Cascade Haar Classifier

El algoritmo *Cascade Haar Classifier*, también conocido como algoritmo de Viola-Jones, es una aportación de Paul Viola y Michel Jones al mundo computacional de la detección de objetos. Este algoritmo dispone de un bajo coste computacional permitiendo que sea empleado en ejecuciones a tiempo real. La motivación de este desarrollo se vio iniciada por el problema de la detección de caras, siendo este uno de los campos donde más se emplea, pero podrá aplicarse a otras clases de objetos que estén caracterizados por patrones típicos de iluminación.

El algoritmo se basa en una serie de clasificadores débiles denominados *Haar-Like features* que se calculan de forma eficiente a partir de una imagen integral. Estos clasificadores se agrupan en cascada empleando un algoritmo de aprendizaje basado en *AdaBoost* para conseguir una capacidad discriminativa en las primeras etapas logrando así un alto rendimiento. *AdaBoost* es un meta-algoritmo adaptativo de aprendizaje automático cuyo nombre es una abreviatura de *adaptative boosting*.

En los siguientes puntos vamos a proceder a describir más profundamente este funcionamiento centrándonos en los puntos más importantes del estudio de Viola-Jones.

### 2.2.1 Haar-Like features

Los *Haar-Like features* son los elementos básicos que utiliza el algoritmo para realizar la detección. Estas *features* son características simples que se buscan en la imagen y que consisten en la diferencia de intensidades luminosas entre las regiones adyacentes y que se definen sobre una ventana de búsqueda básica de 24x24 píxeles. De esta manera, los *features* quedan definidos por unos rectángulos y su posición relativa a la ventana de búsqueda adquiriendo un valor numérico que es el resultado de la comparación que evalúan.

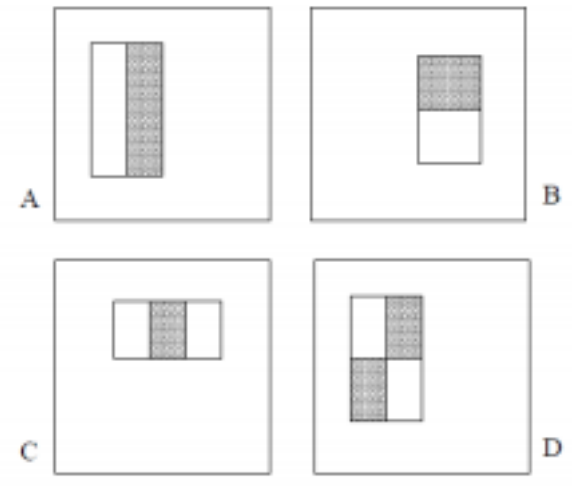


*Figura 2: Diferentes tipos de features definidos en el algoritmo de Viola-Jones Cascade Haar Clasifier.*

En el trabajo de Viola-Jones se definen 3 tipos de *features* representados en la figura 2:

- *Features* de dos rectángulos (Figura 2.A y 2.B) cuyo valor que toma es la diferencia entre las sumas de los pixeles contenidos en ambos rectángulos. Los rectángulos comparten forma y el tamaño de área, y son adyacentes entre ellos.
- *Features* de tres rectángulos (Figura 2.C) cuyo valor es la diferencia entre los rectángulos exteriores y el interior multiplicado por un peso para compensar la diferencia de área.
- *Features* de cuatro rectángulos (Figura 2.D) cuyo valor es la diferencia entre los pares agrupados por intensidades de luminosidad, es decir, pares diagonales.

### 2.2.2 Imagen integral

La suma de los píxeles de un rectángulo puede ser calculada de manera eficiente empleando una representación intermedia denominada imagen integral. La imagen integral en el punto (x, y) contiene la suma de todos los píxeles que están arriba y hacia la izquierda de ese punto en la imagen original.

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

donde $ii(x, y)$ es la imagen integral y $i(x, y)$ es la imagen original.

La imagen integral se puede calcular en un solo barrido de la imagen empleando el siguiente par de sentencias recurrentes:

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

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

donde $s(x, y)$ es la suma acumulada de la fila x, con $s(x, -1) = 0$ y $ii(-1, y) = 0$.

Usando la imagen integral, cualquier suma rectangular se puede calcular con cuatro referencias a memoria como se muestra en las sentencias anteriores. Las

*features* de dos rectángulos se pueden computar con 6 referencias a memoria puesto que comparten vértices. En el caso de *features* de tres rectángulos se pasa a 8, y a 9 para *features* de cuatro rectángulos.

### 2.2.3 Proceso de aprendizaje

Es necesario realizar un proceso de entrenamiento supervisado para crear la cascada de clasificadores. Este proceso se realiza mediante un algoritmo basado en AdaBoost. El *boosting* consiste en tomar una serie de clasificadores débiles y combinarlos para construir un clasificador fuerte con la precisión deseada.

Para seleccionar *features*, se entrenan clasificadores débiles limitados a usar una única feature. Para cada feature, el clasificador débil determina el valor umbral que minimiza los ejemplos mal clasificados. Un clasificador débil $h_j\ (x)$ por tanto consiste en una feature $f_j$, un valor umbral $\theta j$ y un coeficiente $pj$ indicando la dirección del signo de desigualdad.

$$h_j(x) = \begin{cases} 1 \ \ si\ p_j f_j\ (x) < p_j\ \theta_j \\ 0 \ \ e.o.c. \end{cases}$$

A continuación, se describe el algoritmo de AdaaBoost empleado. En cada ronda se selecciona un clasificador débil y por tanto una feature.

- Se parte de un conjunto de imágenes ($x_1$, $y_1$),…,( $x_n$, $y_n$) donde $y_i$ = 0, 1 para ejemplos negativos y positivos respectivamente.
- Se inicializan los pesos $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ para yi = 0, 1 respectivamente donde m es el número de negativos y l el número de positivos.
- Para cada ronda, $t$ = 1,…,T:
  1. Normalizar los pesos:
  $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
  2. Para cada *feature*, *j*, entrenar un clasificador $h_j$ que solo use una *feature*. El error se evalúa teniendo en cuenta los pesos:
  $$w_t\ , \epsilon_j = \sum_i\ w_i\ |h_j\ (x_i\ ) - y_i\ |$$
  3. Se escoge el clasificador, $h_t$ , con menor error $\epsilon_t$ .
  4. Se actualizan los pesos:
  $$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$$
  Donde $e_i$ = 0 si el ejemplo $x_i$ se clasifica correctamente y 1 en caso contrario y $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$
- El clasificador fuerte final es:

$$h(x) = \begin{cases} 1\ si\ \sum_{t=1}^{T} \alpha_t\ h_t\ (x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0\ e.o.c. \end{cases}$$

donde $\alpha t = 1\ \beta t$ .

