



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA

Reconocimiento de Expresiones Faciales

por Computador

Autor

Francisco José Fajardo Toril

Directores

Miguel García Silvente
Eugenio Aguirre Molina



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, 7 de septiembre de 2017



Reconocimiento de Expresiones Faciales

por Computador.

Autor

Francisco José Fajardo Toril

Directores

Miguel García Silvente
Eugenio Aguirre Molina

Reconocimiento de Expresiones Faciales: por Computador

Francisco José Fajardo Toril

Palabras clave: Reconocimiento, expresión, facial, visión, computador, aprendizaje, automático, redes, neuronales, convolucionales.

Resumen

El objetivo de este documento es mostrar al lector el proceso seguido para el desarrollo de una aplicación software capaz de resolver el problema del reconocimiento de expresiones faciales por computador, que puede ser descrito como:

Dada una imagen de entrada, obtener la etiqueta de salida que mejor identifique la expresión facial que aparece en dicha imagen.

De entre todas las técnicas posibles para la resolución del problema, he basado mi solución en el entrenamiento una red neuronal convolucional. El modelo entrenado clasifica la imagen de entrada mediante la asignación de una (o varias) etiquetas, incluyendo un porcentaje por cada una que indica cuál es la más probable de estar presente en dicha imagen. El conjunto de posibles etiquetas es:

- | | |
|-------------|-------------|
| ■ Afraid | ■ Neutral |
| ■ Angry | ■ Sad |
| ■ Disgusted | |
| ■ Happy | ■ Surprised |

Facial Expression Recognition: by Computer

Francisco José Fajardo Toril

Keywords: Facial, expression, recognition, computer, vision, machine, learning, convolutional, neural, network.

Abstract

This document main objective is to show the reader the process I followed to develop a software application which is able to solve the face expression recognition problem through computer. It can be described as:

To obtain the tag that better describe the facial expression that appears in the given input image.

Between all the possible techniques that allow us to solve this problem, I chose to train a convolutional neural network. Trained model classify the input image by the assignment of one or few tags, including a percentage that suggest which is the most probable tag to appear in that image. Here is the whole set of possible tags:

- | | |
|-------------|-------------|
| ■ Afraid | ■ Neutral |
| ■ Angry | ■ Sad |
| ■ Disgusted | |
| ■ Happy | ■ Surprised |

Yo, **Francisco José Fajardo Toril**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 76424577Q, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Francisco José Fajardo Toril

Granada a 7 de septiembre de 2017.

D. **Miguel García Silvente**, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento YYYY de la Universidad de Granada.

D. **Eugenio Aguirre Molina**, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento YYYY de la Universidad de Granada..

Informan:

Que el presente trabajo, titulado ***Reconocimiento de Expresiones Faciales, por Computador***, ha sido realizado bajo su supervisión por **Francisco José Fajardo Toril**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 7 de septiembre de 2017.

Los directores:

Miguel García Silvente Eugenio Aguirre Molina

Agradecimientos

Poner aquí agradecimientos...

Capítulo 1

Introducción

1.1. Motivación

El problema del reconocimiento de expresiones faciales es un paso adelante tras resolver el problema del reconocimiento facial en imágenes. Tal y como mencioné en el resumen puede ser descrito de la siguiente manera:

Dada una imagen de entrada, obtener la etiqueta de salida que mejor identifique la expresión facial que aparece en dicha imagen.

A priori la solución para un ser humano es trivial. Tenemos muchas maneras de adivinar el estado de ánimo de una persona, pero si tenemos que basarnos en nuestra visión, una posible solución sería: *"Simplemente observa los rasgos faciales del sujeto e intenta adivinar su estado de ánimo basándote en tu experiencia pasada con otros seres humanos."*

Imagine que deseamos desarrollar un agente basado en inteligencia artificial para que interactúe con otros seres humanos. Sería de gran utilidad que este agente basara sus acciones en función del estado de ánimo de la persona/as con las que está comunicándose para lograr de esta forma un mayor grado de comprensión o empatía. De la misma manera podríamos pensar en una aplicación que escoge la música idónea para nosotros en función de nuestro estado de ánimo en ese momento a través de una captura mediante la cámara del computador.

1.1.1. Trabajos relacionados

Como se puede ver, la solución a este problema puede ser de utilidad en diversas aplicaciones del mundo real, sin embargo, ¿Cómo podría un computador resolver este problema? Hay diversas propuestas que han tratado de resolver este problema.



Figura 1.1: Puntos de Referencia extraídos con ASM.

Facial Expression Analysis using Active Shape Model[1]

Active Shape Model(ASM)[3] es una técnica que permite, dada una imagen del objeto para el que fue entrenado, localizar un conjunto de puntos de referencia previamente definidos tal y como se puede ver en la Figura 1.1. En el caso de un rostro humano estos puntos pueden estar los principales indicadores de la expresión facial, como pueden ser las cejas, boca, ojos... En este paper [1] en concreto, se tienen en cuenta para cada cara, 68 puntos de referencia. Se procesan los puntos extraídos y se obtiene un modelo geométrico para ese rostro concreto. Acto seguido, haciendo uso de la técnica de clasificación conocida como Support Vector Machine(SVM), se clasifica ese modelo geométrico para predecir a qué expresión facial pertenece obteniendo un porcentaje de acierto del 92,1 %.

Facial expression recognition and synthesis based on an appearance model[2]

En este caso se hace uso de la técnica *Active Appearance Model (AAM)*[4], un método estadístico que se emplea para emparejar, un modelo geométrico, a una nueva forma del mismo tipo para el que fue entrenado, pero que el modelo no ha visto antes, como por ejemplo un nuevo rostro humano. Esta técnica (AAM) es una generalización de la ya antes mencionada ASM[3]. Además, lo que en este caso se intenta es que la información usada esté exenta de redundancia por lo que se aplican técnicas de aprendizaje no supervisado en una fase de preprocessamiento de la información como *Análisis de Componentes Principales (PCA)* o *Análisis de Componentes Independientes(ICA)* para reducir la dimensionalidad de la información en una fase de preprocessamiento. En este caso obtienen alrededor de un 84 % de acierto.

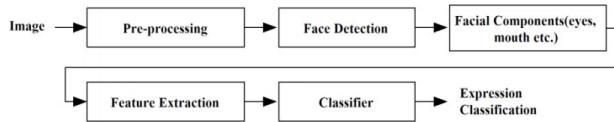


Figura 1.2: Diagrama general del proceso de reconocimiento.

1.1.2. ¿Qué se pretende conseguir con este proyecto?

Las técnicas de ASM[3] y AAM[4] son sensibles a rotaciones de la imagen, es decir, será difícil extraer los puntos de referencia en imágenes que disten mucho del concepto de imagen frontal, como por ejemplo una la fotografía de una persona mirando hacia un lado con una expresión facial determinada. Además, en la Figura 1.2 extraída del siguiente paper[5], se puede ver el proceso general que se sigue a la hora de clasificar una imagen, en nuestro caso reconocimiento de expresiones faciales.



Figura 1.3: Diagrama del proceso objetivo.

La idea principal que se quiere conseguir para resolver el problema consiste en simplificar de alguna manera dicho diagrama para que se asemeje al de la Figura 1.3 de forma que no sea necesario extraer características cada vez que se clasifique una nueva imagen. Otra de las propuestas es que la rotación de la imagen de entrada no suponga un hándicap a la hora de clasificarla. El propósito de estas medidas es:

- Ganar en robustez con imágenes rotadas o inclinadas.
- Realizar la clasificación en tiempo real simplificando el preprocesamiento de las imágenes.
- Combinar todo en una aplicación de escritorio con una interfaz simple por encima para que cualquier usuario pueda utilizarla.

1.1.3. Organización del documento

Este documento tratará de explicar el proceso seguido para el desarrollo de este trabajo. Para ello, se estructurará en los siguientes capítulos:

Portada - Información básica sobre el nombre del proyecto, la institución, el autor y los tutores.

Prefacio - Resumen, palabras clave y agradecimientos.

Introducción - Motivación del trabajo y estado del arte de este campo de investigación.

Objetivos - Se planteará un objetivo principal el cual será dividido en sub-objetivos más simples para poder llevar a cabo la tarea principal así como un resumen de las técnicas aprendidas durante la formación académica empleadas en este trabajo.

Resolución del trabajo - Este capítulo se dividirá en varias secciones, en las que se explicará el método de ingeniería del software empleado para la resolución del trabajo, así como los recursos hardware y software, la especificación de requisitos, planificación, análisis funcional e implementación de la aplicación.

Conclusión - Se hará una valoración del producto final y se valorará si se han alcanzaron o no los objetivos iniciales y en qué medida, teniendo en cuenta los puntos fuertes y débiles de la solución.

Bibliografía - Referencias y citas bibliográficas usadas.

Anexo - Pequeño manual de cómo usar la aplicación.

Capítulo 2

Objetivos

2.1. Objetivo principal

Desarrollar una aplicación de escritorio que dada una imagen detecte si aparece un rostro humano y en que en caso afirmativo, sea capaz de mostrar por pantalla, la etiqueta que mejor identifica la expresión facial que presenta dicho rostro.

Las etiquetas de las expresiones que deseamos predecir son:

- | | |
|-------------|-------------|
| ■ Afraid | ■ Neutral |
| ■ Angry | ■ Sad |
| ■ Disgusted | |
| ■ Happy | ■ Surprised |

Se ha de cumplir este objetivo sin olvidar qué es lo que se pretende alcanzar con este trabajo (apartado 1.1.2). Esto es, simplificar el proceso de pre-procesamiento de las imágenes y de esta forma ver si es factible un proceso de clasificación en tiempo real. Teniendo esto en mente, dividiremos el objetivo principal en distintas fases o sub-tareas.

2.2. Objetivos específicos

1. Preparación de la información con la que entrenaremos el clasificador.
 - a) Obtener una base de datos.
 - b) Pre-procesamiento de la información.
 - c) Creación del conjunto de training, validación y test.
2. Entrenamiento del clasificador.

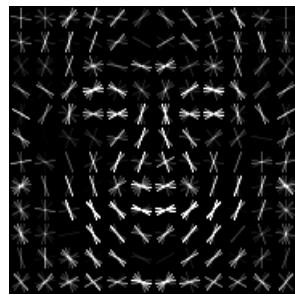


Figura 2.1: Visualización de descriptores HOG en un rostro humano [12]

- a) Selección de parámetros.
- b) Realización de pruebas (Vuelta al paso 2 si es necesario).
3. Desarrollo de interfaz que haga uso del clasificador.

2.3. Técnicas empleadas

Hasta ahora hemos hablado de objetivos, sin entrar demasiado en detalle acerca de qué técnicas serán las que emplearemos para llevarlos a cabo. En la siguiente sección hablaremos sobre ellas, dando una pequeña explicación de su fundamentación teórica.

2.3.1. Detector Facial

Dada una imagen, lo primero que necesitamos saber es si aparece un rostro humano o no y en función de ello hacer que nuestra aplicación, actúe de una manera u otra. De esto es de lo que se encarga precisamente un detector facial.

Para llevar a cabo esta tarea, el detector facial que he usado ha sido entrenado para aprender mediante un algoritmo basado en *SVM*[8] los descriptores de características de una batería de imágenes de rostros humanos previamente marcados y localizados manualmente (Si pudiésemos automatizar el proceso de marcado no necesitaríamos un detector facial). Los descriptores de características aprendidos por el detector son los conocidos como *Histogram of Oriented Gradients (HOG)*[6] de los que hablaremos a continuación.

¿Qué es un descriptor de características?

Es una representación de un conjunto de píxeles, ya sea de una imagen completa o de un trozo de esta. Su misión es simplificar la información que representa una imagen de forma que el descriptor represente la información más útil de la imagen eliminando aquella que no es representativa.

Histogram of Oriented Gradients (HOG)

Hay muchos descriptores de características distintos, pero los que usa nuestro detector son conocidos como descriptores HOG. Estos cuentan las ocurrencias de orientaciones de gradiente en zonas localizadas de una imagen. El gradiente nos indica por cada píxel la magnitud del mayor cambio de intensidad posible y su dirección de oscuro a claro. Podemos observar un ejemplo de esto en la Figura 2.1.

Usando un algoritmo[7] basado en la técnica de aprendizaje automático conocida como *Support Vector Machine*[8], el detector es capaz de aprender la composición de los descriptores de características HOG de un gran número de caras distintas.

Así, dada una imagen, se calculará su descriptor de características y si se encuentra en ella una ventana que contenga descriptores que se asemejen a los aprendidos, habremos detectado satisfactoriamente una cara.

2.3.2. Redes Neuronales Convolucionales

Tal y como mencioné en la motivación de este trabajo, sin entrar demasiado en detalles, que una de las restricciones de este proyecto era minimizar la fase de pre-procesamiento (apartado 1.1.2) antes de clasificar una imagen.