### 2.2.4 Cascada

Viola-Jones construyen la cascada de clasificadores fuertes mediante la construcción de clasificadores más pequeños y eficientes que rechacen muchas ventanas negativas por no contener datos de interés y sólo en aquellas en las que las probabilidades de encontrar el objeto sean mayores se llama a las siguientes rondas de clasificación dentro de la cascada.

Los niveles de clasificación son cada vez más exigentes hasta que pueden confirmar la existencia del objeto que se está buscando. Este sistema de descarte dota de más rapidez al algoritmo.

## 2.3 Detectores y descriptores de imágenes

Una imagen está compuesta por un conjunto de pixeles, no obstante, estos conjuntos, cuando hablamos a nivel computacional, carecen de sentido por si mismos pues requieren una interpretación subjetiva de lo que simbolizan. Para dotar a las máquinas de esta interpretación se utilizan técnicas de extracción de puntos de interés que permiten extraer puntos de información relevante de una imagen mediante descriptores.

Estas técnicas son los detectores de puntos de interés y se encargaran de definir las características a bajo nivel, como esquinas o color, que tiene dicho punto para así poder localizarlo dentro de la imagen tras la aplicación de filtros.

Un descriptor busca extraer información de un área localizada de la imagen, como puede ser la textura alrededor de un punto de interés, y define un método de codificación que sea invariante a cambios de iluminación y transformaciones afines. Esto es debido a que las regiones se deforman con la imagen y su contenido no varía. Cada descriptor se estructura como un vector de características y se devuelven todos juntos como filas dentro de una misma matriz.

Con la adquisición de estos descriptores de ambas imágenes se podrá decidir qué puntos clave tienen un parecido y realizar un proceso de reconocimiento de objetos entre ellas. Este proceso de correspondencia entre los puntos de ambas imágenes toma el nombre de *Matching* y es necesario fijar un umbral que estas correspondencias deben cumplir para como considerarse un acierto.

Las ventajas de esta técnica es la utilización de solo una única imagen de prototipo evitando así la fase de entrenamiento masivo con una colección de imágenes.

Frecuentemente esta detección de puntos de interés y el cálculo de sus descriptores se realizan juntos, por lo que existe una variedad bastante amplia de detectores-descriptores que permiten realizar ambos pasos. A continuación, procedemos a exponer varios de estos métodos que se han estudiado para la realización de este proyecto.

### 2.3.1 SIFT

El algoritmo de visión artificial SIFT (*Scale Invariant Feature Transform*) propuesto por Lowe en 1999 permite detectar y posteriormente la descripción de características en regiones locales de una imagen que sean invariantes a la escala, orientación, iluminación, etc. [REF]

El algoritmo está estructurado en cuatro etapas que procedemos a describir en los siguientes puntos:

- Detección de máximos y mínimos en el espacio-escala.
- Localización de puntos de interés.
- Asignación a la orientación.
- Descriptores de los puntos de interés.

### 2.3.2 SURF

SURF es un detector y un descriptor de alto rendimiento de los puntos de interés de una imagen, donde se transforma la imagen en coordenadas, utilizando una técnica llamada multi-resolución. Consiste en hacer una réplica de la imagen original de forma Piramidal Gaussiana o Piramidal Laplaciana, y obtener imágenes del mismo tamaño, pero con el ancho de banda reducido. De esta manera se consigue un efecto de borrosidad sobre la imagen original, llamado Scale-Space. Esta técnica asegura que los puntos de interés son invariantes en el escalado. El algoritmo SURF está basado en el predecesor SIFT.

Otros detectores son GFTT y FLAN. Otros descriptores son FREAK, BRISK y BIEF.

## 2.4 Reconocimiento Óptico de Caracteres

El reconocimiento óptico de caracteres o OCR (*Optical Character Recognition*) es un proceso computacional dirigido a la digitalización de textos. Para ello, tal y como se observa en la Figura 2, la tecnología OCR requiere como entrada una imagen digital en blanco y negro, ya haya sido obtenida mediante el escaneo de algún documento o de una fotografía, que intentará transformarla en datos comprensibles y manejables para un ordenador obteniendo como resultado final un archivo en formato de texto.

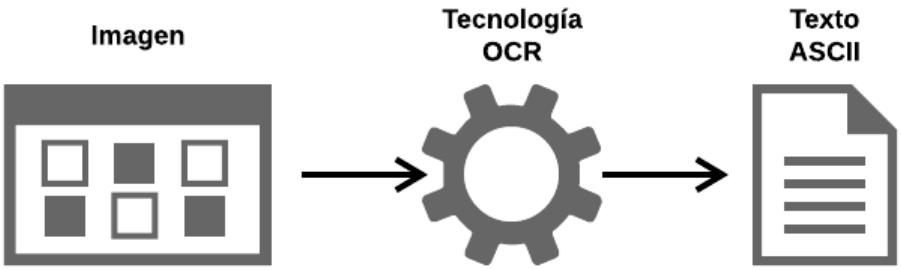


*Figura 3: Funcionamiento OCR*

Para hablar de dicho reconocimiento hay que tener en cuenta diversas variables. Por un lado, tenemos la calidad de la imagen, por otro el idioma del texto y finalmente si dicho texto proviene de un texto manuscrito o de un texto escrito por alguna máquina.

El reconocimiento exacto de la escritura latina se considera en gran parte un problema solucionado pues su exactitud excede el 99%, reduciendo así el requerimiento de una revisión humana para encontrar los errores. En cambio, todavía se sigue trabajando para conseguir el reconocimiento de otras lenguas especialmente aquellas que contienen un número muy grande de caracteres.

La diferencia que supone un texto manuscrito a uno escrito por alguna máquina, está haciendo que las exactitudes de los textos manuscritos solo lleguen a oscilar entre un

80-90%, suponiendo docenas de errores por cada página. Esto hace que esta tecnología en estos casos sea útil solamente en contextos muy limitados. Esta variedad del OCR ahora se conoce en la industria como Reconocimiento Inteligente de Caracteres (ICR por sus siglas del inglés) y están trabajando para mejorar su exactitud.

Otra variante que ha dado lugar el OCR es el Reconocimiento Óptico de Marcas (OMR por sus siglas en inglés). El reconocimiento de marcas sería por ejemplo la autocorrección de exámenes tipo test en los que la respuesta se rodea con un círculo. También ha dado lugar al Reconocimiento Automático de Matrículas (ANPR por sus siglas en inglés).