Figura 2.2: Esquema general de detección haciendo uso de algoritmos basados en Machine Learning [13]

Como podemos ver en la Figura 2.2 haciendo uso de algoritmos de *Machine Learning* debemos extraer, de la imagen de entrada, una serie de características que la identifiquen, dárselas al algoritmo escogido (previamente entrenado usando un conjunto con esas características) el cual, nos proporcionará una salida en función de lo que el clasificador haya aprendido. Esto supone que los algoritmos clásicos basados en *Machine Learning* son fuertemente dependientes de la representación de las características que se escoja para entrenarlos[9].

Los algoritmos de *Deep Learning* sin embargo nos permiten omitir esa fase de extracción de características, porque han sido diseñados para extraer y abs-



Figura 2.3: Detección usando algoritmos basados en Deep Learning [13]

traer su propia representación de la información a través de la composición de varias transformaciones no lineales[9]. En nuestro caso, como el problema trata de clasificar imágenes y viendo los buenos resultados obtenidos en los últimos años en problemas de visión por computador[14], el algoritmo de *Deep Learning* escogido será el conocido como Redes Neuronales Convolucionales o CNNs. De esta manera, el flujo que se seguiría para realizar la clasificación de una imagen usando CNNs será similar al de la Figura 2.3. Podemos abstraer las CNNs como una caja negra que recibe una imagen de entrada, sin ningún tipo de preprocesamiento previo, y proporciona una etiqueta de salida.

Composición

Dicho esto, podemos dar una pequeña visión de qué hay dentro de esa caja negra. La arquitectura de una CNN se define como un conjunto de capas conectadas entre si. Hay varios tipos de capas[16] y cada una de ellas realiza una operación específica sobre los datos de entrada que recibe y proporciona unos datos de salida, los cuales que vuelven a ser los datos de entrada para la siguiente capa tal y como podemos observar en la Figura 2.4.

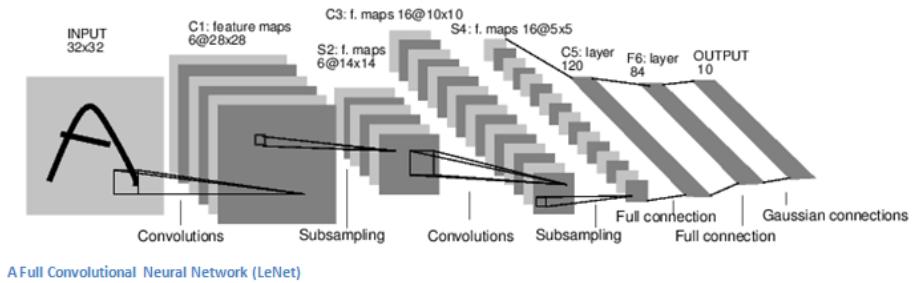


Figura 2.4: Arquitectura de Red: LeNet[14]

Funcionamiento

Ahora que conocemos de qué se compone una CNN vamos a ver a grandes rasgos cómo funciona una red neuronal convolucional. Cada una de las capas de las mencionadas en la sección anterior (2.3.2) se compone a su vez de

una serie de neuronas, las cuales tienen unos parámetros, que irán se irán ajustando (aprendiendo) en la fase de entrenamiento de la red, conocidos como pesos. Las neuronas de una capa, estarán solo conectadas con una pequeña región de la capa anterior. De esta forma, a medida que la red va aprendiendo, capas más avanzadas abstraen la información en mayor medida y aprenden patrones más complejos.

Dada una imagen de entrada, las neuronas de la red se van activando en función de los patrones aprendidos, y en la última capa calcula la puntuación de que la imagen pertenezca a cada una de las clases para las que la red fue entrenada, dando de esta forma, una etiqueta de salida. Hay muchos tipos de capas que pueden formar la arquitectura de red de una CNN. A continuación veremos un resumen de las más importantes y por norma general, las más usadas.

Capas de entrada [16] - Tal y como su nombre indica, recogen los datos n-dimensionales para la red. Suele ser la primera capa de nuestra arquitectura de red. En nuestro caso, los datos de entrada son las imágenes, que serían elementos tridimensionales teniendo de esta manera un vector del tipo $[ancho \times alto \times profundidad]$. La profundidad puede ser 1 o 3 dependiendo de si la imagen es en color (3 canales RGB) o en escala de grises (1 canal).

Capas de convolución [16] - Calcula la salida de neuronas que están conectadas a zonas locales de la imagen de entrada. El valor de cada neurona de salida es el producto escalar entre los pesos de las neuronas de la capa anterior y esa pequeña área local de la imagen. Podemos ver una representación gráfica en la Figura 2.5

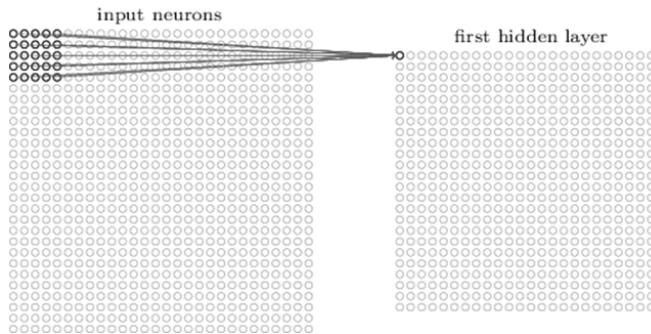


Figura 2.5: Filtro 5×5 convolucionando sobre un volumen de entrada[14].

Capas no lineales o Capas de activación[10, 15] - Por convenio, tras una capa de convolución se suele aplicar una capa de activación con el objetivo de añadir *no-linealidad* a un sistema que ha estado haciendo operaciones lineales durante las capas de convolución. Un ejemplo de

función no lineal, y una de las más usadas en capas no lineales es:

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

También conocida como *Rectified Linear Units* o función *ReLU* que transforma únicamente los valores negativos en cero.

Capas de submuestreo - Reduce el tamaño de la imagen a la mitad por norma general. Se pueden aplicar diferentes técnicas para realizar esta operación. Una de las más conocidas se denomina *maxpooling* y podemos ver como funciona en la Figura 2.6. Esta transformación actúa en las dimensiones espaciales de la imagen (Anchura y Altura).

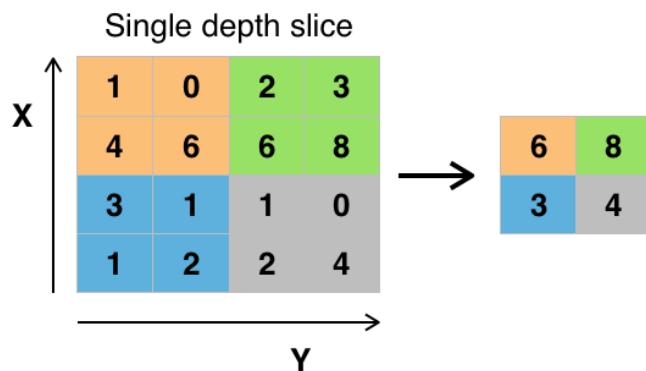


Figura 2.6: Reajuste de una imagen 4×4 a una imagen 2×2 usando *max-pooling*[15].

Capa totalmente conectada[16] - Esta capa calculará la puntuación de cada clase obteniendo una estructura de salida del tamaño $[1 \times 1 \times n]$ donde n es el número de clases de salida. En nuestro caso $n = 7$ (2.1).

Este ha sido un pequeño resumen acerca de lo que las redes neuronales convolucionales ofrecen. Es un tema muy extenso el cual no se puede cubrir de forma completa en este documento, además de no ser el objetivo de este. Si se desea saber más acerca de este tema recomiendo el siguiente libro [17].

Transfer Learning [20]

Se trata de una técnica que se aplica en la fase de entrenamiento de una red neuronal. Consiste en tomar los pesos de un entrenamiento previo como pesos iniciales para nuestro entrenamiento. Dicho de otra manera, es una forma de reutilizar lo que una red neuronal ha podido aprender en otro entrenamiento.

Esta técnica es muy útil cuando queremos ahorrar tiempo durante la fase

de entrenamiento ya que aumenta la velocidad a la que converge el entrenamiento, o bien cuando el número de imágenes que tenemos es reducido y deseamos reutilizar el trabajo que se ha podido hacer con otras bases de datos.

2.3.3. Data Augmentation

Para nosotros los seres humanos, puede ser muy fácil distinguir una persona sonriendo, aunque esta lleve gafas, tenga o no barba, etc. Pero para una Red Neuronal Convolutacional esto no es un hecho automático. Debemos enseñar al modelo, durante la fase de entrenamiento, todas estas posibles variaciones de una misma expresión facial. Debemos mostrarle gente con o sin barba, con o sin gafas, con diferentes condiciones lumínicas, etc. Es por ese motivo que el entrenamiento de una Red Neuronal Convolutacional requiere de una gran número de imágenes.

Como veremos en el Apartado 3.1.3, la base de datos que emplearemos para entrenar la red no contiene suficientes imágenes para representar todas estas variaciones de lo que podría ser una imagen de entrada para nuestra aplicación. Hay muchos factores que pueden variar incluso entre dos imágenes del mismo individuo con la misma expresión facial, como por ejemplo:

- Diferentes condiciones de luz.
- La rotación del rostro con respecto al eje vertical y horizontal de la imagen.
- Si el sujeto lleva algún tipo accesorio, como gafas o tiene vello facial.

Mientras que hay casos en los que no podemos hacer nada para solucionarlo que no sea obtener otras imágenes distintas, podemos simular algunas de estas variaciones aplicando transformaciones afines y combinaciones de estas a las imágenes de la base de datos, para obtener varias versiones de una misma imagen, que aunque realmente son linealmente dependientes, nos servirán para proporcionar mas variabilidad al modelo. Esta técnica que permite el aumento artificial del conjunto de datos es lo que se conoce como *Data Augmentation*.

Capítulo 3

Resolución del trabajo

Dado que el sistema que queremos construir está sujeto a experimentar, como método de ingeniería del software he empleado la técnica de modelo de prototipos rápido o también conocida como modelado de prototipado rápido[18].

3.1. Recursos

En esta sección enumeraremos los recursos que han sido utilizados para el desarrollo de este trabajo. Esto incluye recursos humanos como el autor y tutores, hardware y software utilizados así como las bases de datos.

3.1.1. Recursos Humanos

- **Autor:** Francisco José Fajardo Toril.
- **Tutor A:** Miguel García Silvente.
- **Tutor B:** Eugenio Aguirre Molina.

3.1.2. Recursos Hardware

Para el desarrollo de este trabajo he empleado mi ordenador personal cuyas especificaciones principales son:

CPU - AMD Phenom(tm) II X4 975

GPU - NVidia GTX 760

Memoria - 8GB RAM DDR3

3.1.3. Recursos Software

Para el desarrollo de la aplicación final, ha hecho falta usar distintas librerías que implementan de una forma eficiente las técnicas descritas en la sección anterior(2.3), así como distintos frameworks para desarrollar la interfaz de usuario por ejemplo. En la Figura 3.1 podemos observar los lenguajes de programación que han sido necesarios para la implementación de esta aplicación.

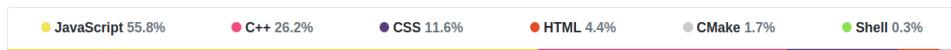


Figura 3.1: Lenguajes de programación que componen la aplicación[19].

OpenCV [22]

Por la facilidad de uso que me suponía debido a experiencia previa y su rapidez porque está implementada en C++ e usado esta librería para la manipulación general de imágenes.

Dlib [23]

Es una librería que implementa un gran número de algoritmos de Machine Learning en C++. En este caso ha sido usada sólo para el procesamiento de imágenes, pero solo he usado el detector de caras que implementa, el cual he introducido en la Sección 2.3, ya que presentaba menos falsos positivos que el detector de caras de OpenCV[12].

Caffe [24]

Es un framework de *Deep Learning* implementado en C++ aunque tiene una interfaz en Python. He escogido esta herramienta porque te permite construir tu propia red neuronal, definir tus propias capas de la arquitectura de red, posee buena documentación y buena comunidad. También puedes entrenar y testear tus modelos usando CPU o GPU, además de que ofrece herramientas que facilitan la creación de las bases de datos, todo esto desde la linea de comandos.

NVIDIA Deep Learning GPU Training System [25]

O también conocido como *NVIDIA DIGITS* es un sistema que funciona sobre *Caffe* y que ofrece una interfaz de usuario muy cómoda. Permite ahorrar mucho tiempo en la gestión de diferentes bases de datos, en la recolección de estadísticas de rendimiento de los modelos entrenados. Además

simplifica en gran medida la capacidad de entrenamiento con GPU, que mediante linea de comandos puede ser algo difícil de configurar.

Electron [26]

Este framework de código abierto permite crear aplicaciones de escritorio de forma nativa usando tecnologías web como son HTML, JavaScript y CSS. Esto nos permite compatibilidad con diferentes sistemas operativos y desarrollar una interfaz de usuario bonita de forma sencilla.

Base de datos: Yalefaces [27]

Esta base de datos la usé para aprender a manejar herramientas como Caffe y hacer pruebas de clasificación. Aunque no la he usado en la aplicación final, veo necesario hacer una mención a ella.