En los orígenes del OCR uno de los principales usos que se le dio fue en las oficinas de correos, dónde a través de esta tecnología eran capaces leer el nombre y dirección del destinatario para así crear un código de barras que imprimen en el sobre con dicha información. Uno de los primeros servicios postales en disponer de dicha tecnología fue el de Estados Unidos.

En la actualidad, los usos que se le dan pueden ser muy diversos, pero hay que ser conscientes que el mayor punto de interés es la digitalización de archivos en todo tipo de soportes que con el paso del tiempo se van deteriorando. Este interés es debido a que la introducción de esta tecnología implica una reducción de recursos humanos y un aumento en la productividad manteniendo la calidad del servicio. Otro uso atractivo es la herramienta desarrollada por Google que permite a través de una imagen de un texto y la combinación de su potente traductor, traducir el texto de dicha imagen.

Con el desarrollo del potencial de la tecnología del OCR han proliferado las empresas que se han dedicado al desarrollo de software relacionado, siendo los más usados ABBY finereader, ReadIris, Omnipage y Tesseract. Todos ellos son de pago exceptuando el último nombrado.

Tesseract actualmente está considerado como uno de los motores OCR libres con mayor exactitud, por lo que sumado a su tipo de licencia será el que utilizaremos en la realización del proyecto.

### 2.4.1 Tesseract

Los inicios del Tesseract fueron en los laboratorios HP (*Hewlett Packard*) de Bristol en 1985 siendo un proyecto de doctorado y rápidamente ganó aliados pues se vio como un posible valor añadido para las tecnologías propias de HP. Este desarrollo se mantuvo hasta el 1995 que se envió a la Universidad de Las Vegas a la prueba anual de precisión de tecnologías OCR donde demostró su valía frente a otros motores de OCR comerciales.

Inicialmente estaba escrito en C, pero en el 1998 se migró a C++. Tras 10 años sin desarrollo, en 2005 fue liberado por HP como código abierto. Actualmente es desarrollado por Google y lo distribuye bajo la licencia de Apache. Esta licencia permite al usuario la libertad de usarlo para cualquier propósito, modificarlo y distribuirlo.

Este motor de OCR ofrece al usuario el reconocimiento de múltiples idiomas llegando a abarcar casi todos los idiomas. Para ello tienen en su repositorio los ficheros de diccionarios, que además podremos entrenar para obtener mejores resultados de clasificación para un tipo de fuente.

#### *2.4.1.1 Funcionamiento*

Para empezar, realiza un análisis de los componentes obteniendo así los contornos de los objetos que son candidatos a ser reconocidos de la imagen binaria de entrada. Esto otorga la ventaja de convertir la detección de texto negro sobre fondo blanco en el análisis del contorno que forma cada carácter.

Luego, agrupa los contornos cercanos en una misma área para poder procesarlos de forma individual. Estas áreas se organizan en líneas de texto para evitar confusiones y no se asignen palabras a líneas incorrectas.

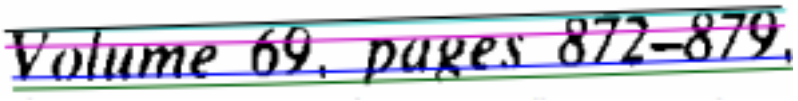


*Figura 4: Línea inclinada detectada por Tesseract*

A la hora de esta detección de líneas cabe destacar que posee un algoritmo que permite identificar líneas inclinadas como podemos observar en la figura 4. Utilizando este algoritmo se consigue solucionar el problema que sucede al escanear por ejemplo un libro, que al llegar las líneas al lomo del libro se inclinan.

También realiza una diferenciación entre líneas de texto según su ancho de carácter sea fijo o proporcional. La diferencia entre el tipo de anchura se puede observar en la figura 5. Esta diferenciación es debido a que en caso de tener un ancho fijo se realiza una división de los caracteres en celdas. En cambio, si son de ancho variable esta división las hará por palabras según los valores de espaciado predefinidos y a la vez, posteriormente, intentará descomponer estas palabras en caracteres.

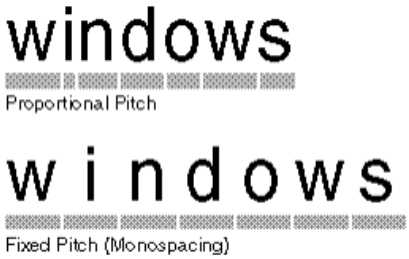


*Figura 5: Diferencia entre ancho fijo o proporcional, en una palabra*

Finalmente, una vez obtenidos los caracteres procede al primer intento de reconocimiento de caracteres mediante un clasificador. En caso de encontrarse antes un reconocimiento positivo del carácter se agrega al entrenador, de tal forma que tal y como se avanza en el análisis del texto se consigue una mayor capacidad de acierto. En caso contrario, éstas son marcadas para revisar en una segunda iteración con todo lo aprendido durante la primera iteración para así poder decidir sobre estos caracteres. Una vez finalizada esta segunda iteración el resultado será el archivo digital con el texto reconocido.

## 2.5 Redes Neuronales

Desde el punto de vista computacional una red neuronal es un grafo dirigido acíclico, evitando así los bucles infinitos, donde los nodos son llamados neuronas y los arcos son llamados conexiones. Esta representación implica que las salidas de unas neuronas serán las entradas de las neuronas con las que tienen conexión. Estas neuronas en lugar

de estar dispuestas de forma arbitraria están agrupadas en capas como se observa en la figura XX.

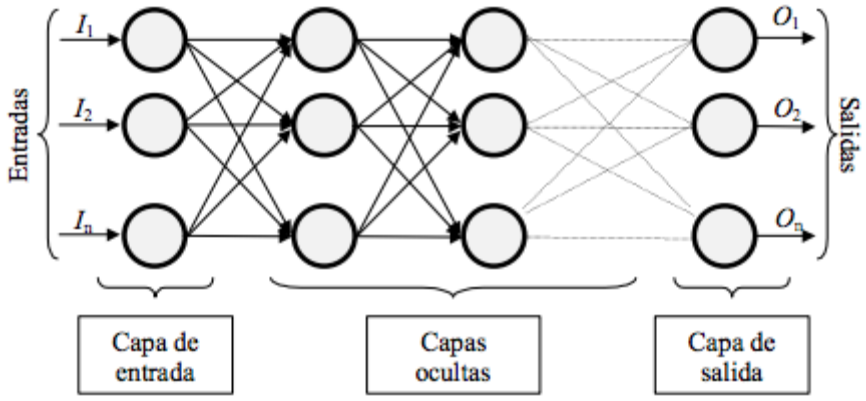


*Figura 6: Ejemplo de red neuronal de tipo Fully-Connected*

En la Figura 6 podemos observar una red neuronal del tipo *Fully-Connected*. Este tipo de red neuronal es el más común y su nombre hace referencia a que cada neurona de una capa está conectada a todas las neuronas de la capa siguiente.