Se compone de 165 imágenes en escala de grises donde se fotografiaron 15 individuos. De cada uno de los individuos hay 11 imágenes, 1 por cada una de las siguientes expresiones básicas: *Happy, normal, sad, sleepy, surprised, y wink*, además de tener imágenes con luz lateral, central y con gafas tal y como se aprecia en la figura 3.2



Figura 3.2: Muestra de imágenes de la base de datos Yalefaces.

Base de datos: Karolinska Directed Emotional Faces (KDEF) [28]



Figura 3.3: Muestra de las imágenes que contiene la base de datos KDEF.

En este caso estamos ante una base de datos mucho más completa y preparada para el problema que queremos resolver. Esta, recoge imágenes de 70 individuos distintos tomadas desde 5 ángulos diferentes mostrando las 7 expresiones faciales que intentaremos predecir (Sección 2.1).

Se realizaron dos sesiones de fotos distintas a todos los individuos, lo que

hace un total de 4900 fotografías, en las que se usó una cuadrícula para que la posición de los ojos y la boca de todos los individuos estuviesen siempre en la misma posición. Una muestra de esta base de datos puede verse en la Figura 3.3.

Resumen

En la Figura 3.4 se puede ver una visión esquemática de la integración de todas estas tecnologías y en qué fase se usan cada una de ellas.

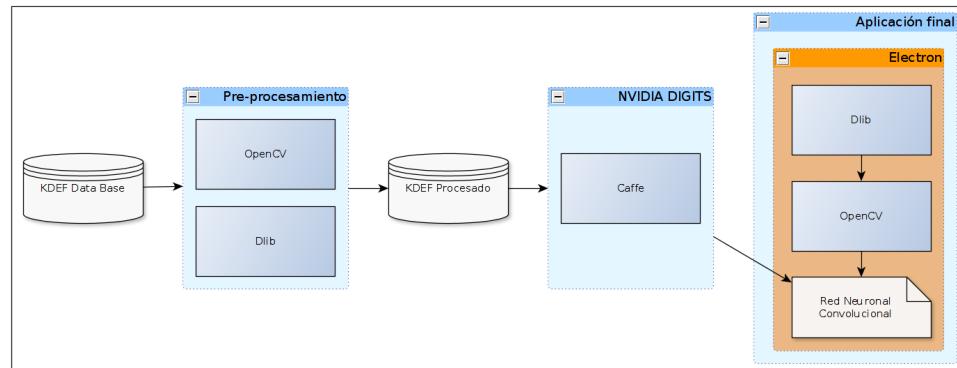


Figura 3.4: Esquema de integración de las distintas tecnologías usadas.

3.2. Especificación de requisitos

En este proyecto se partió de una especificación inicial de requisitos que se fue refinando a medida que se fueron implementando los distintos prototipos. Los requisitos **finales** son:

3.2.1. Requisitos del usuario

- RU1 El usuario puede seleccionar cualquier fichero de su computador como entrada para la aplicación.
- RU2 El usuario puede arrastrar cualquier fichero a la ventana de la aplicación para proporcionarle una entrada.

3.2.2. Requisitos funcionales

- RF1: La aplicación podrá recibir ficheros de entrada mediante el gestor de ficheros predeterminado del sistema operativo o bien arrastrando el fichero a la ventana principal.
- RF2: La aplicación solo tendrá en cuenta aquellos ficheros de entrada cuya extensión sean de formatos de imagen válidos: *.png*, *.jpeg*, *.jpg*.

- RF3: La aplicación solo analizará imágenes en las que se encuentren al menos un rostro humano.
- RF4: El único preprocesamiento que la aplicación realizará sobre la imagen de entrada, será recortar y reajustar el tamaño de la sección en la que se encuentra el rostro.
- RF5: Una vez clasificada, la aplicación mostrará por pantalla la cara detectada junto a los resultados de su clasificación.
- RF6: La interfaz notificará al usuario posibles errores mediante una ventana emergente.

3.2.3. Requisitos no funcionales

- RNF1: Para garantizar la validez del fichero de entrada, la aplicación aceptará solo *paths* de imágenes cuya extensión coincida con los formatos de imagen válidos: *png*, *jpeg*, *jpg*.
- RNF2: La aplicación usará el detector de caras de *Dlib* en la imagen de entrada para determinar si hay un rostro humano.
- RNF3: La ventana emergente de errores indicará:
 - RNF3.1: Cuando el fichero de entrada no tiene un formato de imagen válido.
 - RNF3.2: Cuando no se pudo detectar al menos un rostro en la imagen.
 - RNF3.3: Cuando no se seleccionó ningún fichero de entrada.
- RNF4: En caso de que la imagen sea válida, es decir en caso de localizar un rostro humano, la aplicación lo recortará y lo reajustará a un tamaño de $[256 \times 256]$ haciendo uso de *OpenCV*. Ese recorte se denominará *imagen intermedia*.
- RNF5: Para clasificar la *imagen intermedia* se usará un modelo basado en Redes Neuronales Convolucionales previamente entrenado.
- RNF6: Para mostrar los datos de la clasificación de la *imagen intermedia*, se dibujará un histograma que muestra el porcentaje que tiene la imagen intermedia de contener una expresión determinada. El histograma tendrá un máximo de 5 resultados y será dibujado usando HTML y JavaScript.
- RFN7: El histograma no mostrará por pantalla los resultados en los que el porcentaje de acierto es inferior al 1%.
- RFN8: El histograma mostrará una animación donde los porcentajes van creciendo hasta alcanzar su valor.

3.3. Planificación

En la tabla 3.3 están dispuestos, de forma aproximada, los tiempos estimados y empleados en cada uno de los principales sub-objetivos.

Como se puede observar, el prototipo I es el que más tiempo ha consumido. Esto es porque era el punto de inicio, lo que implica tener que empezar de cero, aprender conceptos teóricos sobre temas que solo conocía de oídas como por ejemplo las Redes Neuronales Convolucionales (Apartado 2.3.2) a usar nuevas tecnologías que las implementaban como Caffe (Apartado 3.1.3) además del resto como puede ser OpenCV (Apartado 3.1.3) mediante la realización de ejemplos para entenderlas de forma correcta. El prototipo I además no constaba con interfaz, se manejaba todo por línea de comandos y algunos scripts en shell bash.

Sub-objetivo	T. estimado	T. empleado por prototipo		
		I	II	III
Preparación de la información	~ 1 mes	~ 2 meses	~ 7 días	~ 5 días
Entrenamiento del clasificador	~ 2 meses	~ 3 meses	~ 1 mes	~ 7 días
Desarrollo de la interfaz	~ 2 meses	-	~ 1 mes	-

Cuadro 3.1: Planificación de tiempos sub-objetivos. Nota: I, II y III equivale al número de prototipo.

En el prototipo II, ya tenía asentados los conocimientos aprendidos mediante la implementación del primer prototipo, lo que agilizó el desarrollo del segundo prototipo. Este se basó en mejorar los resultados del entrenamiento de la Red Neuronal Convolucional mediante nuevas técnicas de preprocesado sobre las imágenes y haciendo test con distintos parámetros de entrenamiento, pero se reutilizaban muchas de las herramientas implementadas en el prototipo I. El descubrimiento de NVIDIA DIGITS (Apartado 3.1.3) también ayudó en gran medida a la reducción de este tiempo gracias al aprovechamiento de la GPU para la fase de entrenamiento. En este prototipo además implementé la interfaz, que llevó menos tiempo del esperado debido a la sencillez que aporta Electron (Apartado 3.1.3) para el desarrollo de interfaces.

En el último prototipo es una versión mejorada del segundo en el que se mejora la precisión del clasificador dejando la interfaz intacta.

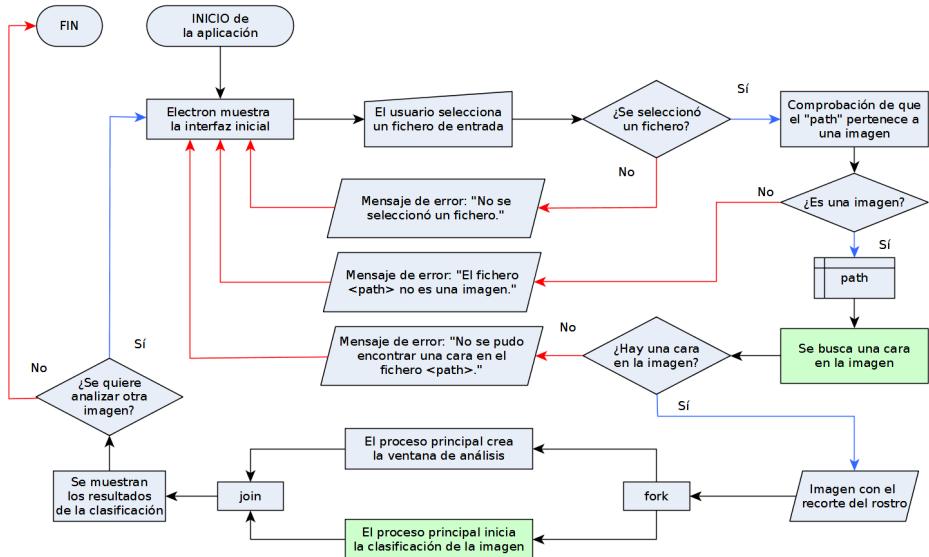


Figura 3.5: Diagrama de flujo de la aplicación final.

3.4. Análisis funcional

La aplicación tiene que ser capaz de obtener un fichero de entrada, comprobar si es válido y en caso afirmativo proceder a la clasificación de este usando el modelo previamente entrenado.

En la Figura 3.5 se ilustra el comportamiento de la aplicación final en función de los posibles casos que se pueden dar durante la ejecución. Los rectángulos verdes indican la ejecución de sub-procesos. El que se encuentra más arriba de los dos en la Figura 3.5, es el único que realiza algún tipo de preprocesamiento en la imagen que se quiere clasificar, y lo único que hace es generar el recorte de la cara. Este era uno de los principales objetivos como se comentó en el Apartado 1.1.2.

Aunque la inter-relación entre el usuario y la aplicación es muy simple, pueden darse varios casos de uso de la aplicación a la hora de proporcionar el fichero de entrada. Como se puede ver en la Figura 3.6 el usuario puede proporcionar el fichero mediante el uso del botón de selección presente en la interfaz, que abre el gestor de archivos por defecto del S.O. Si se da este caso, no haría falta comprobar que el fichero es una imagen porque el propio gestor te permite filtrar por tipos de archivo y solo te permite la selección de archivos con un formato de imagen válido.

El otro caso es que el usuario arrastre directamente el fichero a la ventana. En esta ocasión si sería necesario comprobar que el fichero tiene una extensión de archivo válida ya que ha podido arrastrar cualquier tipo de fichero. Como se puede ver la aplicación final no es demasiado compleja. La com-

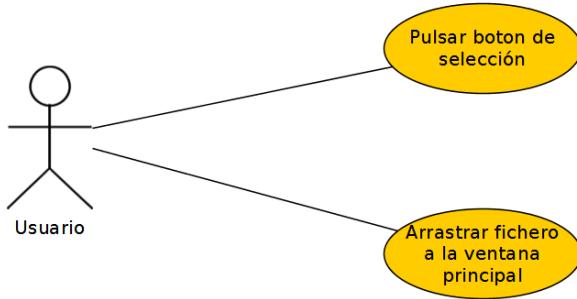


Figura 3.6: Casos de uso

plejidad de este trabajo reside en entrenar una Red Neuronal Convolucional para obtener la máxima precisión posible, tema que abordaremos de lleno en la siguiente sección.

3.5. Implementación y pruebas

Los lenguajes de programación que más he utilizado han sido *C++*, *HTML*, *CSS*, *JavaScript* y *Bash* como se puede aprecia en la Figura 3.1. Aunque este último ha sido empleado en gran parte durante la implementación de los primeros prototipos para su gestión de ficheros en el S.O. En el Apartado 3.1.3 se presentó cada una de ellas pero en esta sección hablaremos un poco más a fondo sobre su uso en la implementación de la aplicación. Si vemos la aplicación como un conjunto de capas, empezaremos a describir cómo están integradas estas tecnologías de arriba hacia abajo, siendo este el orden:

1. Interfaz
2. Módulo de clasificación.
3. Entrenamiento del clasificador.

3.5.1. Interfaz

HTML, *CSS* y *JavaScript* siempre han estado asociados al ámbito de las tecnologías web. Con estos lenguajes, se consiguen desarrollar interfaces de usuario de forma muy cómoda y rápida en el lado del cliente de forma que:

- *HTML* se usa para la estructura.
- *CSS* se usa para el aspecto.
- *JavaScript* se usa para la funcionalidad.

Por mi experiencia en este campo, sabía que si lograba integrar estos tres lenguajes con mi funcionalidad principal, implementada en C++, conseguiría tener una aplicación de escritorio que pudiese funcionar en diversos sistemas operativos de una forma rápida con un aspecto más que notable. Además tendremos la estructura de nuestra interfaz, su estilo y su funcionalidad de forma separada, por lo que, si queremos cambiar el aspecto de la misma, por ejemplo, simplemente habrá que cambiar la hoja de estilos CSS.

Es entonces cuando encontré **Electron** (Apartado 3.1.3), un framework **open source** que hacía posible integrar estos lenguajes asociados a las tecnologías web, HTML, CSS y JavaScript en una aplicación de escritorio, justo lo que buscaba.

Internamente Electron, crea un proceso principal que es el que tiene permisos para hacer llamadas al S.O. ya sea para lectura de ficheros, creación de directorios, etc. A su vez, crea sub-procesos que renderizarán las distintas ventanas de nuestra aplicación. El proceso principal y sus sub-procesos se comunican de forma asíncrona mediante el envío y la recepción de mensajes. De esta forma el proceso principal puede proporcionar información que requiera uno de sus sub-procesos y que este no pueda conseguir porque no tiene acceso a llamadas al S.O. o bien obtener resultados al ejecutar sub-módulos, como por ejemplo, los resultados de clasificar una imagen.

La interfaz en sí es concisa y no pretende ser complicada. Permite al usuario seleccionar un fichero mediante un botón de *selección de fichero*. Este es capturado por el proceso que renderiza la ventana de la interfaz, y le envía al proceso principal un mensaje asíncrono para que realice una llamada al S.O que invoca al gestor de archivos por defecto y le aplica unos filtros para permitir solamente que el usuario pueda seleccionar imágenes.

Una vez el proceso principal obtiene el path del archivo, este se pasa al módulo de clasificación para que realice las pertinentes operaciones con las imágenes, lo cual veremos en el siguiente apartado. Cuando el proceso principal obtiene los datos de salida del módulo de clasificación, este se los pasa al sub-proceso para renderizarlos en pantalla.

Hasta este punto nuestra aplicación estaría de la siguiente manera tal cual se ve en la Figura 3.7, una interfaz que recoge un path.

3.5.2. Módulo de clasificación

Desde un principio quería que la aplicación funcionara en torno a C++ ya que es uno de los lenguajes con mayor rendimiento, característica que es de interés si se está pensando en un sistema que trabaje en tiempo real en un futuro, además de ser el lenguaje que más he usado durante mi paso por la Universidad de Granada y con el que me sentía más cómodo.

Teniendo esto en cuenta, busqué herramientas que estuviesen implementadas en este lenguaje que me pudiesen ayudar a resolver el problema del entrenamiento de una red neuronal convolucional, clasificar una imagen, detectar

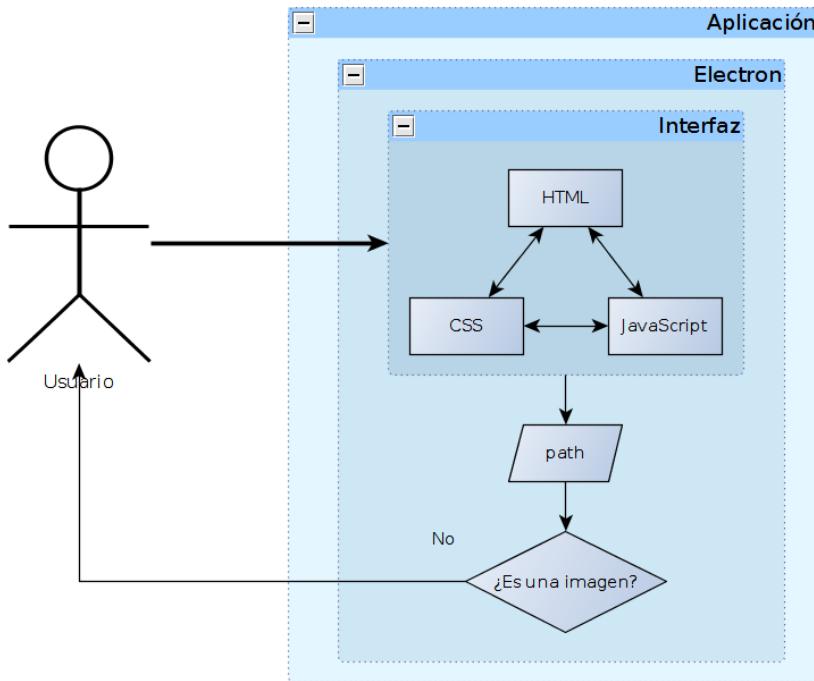


Figura 3.7: Composición parcial de la aplicación (1/3).

rostros en imágenes, etc. Este fue uno de los motivos por el que me decanté a usar herramientas como **OpenCV** (Apartado 3.1.3) o **Caffe** (Apartado 3.1.3) entre otras, ambas implementadas en C++, pero no el único.

La parte de la aplicación que se dedica al análisis y la gestión de la imagen de entrada es el Módulo de clasificación, y tal y como he mencionado antes, es conveniente que sea lo más rápida posible, así que está implementada en C++. El módulo de clasificación consta de dos sub-módulos:

1. El primero de ellos recibe el path de la imagen seleccionada por el usuario mediante la interfaz (Apartado 3.5.1) y comprueba que aparezca al menos una cara. En caso afirmativo, realiza un recorte de esa cara y lo guarda como una imagen de tamaño $[256 \times 256]$. Si no se encuentra una cara no se clasifica la imagen.

Esta herramienta está implementada en C++ y usa OpenCV y Dlib (Apartado 3.1.3). Ambas librerías son open source. Podría haberse usado solamente OpenCV ya que también incluye un detector pero el que implementa Dlib (Apartado 2.3.1) genera menos falsos positivos [12].

2. El segundo sub-módulo es el que se encarga de clasificar el recorte del rostro obtenido antes. Para ello hemos tenido que entrenar y generar

previamente un modelo basado en Redes Neuronales Convolucionales que sea capaz de dicha tarea. Aquí es donde entra en juego Caffe (Apartado 3.1.3).

Como ya mencioné es un framework de deep learning, open source implementado en C++ que nos permite definir nuestra propia arquitectura de red, nuestras propias capas, realizar entrenamientos tanto en CPU como en GPU y ofrece un gran conjunto de herramientas, como por ejemplo, dado un modelo entrenado, generar una clasificación ante una información de entrada. Esto es justo este segundo sub-módulo.

El estado de la aplicación en este momento es el que se muestra en la Figura 3.8 pero nos sigue faltando lo más importante, el clasificador que veremos cómo ha sido obtenido en la siguiente sección.

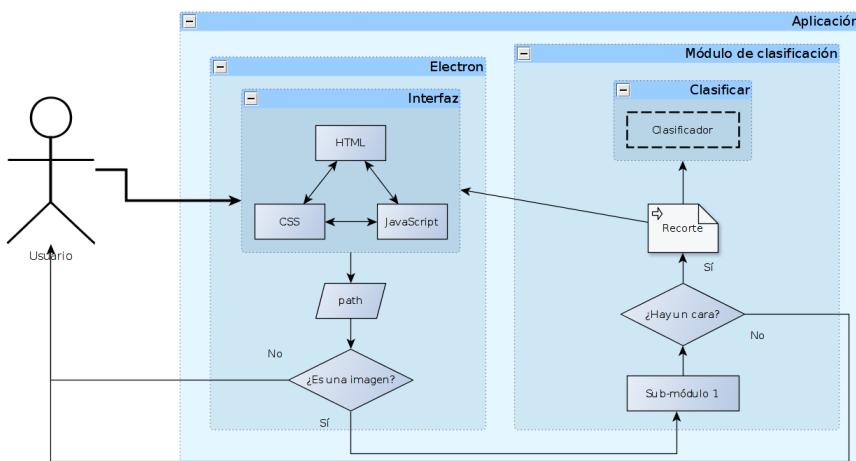


Figura 3.8: Aplicación con interfaz y módulo de clasificación (2/3).

3.5.3. Entrenamiento del clasificador

Para que el módulo de clasificación funcione correctamente debemos entrenar un clasificador basado en redes neuronales convolucionales. En esta ocasión podríamos usar Caffe por si solo, pero decidí usar NVIDIA DIGITS (Apartado 3.1.3) porque simplifica el proceso permitiéndonos centrarnos en lo que realmente importa, la fase de entrenamiento. NVIDIA DIGITS funciona sobre Caffe, aunque también soporta otros frameworks, y ofrece una interfaz sencilla e intuitiva que permite gestionar todos los modelos, todos los conjuntos de datos y recoge de forma automática métricas útiles durante el entrenamiento.

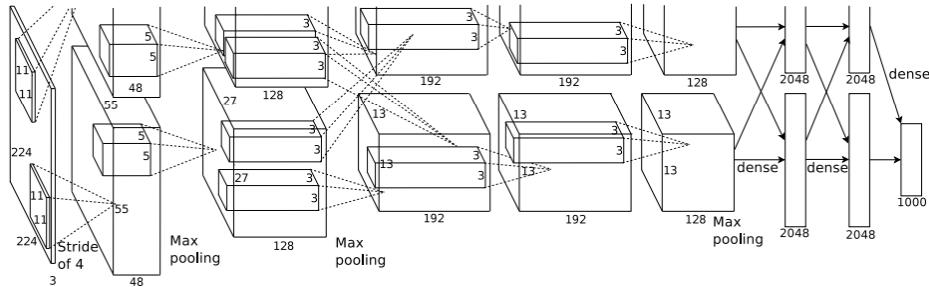


Figura 3.9: AlexNet[11]

Arquitectura de Red

El primer paso para entrenar un modelo es comenzar escogiendo o formando una arquitectura de red. En mi caso, he escogido AlexNet[11] (Figura 3.9), conocida por su buen rendimiento en el concurso de ImageNet LSVRC-2010 que consistía en clasificar 1, 2M de imágenes entre 1000 clases diferentes. Contiene 8 capas de aprendizaje, distribuidas como 5 capas de convolución y 3 capas totalmente conectadas (Apartado 2.3.2).

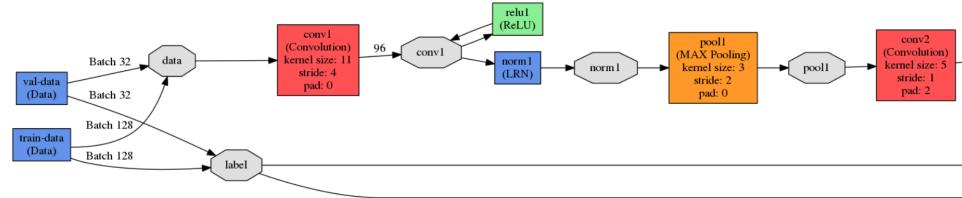


Figura 3.10: Primeras capas de AlexNet

En la Figura 3.10 se puede observar el primer conjunto de capas de nuestra arquitectura. Vemos como la primera capa obtiene los datos de entrada, *validation* y *training*, que llegan a la primera capa de convolución. Concretamente en esta capa, se puede observar que se obtienen 96 filtros de salida, calculados sobre las imágenes de entrada con un tamaño del kernel de convolución de 11×11 y un stride de 4.

A la salida de la capa de convolución se le añade no-linealidad aplicándole la función Rectified Linear Units o ReLu (ver Apartado 2.3.2). En las redes neuronales convencionales se añadía no-linealidad usando la función:

$$f(x) = \tanh(x) \text{ o bien } f(x) = (1 + e^{-x})^{-1}$$

En esta arquitectura se usa la función ReLu con el objetivo de aumentar la velocidad de entrenamiento sin sacrificar demasiado en rendimiento del

clasificador. Acto seguido se normalizan los datos mediante Local Response Normalization o LRN de las 2 primeras capas de convolución, ya que se ha probado que el modelo gana en generalización[11], y se le aplica una capa de submuestreo. Este es el esquema general de la arquitectura de red para las capas de convolución.

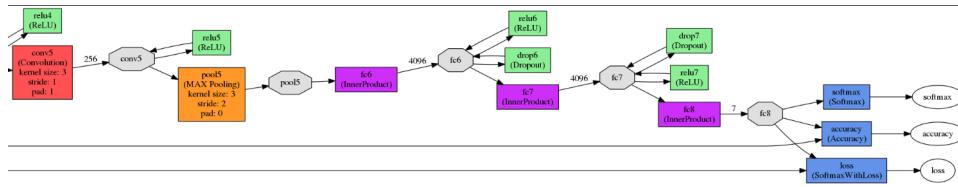


Figura 3.11: Últimas capas de AlexNet

Luego se conecta lá última capa de convolución con la primera de las tres capas totalmente conectadas, como se puede ver en la Figura 3.11, para comenzar a calcular los scores de cada clase y finalmente obtener la etiqueta de salida. Para reducir el sobreajuste en las capas totalmente conectadas se aplica una capa de Dropout una técnica que permite mejorar el rendimiento del clasificador que consiste en poner a cero la salida de cada una de las neuronas ocultas con una probabilidad del 50 %, para que no participen en el proceso de *forward pass* y de *back propagation*[11].

Base de datos

Una vez se tiene una arquitectura de red, debemos conseguir una base de datos con la que poder entrenarla. En mi caso, y como ya mencioné en el Apartado 3.1.3 de Recursos, la base de datos que he podido conseguir es: *Karolinska Directed Emotional Faces (KDEF)* [28].