En la Figura6 también podemos observar la arquitectura de una red neuronal:

- Una primera capa de entrada con tres neuronas que representan el vector de datos de entrada que es la encargada de recibir información del exterior.
- Una serie de capas ocultas que el número no puede ser determinado sin ser probado empíricamente para averiguar que configuración es la adecuada para un problema concreto. Estas capas ocultas son las encargadas de realizar la transformación de los datos de entrada a través de los pesos que las neuronas tienen en sus conexiones.
- Una capa de salida que representa el conjunto de las posibles etiquetas o clases en que se divide la solución del problema.

Las interconexiones entre las diversas neuronas con una organización jerárquica que intentan imitar el sistema nervioso biológico introducen las siguientes ventajas:

- Aprendizaje Adaptativo: Serán capaces de aprender a realizar tareas basadas en un entrenamiento previo.
- Auto-organización: en las etapas de aprendizaje, una red neuronal podrá crear su propia organización o representación de la información recibida.
- Tolerancia a fallos: por lo general, la destrucción parcial de una red neuronal conduce a una degradación de su estructura. No obstante, habrá capacidades de la red que podrán ser retenidas gracias a poseer la información distribuida.
- Operación en tiempo real: los cómputos neuronales pueden ser realizados en paralelo; para esto se diseñan máquinas con hardware especial para obtener esta capacidad o bien vía software.
- El proceso de entrenamiento de la red neuronal será el encargado de hacer que la red aprenda.

Para el cálculo del valor de una neurona, en primer lugar, se procede a realizar un sumatorio de todos los valores de entrada, calculados a través de la multiplicación del valor de la salida anterior por el peso de la conexión. Este método habilita mayor importancia a aquellas entradas con mayor peso manteniendo la característica de que los pesos estén ponderados.

Posteriormente a este valor obtenido previamente se le calculará una función de activación que definirá el tipo de neurona que estamos tratando.

Finalmente hay que proceder a calcular la función de salida que consiste en una un umbral que tiene que superar el valor resultante de la función de activación para mantenerse activa. Este proceso, puede tomar el nombre de función identidad y no tener la necesidad de determinar un umbral, en caso contrario el resultado ha de ser mayor o igual que el umbral para continuar activa.

Existen muchas funciones de activación (*sigmoid*, softmax, *tanh*, *maxout*, *ReLU*, etc) a continuación procedemos a comentar aquellas que han sido utilizadas en el desarrollo de este proyecto.

Por un lado, la función sigmoid consiste en transformar el valor x de la entrada en un valor de salida entre 0 y 1 aplicando la siguiente fórmula:

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

Otra función es la función lineal rectificada (*Rectified Linear Unit*, ReLU) la cual requiere un coste computacional menor debido a ser una función tan simple como:

$$f(x) = \max(x, 0)$$

Las funciones de activación, *softmax*, se suelen utilizar en las capas ocultas en cambio para la capa de salida hay que tener en cuenta que es un caso especial. El resultado ha de ser proporcional con respecto a todas las neuronas de la capa de salida, ya que el valor de la suma de todas estas neuronas ha de ser 1 por lo que toma la formula siguiente:

$$f_j\,(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

donde z es el vector K-dimensional de las salidas, K el nombre de salidas y j el nombre de la salida en cuestión. Esta forma de clasificación en las salidas nos facilitará una probabilidad, siendo esta la probabilidad que tiene el objeto de entrada de pertenecer a ese conjunto.

Uno de los procesos más importantes en una red neuronal es el proceso de aprendizaje. En él se realiza un entrenamiento de la red mediante el procesamiento a través de ella de una cantidad de datos bastante poblada que permitirá ajustar el valor de los pesos, así como del umbral. Estos datos se le suministran de manera aleatoria y secuencial. Este aprendizaje puede ser supervisado o no supervisado. Dentro de esta categorización se encuentra, en el modo supervisado, el aprendizaje por corrección de error en el que se introducen unos valores de entrada y los valores de salida obtenidos se comparan con los valores de salida correctos. Si se encuentran diferencias se ajusta la red. Este tipo de aprendizaje será el que se llevará a cabo en este proyecto.

Las redes neuronales pueden considerar como entrada una imagen, para ello adaptan el vector de entrada de la capa entrada hace referencia a los píxeles de dicha imagen reubicados como un vector obteniendo así tantas neuronas como píxeles contenga la

imagen. Sin embargo, presentan un problema complejo cuando se trabaja con muestras de altas dimensiones ja que, por un lado, se adquiere una compleja escalabilidad y, por otro, en el caso de las imágenes, el hecho de dedicar una conexión de un píxel a todas les neuronas puede que no funcione correctamente en la práctica. Esta problemática la explica Krizhevsky en [REF KRIZ], argumentando que las propiedades que contienen las imágenes no las tienen otros conjuntos de datos simplemente observando la matriz de covarianza. En ella se observa la principal característica que poseen las imágenes, esto es, los píxeles cercanos están fuertemente relacionados y aquellos píxeles más lejanos estarán débilmente relacionados.

Ante esta problemática, surge la necesidad de una herramienta que ataje ante el problema de la conexión total y se centre en regiones locales de la misma para realizar una extracción de información útil. Por ello aparece el concepto de red neuronal convolucional (*Convolutional Neural Network*, CNN) que asumen que los datos de entrada son imágenes optimizando así su arquitectura para trabajar con entradas de más de una imagen.

### 2.5.1 Redes Neuronales Convolucionales

Tras la problemática expuesta anteriormente, las redes neuronales convolucionales preparan las neuronas en forma tridimensional (ancho, largo y profundidad de capa) como se puede observar en la Figura 7. Este formato tridimensional pretende abarcar las dimensiones de una imagen ya sea formato RGB (*Red Green Blue*) o HSV (*Hue Saturation Value*). Este tipo de neurona, al igual que en una red neuronal convencional, podrán ser agrupadas en capas y, a su vez, las capas de este tipo podrán ser ubicadas una a continuación de las otras.

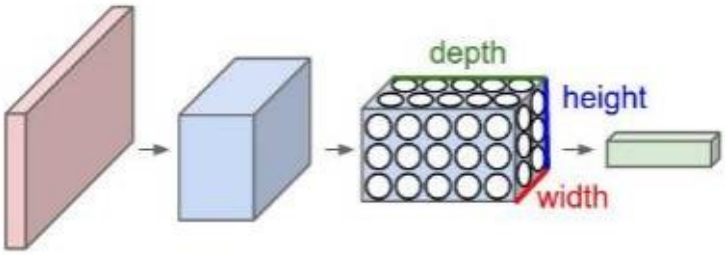


*Figura 7: Disposición de las neuronas en una CNN de forma tridimensional*

En el caso de las redes neuronales convolucionales se hace uso de diferentes tipos de capas para configurar la arquitectura, siendo estas: capa convolucional, capa *pooling* y capa *fully-connected*.