Para cada entrenamiento debemos dividir esa base de datos en dos conjuntos a los que llamaremos:

- **Conjunto de entrenamiento o de training:** Es el conjunto que se usa para el aprendizaje de la red neuronal.
- **Conjunto de validación:** Es el conjunto que se usa para verificar si la red neuronal está aprendiendo o no.

Por último decir que para medir el rendimiento real del clasificador es necesario otro conjunto de datos externo a la base de datos. A este conjunto lo llamaremos **conjunto de test**.

3.6. Fase de entrenamiento

En este apartado veremos las distintas pruebas que he realizado, aunque sólo mencionaré las evoluciones más importantes debido al gran número de experimentos que he realizado. Véase que el conjunto de imágenes de test siempre será de un total de 140 imágenes escogidas al azar que no tienen nada que ver con la base de datos empleada para el entrenamiento.

3.6.1. Experimento N°1

Este primer experimento es una toma de contacto para probar la herramienta NVIDIA DIGITS por lo que el conjunto de imágenes es muy pequeño.

Base de datos

Para este primer experimento he cogido simplemente las caras frontales de la base de datos, dejando a un lado las laterales (Figura 3.12).

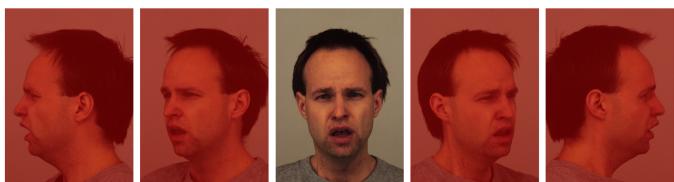


Figura 3.12: Ángulos seleccionados de KDEF por cada sujeto. (Los ángulos de las imágenes en rojo se han ignorado.)

Preprocesamiento

Para cada una de las imágenes escogidas de la base de datos:

- He recortado el mismo rectángulo a todas las imágenes para obtener un recorte de los rostros centrado, ya que los sujetos que aparecen en la base de datos KDEF tienen los ojos en las mismas coordenadas.
- He reajustado la imagen resultante del recorte a un tamaño de 256×256 .

Con esto, he obtenido una batería de imágenes como las de la Figura 3.13. Tras el preprocesamiento, se ha separado el conjunto de imágenes en los dos subconjuntos mencionados anteriormente, training y validación. El tamaño de ambos conjuntos se muestra en la tabla 3.2.

Véase que los subconjuntos de training y validación **deben estar balanceados** para que los resultados del entrenamiento tengan validez. En este caso el conjunto de training tiene 105 imágenes por cada clase mientras que el conjunto de validación tiene 35 imágenes.



Figura 3.13: Muestra de imágenes empleadas para el primer experimento de entrenamiento.

	Training	Validación (25 %)	Total
Nº de imágenes	735	245	980

Cuadro 3.2: DB empleada - Primer experimento

Parámetros de entrenamiento

- Epochs: 100
- Coeficiente de aprendizaje inicial: 0.02
- Policy: Sigmoid (Figura 3.15)

Resultados del entrenamiento

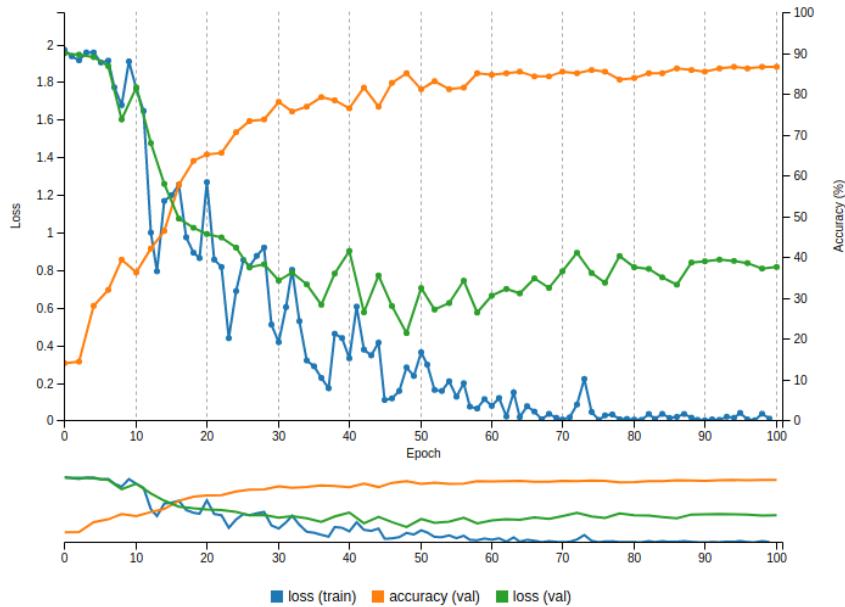


Figura 3.14: Resultados del primer entramiento.

Como podemos ver en los resultados Figura 3.14, con el paso de los epochs o épocas, la precisión del clasificador ha ido mejorando. Esto se

puede ver en como la línea naranja tiene una tendencia positiva. También podemos observar como a partir de los 70 epochs aproximadamente, el modelo converge y no mejora.

La función de pérdida (loss) tanto para training como para validación comienza disminuyendo de forma exponencial en las primeras épocas hasta que se estabiliza y no oscila tanto como al principio. Esto es debido a que el coeficiente de entrenamiento va disminuyendo también (Figura 3.15), por lo que el modelo tiende a converger. Podemos interpretar esta función de pérdida como *cúanto le queda por aprender a nuestro modelo* ya sea del conjunto de training (línea azul) o validación (línea verde).

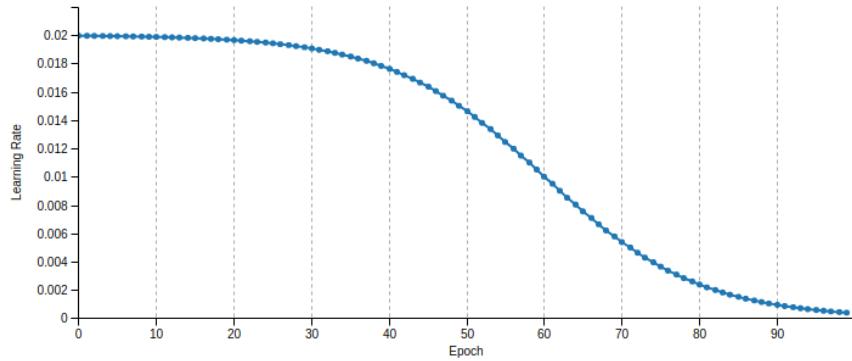


Figura 3.15: Política de disminución del coeficiente de aprendizaje.

También podemos observar en la Figura 3.14 como a partir de la mitad del entrenamiento, la función de pérdida del conjunto de validación cambia su tendencia descendente para comenzar a aumentar. Esto puede significar que estamos empezando a sobreajustar el modelo porque la función de pérdida para training es prácticamente 0.

Para medir el rendimiento real del clasificador, debemos clasificar el conjunto de test y cuantificar los aciertos y los errores que se han cometido. Esto se conoce como calcular la matriz de confusión.

	afraid	angry	disgusted	happy	neutral	sad	surprised	per-class
afraid	13	1	1	0	0	4	1	65,00 %
angry	17	1	0	0	1	1	0	5,00 %
disgusted	9	1	0	0	0	10	0	0,00 %
happy	9	1	0	3	2	5	0	15,00 %
neutral	10	0	0	0	3	7	0	15,00 %
sad	8	0	1	0	1	8	2	40,00 %
surprised	16	0	0	0	0	0	4	20,00 %

Cuadro 3.3: Matriz de confusión

Tras clasificar el conjunto de test, la matriz de confusión de este primer modelo es la que se muestra en la tabla 3.3. Podemos ver en la matriz como los aciertos ocuparán las casillas en gris, el resto son fallos. Como vemos el clasificador tiene un rendimiento muy pobre y ha obtenido un porcentaje de acierto en el conjunto de test del 22,86 % pese a tener un 87,00 % de acierto con el conjunto de validación, con solo una de las clases por encima del 50 % de acierto. Era de esperar por:

- El reducido número de imágenes con el que hemos entrenado el clasificador.
- Con el recorte del mismo rectángulo para todas las imágenes, hemos supuesto que todas las futuras imágenes de entrada, tendrán los ojos siempre en las mismas coordenadas.

		Actual Class	
		Afraid	Non-Afraid
Predicted Class	Afraid	13	69
	Non-Afraid	7	51

Cuadro 3.4: Matriz de confusión de la expresión **Afraid**

Un ejemplo de la matriz de confusión de la expresión Afraid (Tabla 3.4) se ve que el número de imágenes del conjunto de test que han sido clasificadas como Afraid cuando en realidad no lo son. Este hecho se conoce como *falsos positivos*, y en este caso hay 69 ocurrencias, lo que hace a esta clase tener un 57,50 % de probabilidad de falso positivo, un valor muy alto. En la siguiente sección veremos si se soluciona.

3.6.2. Experimento N°2

Puesto que el problema aparente del pobre rendimiento del primer experimento han sido el escaso número de imágenes y su recorte constante para todas las imágenes, en este segundo experimento vamos a intentar subsanar eso.

Base de datos

He seleccionado no sólo las imágenes frontales, como en el primer experimento, si no también las imágenes laterales y he ignorado las de perfil (Figura 3.16). Esto supone un total de 2940 imágenes.



Figura 3.16: Ángulos seleccionados de KDEF por cada sujeto. (Los ángulos de las imágenes en rojo se han ignorado.)

Preprocesamiento

En esta ocasión y a diferencia del primer experimento, he intentado entrenar el clasificador, recreando las condiciones que se pueden presentar en una imagen de entrada de la aplicación. Tal y como se explicó en el Apartado 3.5.2 , la aplicación usa un detector de caras para crear un recorte y este es el que se le da como entrada al clasificador. Por tanto, para cada una de las imágenes seleccionadas de la base de datos:

1. Se crean dos nuevas imágenes aplicando un cambio de perspectiva. Para ello calcula la matriz de transformación entre el rectángulo original y en el que se quiere proyectar la imagen. Figura 3.17
2. Se utiliza el mismo detector facial (Apartado 2.3.1) que usa la aplicación para obtener un recorte del rostro. En la imagen 3.18 se puede ver una comparativa de ambos métodos.
3. A partir de las imágenes anteriores, se rotan de forma aleatoria entre 10° y 20° para crear una nueva imagen rotada. Usamos el reflejo de los bordes en lugar de un valor constante para los casos en los que hay que llenar la imagen rotada. De esta forma añadimos información de la propia imagen, aunque sea repetida. Puede verse en la imagen superior derecha de la Figura 3.19.



Figura 3.17: Cambio de perspectiva artificial. La imagen central es la imagen original y las otras dos se han generado de forma artificial.



Figura 3.18: **Izquierda:** Recorte utilizando el detector facial. **Derecha:** Recorte usando el rectángulo del primer experimento.

4. Para cada una de las imágenes generadas en los puntos 1, 2 y 3, se modifica la intensidad del valor de los píxeles de forma que se suma o se resta una cantidad aleatoria, para generar dos nuevas imágenes, una más clara y otra más oscura. El valor aleatorio puede oscilar entre 25 y 65 unidades. Una muestra de esto se puede ver en la Figura 3.20.
5. A partir de las imágenes anteriores, creo nuevas aplicando zooms aleatorios, entre un 10 % y un 15 % más cerca. En la Figura 3.21 se puede ver un ejemplo de esto.
6. Por último, ante la situación de que la cara detectada fuese más pequeña que el tamaño de las imágenes que se clasifican, 256×256 , al reajustar su tamaño a esas mismas dimensiones, la imagen que recibe el clasificador estaría pixelada. Es por esto que también simulé pixelados reajustando cada una de las imágenes generadas anteriormente. Para ello, pasé cada una de las imágenes de tamaño 256×256 a tamaños 40×40 y 80×80 obteniendo dos imágenes nuevas. Ahora de nuevo las reajusté a 256×256 usando un método de interpolación aleatorio de entre todos los que ofrece OpenCV[21]. El resultado es el



Figura 3.19: Imágenes no rotadas a la izquierda. Rotaciones aleatorias a la derecha.

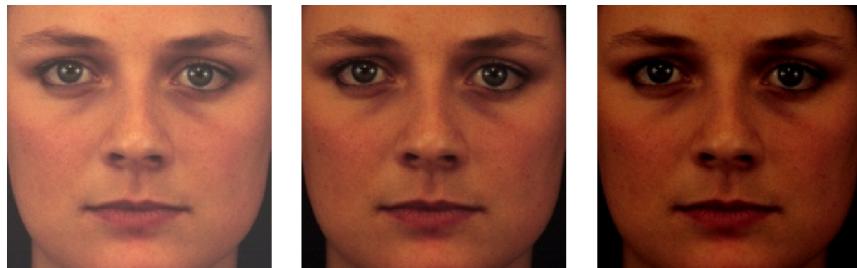


Figura 3.20: De izquierda a derecha: Imagen iluminada - Imagen original - Imagen oscurecida.

que se puede ver en la Figura 3.22.