#### *2.5.1.1 Capa Convolucional*

En esta capa es donde se encuentra la idea principal de este tipo de red neuronal. Consiste en la realización de operaciones de productos y sumas entre la capa de partida y los K filtros a aplicar que generan un mapa de características, estas operaciones se pueden observar en la Figura 8 Estas características corresponden a cada posible ubicación del filtro en la imagen original. El mismo filtro sirve para extraer la misma característica en cualquier parte de la entrada reduciendo así el número de conexiones.

De modo, que si la entrada tiene un tamaño $m_{filas}$ x $m_{columnas}$ x $m_{profundidad}$ y el filtro tiene un tamaño de $n_{filas}$ x $n_{columnas}$ x $n_{profundidad}$ se obtendrá una salida de dimensiones $((m_{filas} - n_{filas}) + 1)$ x $(m_{columnas} - n_{columnas}) + 1)$ x $n_{profundidad}$).

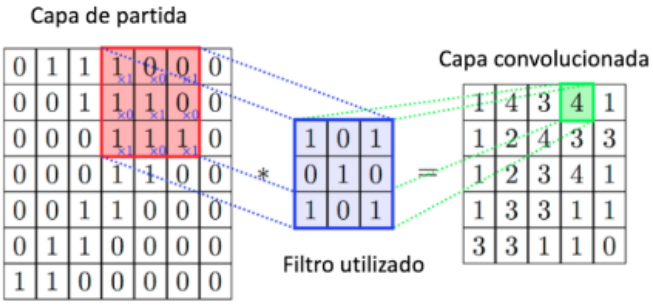


*Figura 8: Convolución de una capa 7x7 con un filtro 3x3*

### 2.5.1.2 Capa Pooling

Esta capa se encarga de reducir el nombre de parámetros la información de su entrada aplicando la operación *pooling*. Este método es frecuente introducirlo entre tipo de capas convolucionales para reducir el tamaño de representación, reduciendo así también el esfuerzo computacional y conseguir controlar el sobreentrenamiento.

La capa *pooling* opera independientemente en cada nivel de profundidad y la forma de realizar la reducción es mediante la extracción de estadísticas como el promedio o el máximo de una región fija del mapa de características. Habitualmente la reducción se realiza con la operación max con una ventana de 2x2 consiguiendo así una reducción en el volumen de los datos de entrada (anchura y altura) a la mitad. En la Figura 9 se observa un ejemplo de su uso.

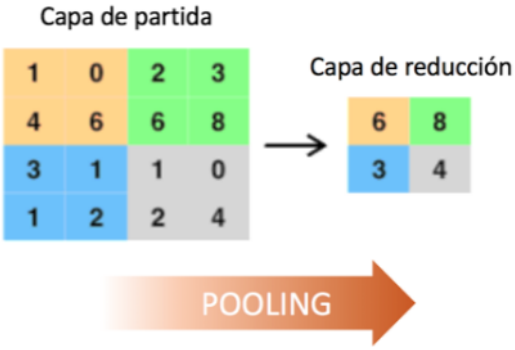


*Figura 9: Operación de max pooling con tamaño 2x2*

### 2.5.1.3 Capa Fully-connected

Este tipo de capa son las mismas que se utilizan en una red neuronal convencional, creando conexiones todos con todos. Habitualmente suele ser utilizada después de una capa convolucional o *pooling*, por lo que nos encontramos con que requiere reorganizar los elementos de la matriz tridimensional en un vector.

En estas capas existe un parámetro de gran utilidad, el *Dropout*. Este parámetro de configuración determina con que probabilidad va a existir la conexión entre dos neuronas. Este parámetro permite reducir el sobreentrenamiento de una red. El cambio entre la activación o no de este campo se puede observar en la Figura 10.

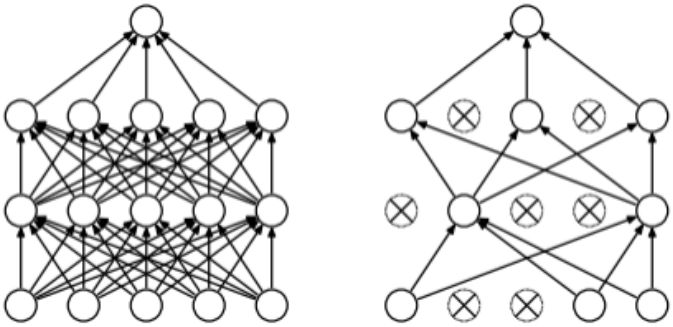

*Figura 10: Diferencia entre Red Neuronal sin o con Dropout*

Una vez presentadas las redes neuronales convolucionales y sus características más significativas, procedemos a realizar un pequeño análisis de los diferentes aplicativos donde podemos hallar estas estructuras.

Las redes neuronales no son una idea nueva, llevan compartiendo más de 50 años desde que se publicaron los primeros conceptos en este campo. Sin embargo, en los últimos años la revolución de la inteligencia artificial ha revivido el concepto de red neuronal y se han conseguido grandes avances gracias a las mejoras en los ordenadores y al uso de GPUs para este tipo de computaciones.

Este tipo de redes han revolucionado el mundo computacional y hoy en día las grandes corporaciones han realizado avances muy positivos en el tratamiento de números, de voz, objetos en imágenes, etc. Por ejemplo, Google ha desarrollado con Street View una red neuronal convolucional con una precisión del 96% a la hora de reconocer números de calle, la mejora del reconocimiento de voz en Android o, en un campo muy en alza también, reducir el consumo eléctrico en su centro de datos.

La cantidad de modelos de redes neuronales existentes abarcan diversos campos, en el siguiente punto nos centraremos en el modelo VGGFace que permite el reconocimiento de caras de personas. En el desarrollo de este trabajo nos hemos basado en esta red debido que es el modelo que más se acerca a la red neuronal que queremos configurar y por su alto porcentaje de acierto.

#### *2.5.1.4 VGGFace*

VGGFace es una red neuronal convolucional cuya implementación está basada en VGG (Visual Geometry Group), siendo esta última desarrollada por Karen Simonyan y Andrew Zisserman (Universidad de Oxford) en 2015.

VGGFace, mantiene la estructura de VGG, y está reentrenada específicamente para reconocimiento y verificación de caras. Esta red esta entrenada con 982,803 imágenes buenas, siendo el 95% frontales y un 5% de perfil. Estas imágenes pertenecen a 2,622 identidades distintas y permite llevar a cabo verificaciones con un acierto superior al 97%.

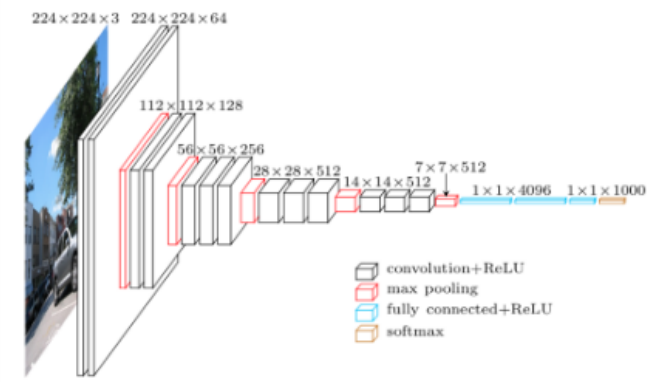


*Figura 11: Definición del modelo VGG*

Como se observa en la Figura 11 la arquitectura de VGG contiene 21 capas contando con la capa de *softmax* para realizar la distribución de probabilidades y su primera capa convolucional contiene 64 filtros.

El patrón típico que sigue esta red neuronal convolucional es que después de dos o tres capas de convolución viene una capa de combinación de *max pooling*. También contiene capas de *fully-connected en los últimos niveles de la red neuronal*.

## 2.6 Caffe

Caffe consiste en un *framework* que proporciona un marco limpio y modificable para el aprendizaje profundo, así como una colección de modelos de referencia, que ha sido desarrollada por Yangquin Jia como parte del proyecto final de doctorado en la Universidad de California en Berkeley bajo una licencia BSD. En la actualidad, es mantenido y desarrollado por BVLC (*Berkeley Vision and Learning Center*) con la ayuda de contribuyentes en GitHub (plataforma para alojar proyectos utilizando el sistema de control de versiones Git).

Caffe es utilizado para el entrenamiento, prueba, ajuste y despliegue de redes neuronales convolucionales y otros modelos profundos. Está desarrollado en C++ de forma modular, y en CUDA (*Compute Unified Device Architecture*, plataforma de computación en paralelo) para el cálculo de GPU (*Graphics Processing Unit*), ofreciendo interfaces para Python y Matlab.

En sus orígenes fue diseñado para la visión, no obstante, en la actualidad ha sido adoptado y mejorado por los usuarios para el reconocimiento de voz, robótica, neurociencia y astronomía.

Caffe

*Figura 12: Logo de Caffe*

La definición de capa de Caffe es la esencia de una capa de una red neuronal: toma uno o más *blobs (Binary Large Objects*, objetos binarios grandes*)* como entrada y produce uno o más *blobs* como salida. Además, proporciona un conjunto completo de los tipos de capas de las redes neuronales. Para la definición de las capas, Caffe utiliza un fichero de texto donde se definirá el tipo de capa y sus entradas, así como sus salidas.

# 3. Desarrollo

En este capítulo procedemos a exponer el desarrollo que se ha llevado a cabo en la realización del proyecto. Para ello nos centramos en los cuatro objetivos descritos en el punto 1.2 del documento.

En un primer lugar se extraen las muestras del rostro del usuario mediante el uso de clasificadores en cascada de *Haar*, posteriormente procedemos a extraer los datos del documento de identificación del documento de identificación y la extracción de datos de dicho documento mediante el reconocimiento óptico de caracteres. Finalmente, se pretende realizar la validación del usuario mediante el uso de redes neuronales convolucionales tomando como entrada las muestras del usuario como la imagen del rostro extraída del documento de identificación.

Además, inicialmente se realiza una descripción del entorno de trabajo y sus instalaciones más importantes.

## 3.1 Entorno de trabajo

Para llevar a cabo el proyecto ha sido necesario en primer lugar realizar la preparación del entorno de trabajo en el cual hemos procedido ejecutar a instalación de diversas herramientas.

El sistema operativo del que partimos es Ubuntu 16.04 y en el realizaremos la instalación de OpenCV 2.4 así como de Caffe y sus dependencias con otras librerías. También se procede a la instalación de la librería de Tesseract, que será la encargada de proporcionarnos el reconocimiento óptico de caracteres.

Una de las librerías importantes que también realizamos su instalación, más allá de por las dependencias que tiene con OpenCV, es GCC. GCC consiste en un compilador de C++, que necesitaremos para realizar la compilación de los códigos generados, pues C++ será nuestro lenguaje de desarrollo.

El entorno ha sido preparado en un computador con las siguientes características:

- Procesador: 2,4GHz Intel Core 2 Duo
- Memoria RAM: 8GB 1067 MHz DDR3
- Gráfica: NVIDIA GeForce 320M 256MB
- Disco Duro: HDD 250 SATA 3Gb/s, 5400 RPM, 8 MB Cache

## 3.2 Extracción de la imagen del usuario

El primer punto de nuestro proyecto será la obtención de la imagen del rostro del usuario en tiempo real a través de la webcam y de clasificadores en cascada de *Haar*. Para ello, se utiliza la implementación que proporciona OpenCV de dicho algoritmo.

La implementación proporcionada nos sirve de base para la realización de nuestra detección de cara, por lo que vamos a comentar los cambios más significativos que realizamos sobre la implementación que nos proporcionan.

El método *main* que nos proporcionan para nosotros será una función dentro de nuestro programa. A esta función se les realiza los cambios siguientes:

- Como nos centramos en el reconocimiento de rostro se obvia el reconocimiento de ojos, cargando así solo el fichero de clasificadores en cascada *Haar* ya entrenado de caras haarcascade_frontalface_alt.xml.
- La implementación ofrecida utiliza el tipo *CvCapture* para trabajar con el dispositivo de cámara, nosotros realizamos cambio al tipo *VideoCapture*. Estos son parecidos, *CvCapture* es la implementación para C mientras *VideoCapture* es la implementación para C++. Por lo mismo se introducen las lecturas de los fotogramas acordes a este tipo.
- La creación de un contador para evaluar cuantas muestras queremos recoger de la cara y así finalizar la ejecución de este proceso.