La finalidad de estas transformaciones es aportar generalidad (Figura 3.23) al clasificador, ya que la imagen de entrada de la aplicación no siempre tendrá las coordenadas de los ojos en la misma posición, las mismas condiciones de luz, inclinación, etc.

Todas estas transformaciones han incrementado nuestro conjunto de datos de manera notable, obteniendo el conteo para los conjuntos de training y validación de la Tabla 3.5.

Parámetros de entrenamiento

- Epochs: 10



Figura 3.21: Izquierda: Imagen con zoom aleatorio. Derecha: Imagen original.



Figura 3.22: De izquierda a derecha: **1^a** Imagen original - **2^a** Imagen reajustada de $[80 \times 80]$ $\rightarrow [256 \times 256]$ - **3^a** Imagen reajustada de $[40 \times 40]$ $\rightarrow [256 \times 256]$.

- Coeficiente de aprendizaje inicial: 0.01
- Policy: Step Down (Figura 3.25)

Resultados del entrenamiento

En la Figura 3.24 podemos observar los resultados de este segundo entrenamiento. Vemos como a partir del epoch 5 el entrenamiento del modelo converge. También podemos apreciar como se minimiza la función de pérdida (loss) tanto de entrenamiento como de validación en los 2 primeros epochs. A pesar de que hemos reducido el coeficiente de aprendizaje inicial con respecto a al primer entrenamiento, la red ha necesitado muy pocos epochs para minimizar esta función.

Este hecho tiene sentido porque recordemos que se cuenta un epoch cada vez que la red le da una pasada a todas las imágenes. Puesto que hay muchas imágenes y son todas similares, pese a las transformaciones que hemos realizado en el pre-procesamiento, necesita menos epochs para aprender la mayoría de sus características.

En esta ocasión la precisión obtenida con el conjunto de validación es del

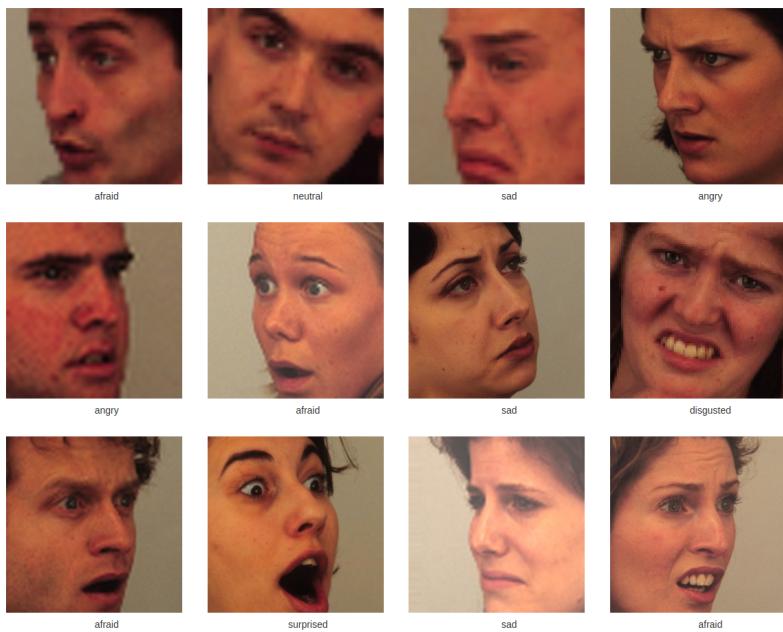


Figura 3.23: Muestra de imágenes de la base de datos tras las transformaciones.

	Training	Validación (25 %)	Total
Nº de imágenes	78384	27108	105492

Cuadro 3.5: DB empleada - Segundo experimento

85 % frente al 87 % del primer experimento. ¿Ha disminuido el rendimiento del clasificador?

Cuando clasificamos el mismo conjunto de test del primer experimento con el clasificador entrenado en este segundo experimento, obtenemos la matriz de confusión de la Tabla 3.6.

Si comparamos esta tabla con la matriz de confusión del primer experimento (Tabla 3.3) la ganancia es más que notoria. En este caso la diagonal está mucho más llena de aciertos y el clasificador no erra la mayor parte del tiempo con los falsos positivos de la clase Afraid, como se vió en los resultados del primer entreno. Además podemos ver como el clasificador no sabe distinguir la expresión *Angry* y *Sad*, que son asociadas a la clase *Disgusted*. La mejoría en el resto de clases se traduce en un 44,29 % de acierto con el conjunto de test frente al 22,86 % del experimento anterior, por lo que podemos decir que las técnicas de preprocesamiento han sido **positivas** en este caso.

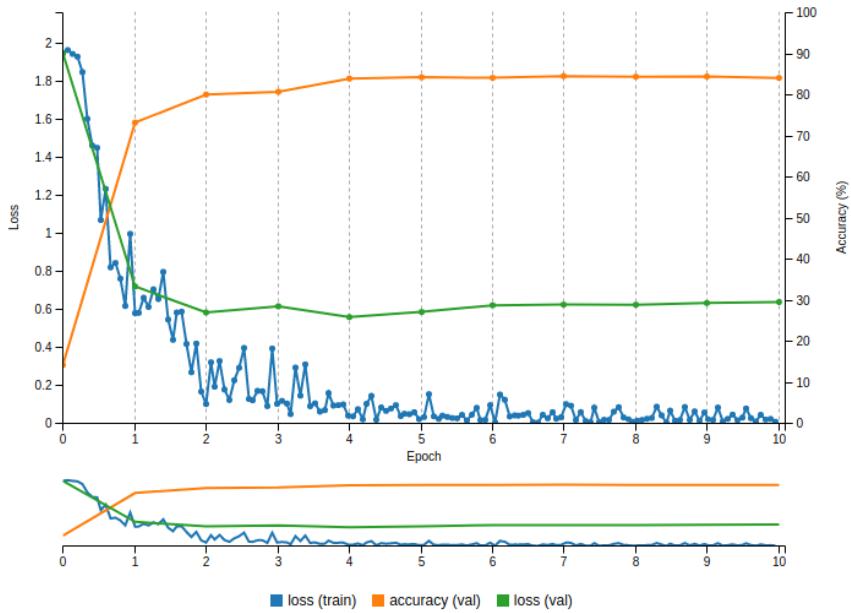


Figura 3.24: Resultados del segundo entrenamiento.

	afraid	angry	disgusted	happy	neutral	sad	surprised	per-class
afraid	11	1	4	2	0	2	0	55,00 %
angry	1	3	15	0	0	0	1	15,00 %
disgusted	2	2	13	1	1	1	0	65,00 %
happy	0	0	5	14	0	0	1	75,00 %
neutral	1	3	4	0	10	2	0	50,00 %
sad	3	4	8	0	1	4	0	20,00 %
surprised	5	1	2	2	3	0	7	35,00 %

Cuadro 3.6: Matriz de confusión

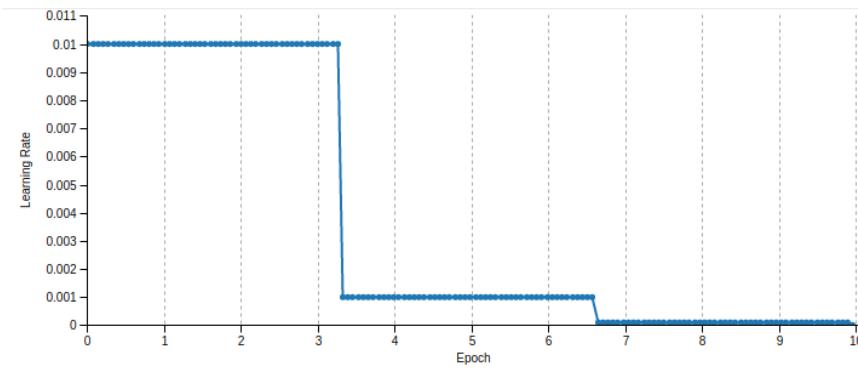


Figura 3.25: Política de disminución del coeficiente de aprendizaje.

3.6.3. Experimento N°3

Viendo la mejoría de rendimiento tras el proceso de pre-procesado del segundo experimento, en este tercero intentaremos producir muchas más imágenes de la misma manera.

Base de datos

Para ello se han tenido en cuenta la misma batería de imágenes de la base de datos que en el segundo experimento (Véase la Figura 3.16).

Preprocesamiento

Se ha aplicado el mismo pre-procesado que en el segundo experimento, pero para cada una de las transformaciones se han producido un mayor número de imágenes. Por ejemplo, a la hora de generar los pixelados, por cada imagen, se usan todos los métodos de interpolación disponibles en lugar de un aleatorio. La idea es intentar obtener aun más generalidad. Esto provoca, que el tamaño de los conjuntos de validación y training aumenten tal y como se ve en la Tabla 3.7.

	Training	Validación (25 %)	Total
Nº de imágenes	352998	121824	474822

Cuadro 3.7: DB empleada - Tercer experimento

Parámetros de entrenamiento

- Epochs: 3
- Coeficiente de aprendizaje inicial: 0.02
- Policy: Inverse Decay (Figura 3.27)

Resultados del entrenamiento

La Figura 3.26 ilustra los resultados de este tercer entrenamiento. Sólo se han hecho falta 3 epochs para que el modelo converja. Incluso, a partir del primero, da la sensación de que el entreno comienza sobre-ajustar el modelo dado que la función de pérdida de validación comienza a tener una tendencia ascendente en lugar de descendente.

Tras varios intentos con este conjunto de datos, he conseguido los mejores resultados con una política de disminución del coeficiente de aprendizaje de Inverse Decay como se ve en la Figura 3.27.

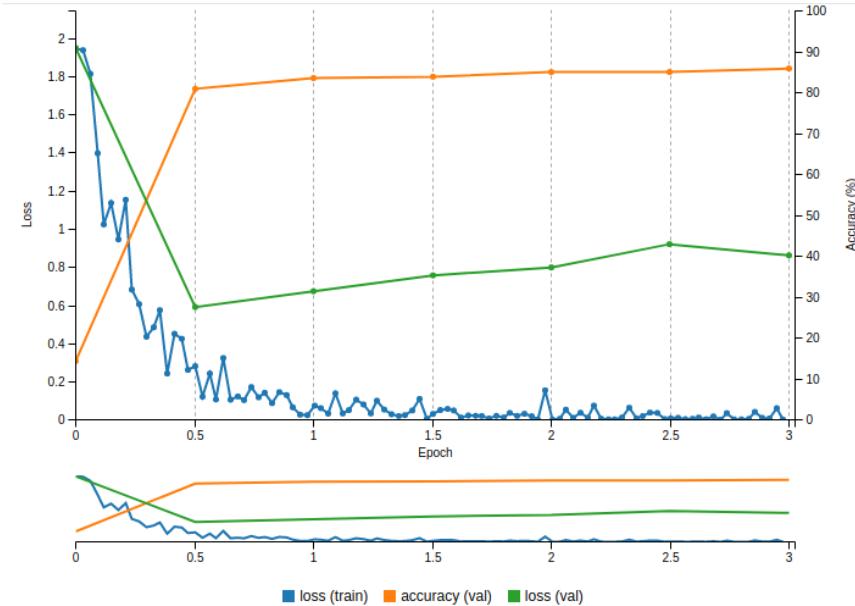


Figura 3.26: Resultados del tercer entrenamiento.

El rendimiento de este clasificador con el conjunto de validación es del 86 % de acierto. Al clasificar el conjunto de test obtenemos un 51,43 % de acierto en este tercer experimento frente al 44,29 % del primero. Una mejoría sustancial del 7,14 % pese a mi sospecha acerca del sobre-ajuste en el entrenamiento.

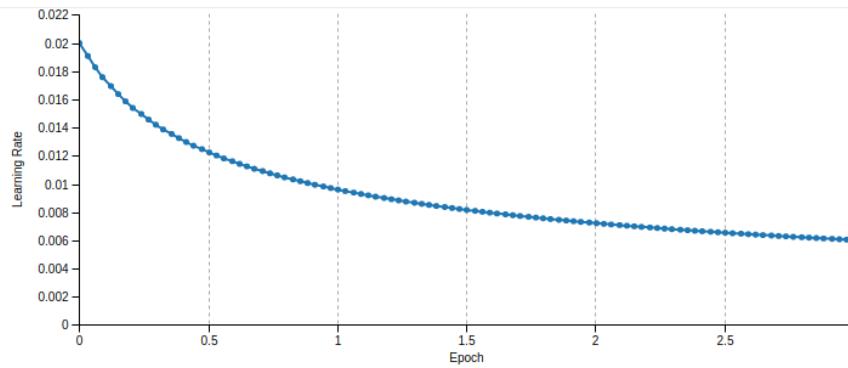


Figura 3.27: Política de disminución del coeficiente de aprendizaje.

La matriz de confusión de este experimento puede verse en la Tabla 3.8. Si comparamos estos datos con los datos de la matriz de confusión del se ha logrado mantener y mejorar el acierto de todas las clases. Sobre todo en casos como los de la clase Angry que ha visto su rendimiento mejorado del 15,00 % al 30,00 % y Surprised del 35,00 % al 50,00 %. La clase Sad es la que

peor rendimiento sigue teniendo pese a haber mejorado de un 20,00 % a un 25,00 %.

	afraid	angry	disgusted	happy	neutral	sad	surprised	per-class
afraid	13	1	2	2	2	0	0	65,00 %
angry	4	6	9	0	0	1	0	30,00 %
disgusted	1	3	13	1	0	2	0	65,00 %
happy	0	0	4	14	1	0	1	70,00 %
neutral	2	3	1	0	11	3	0	55,00 %
sad	1	6	5	0	2	5	1	25,00 %
surprised	4	1	1	1	0	3	10	50,00 %

Cuadro 3.8: Matriz de confusión

3.6.4. Experimento N°4

El clasificador va mejorando. En este nuevo experimento voy a cambiar el método de pre-procesamiento que se ha usado en el segundo y tercer experimento, porque como se ha visto en el tercero, aumentar el conjunto de imágenes de forma metódica no va a mejorar mucho más el rendimiento del clasificador.

Base de datos

Se escogerá la misma batería de imágenes de la base de datos que en el segundo y tercer experimento (Véase la Figura 3.16).

Preprocesamiento

Para cada una de las imágenes escogidas de la base de datos se generan N nuevas imágenes. Cada una de esas imágenes es una combinación aleatoria de las siguientes transformaciones:

- Flip horizontal.
- Zooks aleatorios.
- Cambios de perspectiva aleatorios.
- Ruido Gaussiano aleatorio.
- Añadir iluminación o bien oscurecer la imagen.
- Rotaciones aleatorias.

Para aplicar una combinación aleatoria de las transformaciones anteriores en un orden aleatorio, se aplicará el Algoritmo 1. Para generar N imágenes aleatorias se aplicara N veces sobre la misma imagen.

Data: Imagen n-ésima

Result: Imagen de salida transformada de forma aleatoria

```
initialization;  
img ← Imagen de entrada;  
t ← Número aleatorio en el intervalo [3,6];  
indices[ ] ← {1, 2, 3, 4, 5, 6};  
shuffle(indices[ ]);  
for i ← 0 to t do  
| n ← indices[i];  
| img ← AplicarTransformacion(n);  
end
```

Algorithm 1: Cómo se aplican transformaciones aleatorias

Una vez se tienen las N imágenes, se detecta el rostro en ellas y se guardan para el conjunto final junto a sus versiones pixeladas, como se hizo en el entrenamiento 2. Podemos ver una muestra de estas imágenes aleatorias en la Figura 3.28. Podemos distinguir las rotaciones, la iluminación, el ruido Gaussiano, cambios de perspectiva, etc, todo junto en una misma imagen y con resultados diferentes. También se pueden ver imágenes originales porque por supuesto se usan imágenes sin transformaciones.

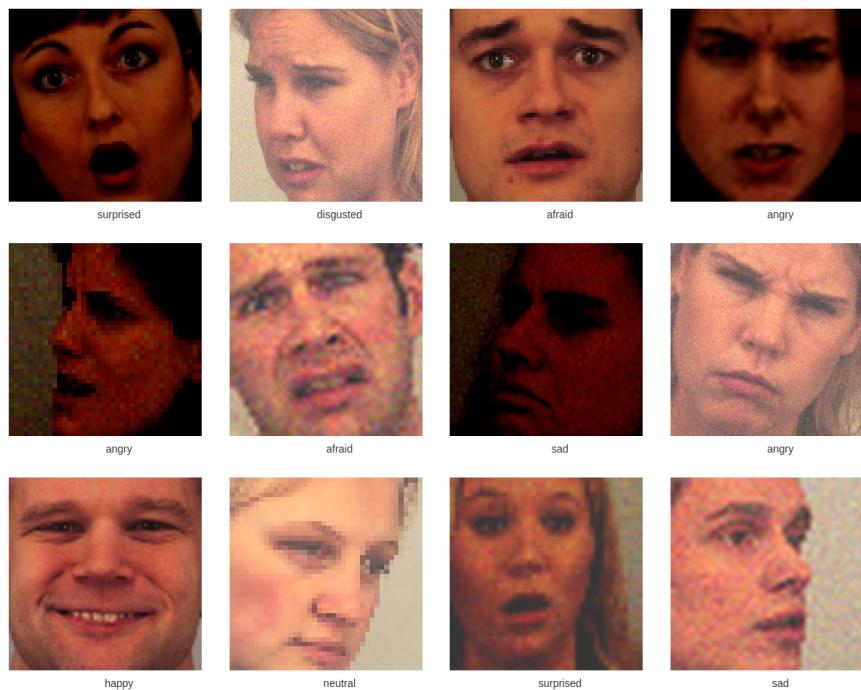


Figura 3.28: Muestra de imágenes transformadas de forma aleatoria.

Con estas transformaciones obtenemos un conjunto de datos del tamaño que se muestra en la Tabla 3.9.

	Training	Validación (25 %)	Total
Nº de imágenes	60189	20931	81120

Cuadro 3.9: DB empleada - Cuarto experimento

Parámetros de entrenamiento

- Epochs: 10
- Coeficiente de aprendizaje inicial: 0.01

- Policy: Step Down (Figura 3.30)

Resultados del entrenamiento

Pese a tener un número de imágenes similar al del segundo experimento (Tabla 3.5), el modelo tarda más epochs en converger. Esto se puede deber a la mayor variabilidad que hemos introducido con las transformaciones aleatorias por lo que tarda más en poder minimizar las funciones de pérdida. También vemos como la función de pérdida de validación se encuentra cerca de la función de pérdida de training, por lo que, en principio, esto indica que no hemos sobre-ajustado el modelo.

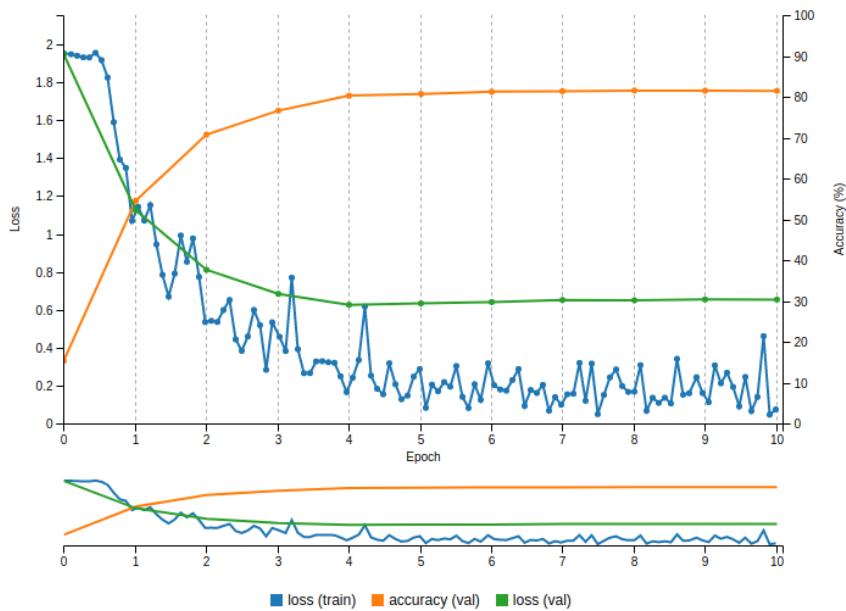


Figura 3.29: Resultados del cuarto entrenamiento.

Este entrenamiento ha alcanzado un 82 % de precisión sobre el conjunto de validación. Al clasificar el conjunto de test he obtenido un 53,57 %, lo que supone un aumento del 2,14 % de precisión usando aproximadamente una sexta parte del número de imágenes que se usó en el tercer experimento. Por lo que podemos afirmar que esta técnica generaliza mejor que la de ese tercer experimento.

La matriz de confusión de este entrenamiento se muestra en la Tabla 3.10 y es similar a la del tercer experimento. Se mantienen las deficiencias y las fortalezas del entreno anterior. La clase Angry y Sad siguen siendo las peores.

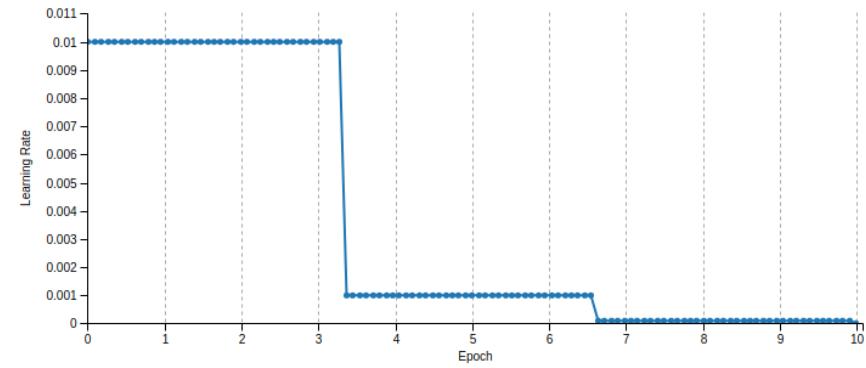


Figura 3.30: Política de disminución del coeficiente de aprendizaje.

	afraid	angry	disgusted	happy	neutral	sad	surprised	per-class
afraid	12	0	2	1	1	1	3	60,00 %
angry	1	6	9	3	0	0	1	30,00 %
disgusted	3	2	13	1	0	0	1	65,00 %
happy	0	0	3	16	0	0	1	80,00 %
neutral	3	3	1	0	12	1	0	60,00 %
sad	3	4	7	0	2	3	1	15,00 %
surprised	4	0	1	0	1	1	13	65,00 %

Cuadro 3.10: Matriz de confusión

3.6.5. Experimento N°5

Este quinto experimento fue una prueba para intentar unir los puntos fuertes de los modelos entrenados en el tercer y cuarto experimento. Para ello intenté, usando el conjunto de imágenes generado en el experimento N°4, entrenar un nuevo modelo, pero para la inicialización de la red usar los pesos de un modelo ya entrenado previamente. Más concretamente usé los pesos del modelo entrenado en el tercer experimento. Esta técnica se conoce como Transfer Learning y fue introducida en el Apartado 2.3.2.

Parámetros de entrenamiento

- Epochs: 5
- Coeficiente de aprendizaje inicial: 0.01
- Policy: Exponencial Decay (Figura 3.32)

Resultados del entrenamiento

Si observamos la Figura 3.31 podemos ver que gracias a escoger los pesos de un modelo ya entrenado el modelo comienza con una precisión sobre el conjunto de training de más del 90 %.

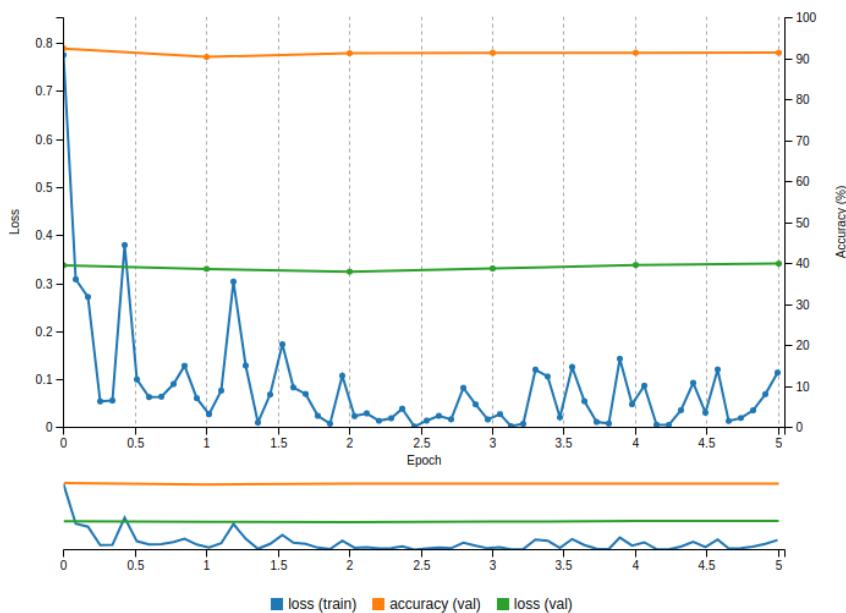


Figura 3.31: Resultados del quinto entramiento.

He escogido un coeficiente de aprendizaje con una disminución exponencial (Figura 3.32) porque interesa que el modelo aprenda lentamente, ya que

partimos de pesos ya calculados de una base de datos similar.

Vemos como la función de pérdida de entrenamiento fluctúa de bastante, pero a medida que va disminuyendo el coeficiente de aprendizaje las fluctuaciones son menores.