El método detectAndDisplay que nos proporcionan también recibe cambios:

- El tipo de la respuesta del método se ha cambiado de *void* a *String* para controlar desde el *main* que imagen de la cara se guarda, así como controlar el número de muestras.
- Una vez realizada la detección de la cara, se añade un tratamiento a la región que la contiene para posteriormente poder guardar el área de interés con el rostro. También, tal y como hemos comentado al principio de este punto, se obvia el reconocimiento de los ojos.

```
1  Point center( faces[i].x + faces[i].width*0.5, faces[i].y + faces[i].height*0.5 );
2  //Definició de la subimatge que contindrá la cara
3  cv::Mat subImg;
4  if( (faces[i].y-30 + ((faces[i].height + 20)*145)/110) >= frame.rows || (faces[i].y - 30) < 0 || (faces[i].x - 10) < 0 || (faces[i].x - 10 + (faces[i].width + 20)) >= frame.cols){
5    subImg = frame(cv::Range(faces[i].y, faces[i].y + faces[i].height), cv::Range(faces[i].x , faces[i].x + faces[i].width )).clone();
6  }
7  else{
8    subImg = frame(cv::Range(faces[i].y - 30, (faces[i].y-30 + ((faces[i].height + 20)*145)/110)), cv::Range(faces[i].x - 10, faces[i].x - 10 + (faces[i].width + 20))).clone();
9  }
10
11 std::ostringstream ss;
12 ss << directorioResultados << "face" << time(NULL) << rand()%100 << ".jpg";
13
14 cv::imwrite(ss.str(),subImg);
15 captureFace = ss.str();
16 rectangle(frame, Point(faces[i].x, faces[i].y), Point(faces[i].x + faces[i].width, faces[i].y + faces[i].height), CV_RGB(0,255,0));
```

*Figura 13: Cambios realizados en detectAndDisplay una vez ya tenemos el rostro detectado*

Tal y como se observa en la Figura 13, se realiza el cálculo de la sub imagen que será aquella que contenga el rostro y será la que nos interese guardarnos para poder procesarla posteriormente. Para el cálculo de esta región, que va de la línea 4 a la 9 de la Figura 13, se realiza un cálculo proporcional dependiendo de si el rostro es detectado en los bordes del fotograma.

## 3.3 Detección y normalización del documento de identificación

El segundo punto en el proyecto es la toma de imágenes en tiempo real al documento de identificación del usuario. No obstante, no nos sirven unas imágenes tomadas sin control, sino que se necesita confirmar que en la imagen existe el documento de identificación.

A fin de llevar a cabo esta identificación del documento el primer paso a realizar es la definición de una plantilla del objeto a buscar. En la realización de este proyecto se ha definido como plantilla una imagen del Documento Nacional de Identificación

en su versión DNIe (Documento Nacional de Identificación electrónico) que podemos observar en la Figura 14.


*Figura 14: Plantilla del objeto (DNIe) a buscar*

Con la plantilla definida, se recure a las técnicas de detección de detectores y descriptores de puntos de interés. Estas técnicas, se deben realizar tanto en nuestra plantilla como en la imagen de tiempo real. Una vez tenemos los descriptores en nuestro poder de ambas imágenes, se recurrirá al proceso de *matching* que nos permitirá la correspondencia entre los puntos de la plantilla con la imagen de real. Este proceso, también es conocido como encontrar homografía entre imágenes a través de sus puntos de interés.

Una vez explicado el proceso a realizar vamos a pasar a la fase de implementación. En este caso nos hemos basado en el tutorial que ofrece OpenCV para la identificación de puntos característicos en imágenes 2D y la identificación de objetos a través de sus homografías.

Del mismo modo que ocurre en el caso anterior, el código que nos proporcionan se encuentra en una función *main* y nosotros nos declararemos nuestra función propia dentro de nuestro aplicativo a la cual dotaremos de esta funcionalidad. Además, en este caso también realizamos cambios para conseguir un valor añadido.

Los cambios que realizamos son los siguientes:

- Nos basamos en la toma de imágenes en tiempo real, por lo que la plantilla si estará cargada de memoria, en cambio la otra imagen a analizar estará tomada en tiempo real a través de la clase VideoCapture como en el proceso del punto 3.2.
- Una vez calculados los descriptores de ambas imágenes y se realiza el proceso de *matching* para encontrar la correspondencia, se calcula si los puntos de las esquinas que nos da el matching están contenidos dentro de la imagen. De esta forma en caso de no estarlo se le indica al usuario que introduzca el documento dentro de la imagen.
- Con los puntos de las esquinas también calculamos que el área que describen mantenga la proporción aurea tal y como cumplen los documentos de identificación. Para ello se analiza durante la ejecución que proporción suele marcar esta área y acotamos al intervalo entre 1.5 y 1.8 para darlo como válido.
- Se dota de un contador para evaluar cuantas muestras queremos recoger del documento y así finalizar la ejecución de este proceso.

Para la toma de la decisión de qué tipo de detector y de descriptor funcionarían mejor en nuestro proyecto recurrimos a la prueba y error con nuestra plantilla. Durante este ensayo también se ha ido mejorando la plantilla, así como ampliando, pues inicialmente la plantilla constaba de solo un área perteneciente a la esquina izquierda superior intentando así trabajar con menor área y ajustar mejor los parámetros. Finalmente, se ha obtenido mejores resultados con un detector de tipo GFTT y un descriptor de tipo SIFT. El tipo de datos con el que se trabaja en la realización del proyecto son de carácter sensible, dificultando así que puedan estar al abasto del público general.

Una vez tenemos el documento de identificación localizado dentro de la imagen tomada es hora de procesarlo, para ello la imagen tiene que ser proyectado sobre un espacio normalizado del que extraer la información.

### 3.3.1 Extracción de los datos del documento de identificación

Para la extracción de información del documento de identificación que acabamos de tomar por la webcam, procedemos a dividir estas imágenes en subáreas que contendrán el texto a extraer.

Para la extracción de este texto utilizamos Tesseract. Inicialmente el tesseract se pensaba manejar desde OpenCV, pero por problemas con la librería acabamos manejándola desde C++ pero interaccionando con el terminal.

Como se comentó en el estado del arte, las imágenes que analiza tesseract tienen que ser en blanco y negro, por lo que hay que realizar una serie de cambios en la imagen.

```
1. Mat frame_HSV;
2. cvtColor(output, frame_HSV, CV_BGR2HSV);
3. std::vector<cv::Mat> HSV_planes;
4. cv::split(frame_HSV,HSV_planes);
5. output = HSV_planes[2].clone();
```

Para quedarse con la imagen en blanco y negro aplicamos el código anterior. La imagen que tenemos que transformar esta en la variable output, a la cual se le realiza un cambio de BGR (en OpenCV esta es la disposición de los canales RGB) al modelo HSV (*Hue*, *Saturation*, *Value*) quedándonos con la capa del valor.

Posteriormente a esta imagen se le realiza una transformación a través de una función que hemos definido y que se puede observar en el bloque de código posterior:

```
1. Mat transformarImp(Mat imagenReal)
2. {
3.   Mat tmp;
4.   Mat tmpHSV;
5.
6.   cv::normalize(imagenReal,tmpHSV,255,0,NORM_MINMAX);
7.
8.   cv::resize(tmpHSV,tmpHSV,cv::Size(tmpHSV.cols*2,tmpHSV.rows*2));
9.   cv::GaussianBlur(tmpHSV,tmp,Size(15,15),0,0);
10.
11.   tmp = tmp - tmpHSV;
12.
13.   cv::normalize(tmp,tmp,255,0,NORM_MINMAX);
14.   cv::threshold(tmp, tmp, 0.0, 255.0, THRESH_BINARY | THRESH_OTSU);
15.
```

```
16.   return tmp;
17. }
```

Una vez la imagen ya preparada para ser tratada por tesseract, teniendo el estilo de la Figura 15, procedemos a ello y analizamos el texto que nos devuelve. En este caso nos hemos centrado en la extracción del número de documento de identificación, de esta forma, podemos controlar que se realice correctamente la lectura definiendo que del primer carácter numérico han de continuarlo siete caracteres numéricos y una letra. A posteriori analizamos que esta letra cumpla con la letra que le corresponden a la cadena numérica obtenida.

En caso de pasar este filtro se acepta la imagen encontrada en la ejecución y se procese a guardar la subregión que contiene la imagen del documento de identidad que será la que tendremos que validar contra la imagen tomada en el paso 3.2.


*Figura 15: Imagen tratada y preparada para ser analizada por Tesseract*

## 3.4 Validación del usuario

La validación del usuario consiste en que a partir de la imagen que se ha tomado del rostro del usuario en tiempo real y de la imagen que contiene el documento de identidad que se ha sustraído también en tiempo real poder validar que se trata del mismo usuario.

Para ello, se hace uso de las redes neuronales convolucionales en especial nos basamos en el modelo VGGFace que se ha presentado anteriormente. Para ello, se ha configurado una red que recibe como parámetros de entrada dos imágenes y produce dos salidas siendo estas con valores 0 (negativo) y 1(positivo).

### 3.4.1 Entrenamiento de la red

El entrenamiento de la red se ha realizado de forma supervisada, introduciendo los dados y su solución. Para ello, se ha tenido que recolectar muestras de este tipo de imágenes cosa que ha resultado dificultosa por el contenido sensible que contienen los documentos de identidad.

Debido a esta dificultad, hemos producido una combinación de imágenes partiendo de la base de imágenes lfw-face-dataset.

Para la realización de estas combinaciones se han realizado composiciones a la par entre todas las imágenes de todas las personas. Estas combinaciones están montadas sobre una imagen de 3 canales, donde el primer canal contiene una imagen, el segundo contiene ceros y el tercero la otra imagen. Aquellas que están compuestas por imágenes de la misma persona son las muestras con resultado positivo, mientras que una composición de una imagen de una persona con otra de una persona diferente su resultado es negativo.

# 4. Conclusiones

## 4.1 Valoración personal

La realización de este proyecto ha resultado ser una labor costosa. No obstante, también ha resultado ser una gran experiencia a nivel de aprendizaje propio.

La cantidad de conceptos nuevos de inteligencia artificial y aprendizaje automático que se han tenido que analizar para llevar a cabo la realización del proyecto han resultado muy de mi interés.

Este tipo de conceptos, durante la carrera, no han sido estudiados con profundidad. Pues pertenecen a una especialización completamente lejana a la que durante estos años he cursado.

Respecto a las redes neuronales convolucionales, mi valoración es que tienen un potencial muy elevado pero un coste computacional alto que con mis recursos ha sido difícil manejar. Para empresas con ordenadores más potentes, este campo puede agilizar muchos procesos y también reinventar algunos de ellos.

## 4.2 Futuras ampliaciones y mejoras

En el desarrollo inicial del proyecto se han controlado diversos factores como aquellos humanos. Por ejemplo, no se obliga al usuario a introducir el documento de identidad dentro de un recuadro especifico, pero sí que hemos controlado el que el usuario ponga el documento fuera de la imagen.

No obstante, sobre haber controlado algunos de estos factores, otros se nos han escapado como por ejemplo que en la toma de muestras de la imagen del usuario realmente pueda haber dos rostros en la imagen.

Otro factor a estudiar es la composición de la red neuronal convolucional, igual se debería de intentar buscar otra forma que no se necesitaran tantas muestras para el entrenamiento ya que el tipo de imagen que se requiere (imagen del documento de identidad) no está al abasto tan fácilmente. También se debería estudiar como poder conseguir una imagen lo más parecida a las que contienen los documentos de identidad (por color, resolución y posición) a través de aquellas imágenes que si tienen libre acceso.

# 5. Bibliografía


1. OpenCV. *About* < https://opencv.org/about.html > [Consulta: Junio 2017]
2. OpenCV. *Welcome to opencv documentation!* <https://docs.opencv.org/2.4.13.4/> [Consulta: Julio 2017]
3. Ordoñez P., J. (2009) "Estado del Arte" <https://jpordonez.files.wordpress.com/2009/06/estado-del-arte.pdf> [Consulta: Junio 2017]
4. https://research.google.com/pubs/pub33418.html
5. https://arxiv.org/pdf/1408.5093.pdf
6. http://www.merl.com/publications/docs/TR2004-043.pdf
7. https://goo.gl/ildmLA
8. N Srivastava, GE Hinton, A Krizhevsky, I Sutskever, R Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research 15 (1), 1929-1958
9. https://gizmodo.com/google-built-an-insane-neural-network-to-id-house-numbe-1499897368
10. https://www.xatakandroid.com/aplicaciones-android/android-mejora-gracias-al-estudio-de-redes-neuronales
11. https://www.genbeta.com/actualidad/como-usa-google-redes-neuronales-para-ahorrar-electricidad
12. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf [REF KRIZ]
13. https://arxiv.org/abs/1311.2901
14. http://www.robots.ox.ac.uk:5000/~vgg/publications/2015/Parkhi15/parkhi15.pdf
15. Caffe. *Ubuntu Installation* <http://caffe.berkeleyvision.org/install_apt.html> [Consulta: Junio 2017]
16. GCC. *Installing GCC.* <https://gcc.gnu.org/install/> [Consulta: Junio 2017]
17. OpenCV. *Cascade Classifier* <https://docs.opencv.org/2.4/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html > [Consulta: Junio 2017]
18. Tesseract. *Wiki Tesseract* < https://github.com/tesseract-ocr/tesseract/wiki/Compiling > [Consulta: Mayo 2018]
19. OpenCV. *Features2D + Homography to find a known object.* <https://docs.opencv.org/2.4/doc/tutorials/features2d/feature_homography/feature_homography.html> [Consulta: Junio 2017]