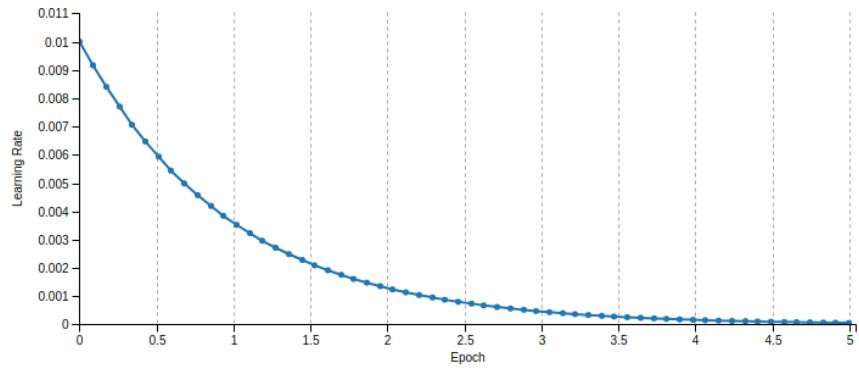


Figura 3.32: Política de disminución del coeficiente de aprendizaje.

Al clasificar el conjunto de test obtenemos un 54,29 % de precisión en el acierto lo que supone menos de un 1% de mejora. La matriz de confusión de la Tabla 3.11.

	afraid	angry	disgusted	happy	neutral	sad	surprised	per-class
afraid	12	1	3	0	1	1	2	60,00 %
angry	1	7	11	0	1	0	1	35,00 %
disgusted	1	3	14	1	0	1	0	70,00 %
happy	0	0	4	16	0	0	0	80,00 %
neutral	0	7	1	0	12	0	0	60,00 %
sad	1	3	9	0	3	4	0	20,00 %
surprised	3	1	1	0	1	3	11	55,00 %

Cuadro 3.11: Matriz de confusión

Pese a todos los experimentos, no he conseguido aumentar la precisión en expresiones como Angry y Sad, por lo que estoy pensando que quizás, las imágenes de estas dos clases de la base de datos no sean demasiado representativas de las imágenes de la misma clase en el conjunto de test.

Una vez tenemos el clasificador entrenado la aplicación final está completa ya que podemos mandar el recorte a este, clasificarlo con una de las herramientas que incluye Caffe y recoger los resultados para mostrarlos en la interfaz. Continuando con el esquema de la aplicación mostrado en la Figura 3.8, el gráfico final sería entonces el de la Figura 3.33.

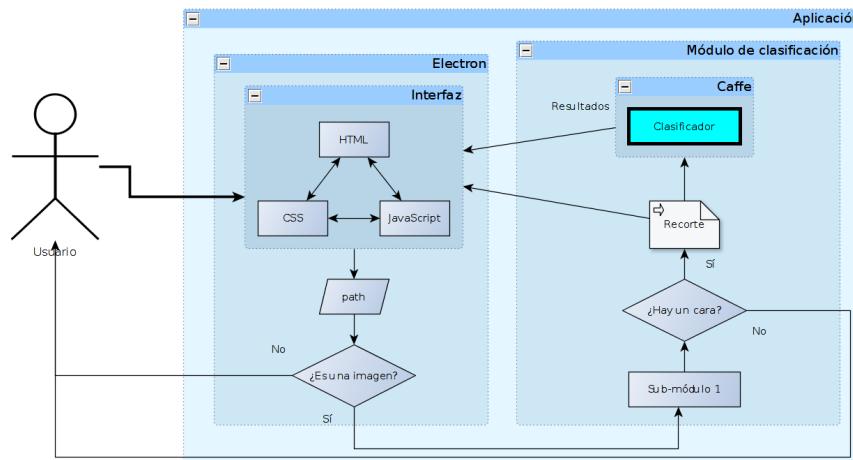


Figura 3.33: Gráfico de la aplicación final

3.7. Pruebas

Capítulo 4

Conclusiones y trabajo futuro

Decir lo que se ha conseguido realizar comentando sus puntos fuertes y débiles. Decir si se han alcanzado los objetivos específicos y el general propuesto y en qué grado.

Indicar las asignaturas del grado más relacionadas con la ejecución del TFG y cómo el TFG ha ayudado a afianzar los conocimientos adquiridos en el Grado.

Valoración personal si se quiere.

Avanzar algunas líneas de trabajo futuro para solucionar las debilidades detectadas o para conseguir nuevas funcionalidades interesantes.

Capítulo 5

Bibliografía

Bibliografía

5.1. Artículos científicos

- [1] R. Shbib, and S. Zhou. *Facial Expression Analysis using Active Shape Model*, School of Engineering, University of Portsmouth, United Kingdom 2015.
- [2] B. Abboud, F. Davoine, and M. Dang. *Facial expression recognition and synthesis based on an appearance model*, Heudiasyc Laboratory CNRS, University of Technology of Compiegne, BP 20529, 60205 Compiegne Cedex, France 2004.
- [3] T. F. Cootes, C. J. Taylor, D. H. Cooper, and J. Graham. *Active Shape Models - Their Training and Application*, Department of Medical Biophysics, University of Manchester, Oxford Road, Manchester M13 9PT, England 1994.
- [4] T. F. Cootes, G. J. Edwards, and C. J. Taylor. *Active Appearance Models*, Wolfson Image Analysis Unit, Department of Medical Biophysics, University of Manchester, Manchester M13 9PT, UK 1998.
- [5] J. Kumari, R. Rajesh, and KM. Pooja. *Facial expression recognition: A survey*, Dept. of Computer Science, Central University of South Bihar, India 2015.
- [6] N. Dalal, and B. Triggs. *Histograms of Oriented Gradients for Human Detection*, INRIA Rhône-Alps, 655 avenue de l'Europe, Montbonnot 38334, France 2005.
- [7] D. E. King. *Max-Margin Object Detection*, arXiv:1502.00046 [cs.CV], 2015.
- [8] C. Cortes, and V. Vapnik. *Support Vector Networks*, Kluwer Academic Publishers, Boston. Manufactured in The Netherlands, 1995.
- [9] Y. Bengio, A. Courville, and Pascal. Vincent. *Representation Learning: A Review and New Perspectives*, Department of computer science and

- operations research, U. Montreal also, Canadian Institute for Advanced Research (CIFAR), 2014.
- [10] V. Nair, and G. E. Hilton. *Rectified Linear Units Improve Restricted Boltzmann Machines*, Department of Computer Science, University of Toronto, Toronto, ON M5S 2G4, Canada 2010.
 - [11] NIPS2012_4824, Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E. *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 25, by F. Pereira and C. J. C. Burges and L. Bottou and K. Q. Weinberger, pages 1097-1105,Curran Associates, Inc. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> 2012.
- ## 5.2. URLs
- [12] D. King. *Dlib 18.6 released: Make your own object detector!*, URL: <http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html>, 2014.
 - [13] A. Moujahid. *A Practical Introduction to Deep Learning with Caffe and Python*, URL: <http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/>, 2016.
 - [14] A. Deshpande. *A Beginner's Guide To Understanding Convolutional Neural Networks*, URL: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>, 2016.
 - [15] A. Deshpande. *A Beginner's Guide To Understanding Convolutional Neural Networks - Part 2*, URL: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/>, 2016.
 - [16] CS231n *Convolutional Neural Networks for Visual Recognition*, URL: <http://cs231n.github.io/convolutional-networks/>.
 - [17] M. Nielsen, *Neural Networks and Deep Learning*, URL: <http://neuralnetworksanddeeplearning.com/>.
 - [18] Ecu Red Conocimiento con todos y para todos, *Modelo de Prototipos*, URL: http://www.ecured.cu/index.php/Modelo_de_Prototipos.

- [19] F. Fajardo, *Face Expression Detector*, Repositorio GitHub del proyecto, URL: <https://github.com/FranFT/Face-Expression-Detector>, 2017
- [20] CS231n *Convolutional Neural Networks for Visual Recognition: Transfer Learning*, URL: <http://cs231n.github.io/transfer-learning/>.
- [21] OpenCV Documentation, *Interpolation methods*, URL: http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#resize.

5.3. Herramientas

- [22] *OpenCV*, URL: <http://opencv.org/>.
- [23] *Dlib*, URL: <http://dlib.net/>.
- [24] *Caffe*, URL: <http://caffe.berkeleyvision.org/>.
- [25] *NVIDIA DIGITS*, URL: <https://developer.nvidia.com/digits>.
- [26] *Electron*, URL: <https://electron.atom.io/>.

5.4. Bases de datos

- [27] *Yalefaces Database*, URL: <http://cvc.cs.yale.edu/cvc/projects/yalefaces/yalefaces.html>.
- [28] Lundqvist, D., Flykt, A., & Öhman, A. (1998). *The Karolinska Directed Emotional Faces - KDEF*, CD ROM from Department of Clinical Neuroscience, Psychology section, Karolinska Institutet, ISBN 91-630-7164-9

Capítulo 6

Anexo

6.1. Instalación y ejecución

Para instalar la aplicación se proporciona el script **run.sh**. Este se encargará de instalar las dependencias necesarias y de compilar la aplicación para dejarla lista para su uso.

El script ha sido probado sobre una instalación fresca de *Ubuntu 16.04 LTS* consiguiendo ejecutar con éxito la aplicación. Para instalar la aplicación hay que ejecutar los siguientes comandos:

1. Actualizar el sistema operativo.

```
1 $ sudo apt-get update  
2 $ sudo apt-get upgrade
```

2. Descomprimir el archivo.
3. Situarse dentro del directorio y ejecutar el script con el argumento de instalación.

```
1 $ cd <app_directory_path>  
2 $ ./run.sh --install
```

El script **run.sh** con el argumento *--install* instalará todas las dependencias necesarias, por lo que **solo es necesario usarlo la primera vez**. Tras instalar las dependencias, se descargarán los módulos necesarios de NodeJS, entre ellos Electron. Por último se compilará Caffe en el sub-directorio *build* y se ejecutará la aplicación.

Si deseamos ejecutar la aplicación una vez instaladas dependencias y compiladas simplemente habrá que ejecutar el script sin argumentos.

```
1 $ ./run.sh
```

6.2. Uso

Tras ejecutar la aplicación se muestra la interfaz de la Figura 6.1:

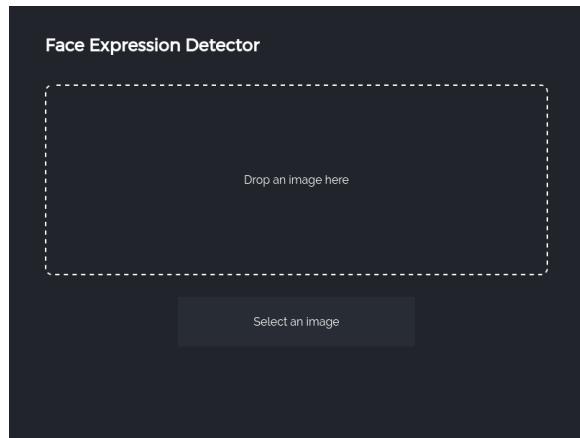


Figura 6.1: Interfaz de usuario

Podemos arrastrar un archivo al área reservada para ello con el texto *Drop an image here*, Figura 6.2, o bien podemos presionar el botón de selección del fichero, que abrirá el gestor de archivos por defecto del sistema operativo para realizar una selección al modo tradicional.

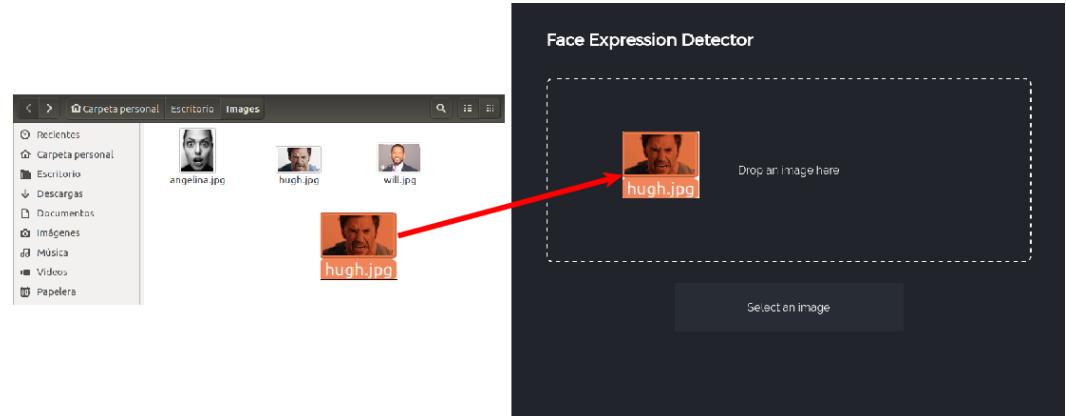


Figura 6.2: Selección de la imagen de entrada mediante *Drag & Drop*

Tras proporcionar un fichero de entrada pueden suceder tres cosas:

1. *El fichero de entrada no es una imagen* → Se notificará por una ventana emergente y se deberá proporcionar otro fichero de entrada.

2. *No se encuentra una cara en la imagen* → Se notificará por una ventana emergente y se deberá proporcionar otro fichero de entrada.
3. *Se encuentra un rostro en la imagen* → Se mostrará en la interfaz la pantalla de análisis Figura 6.3.



Figura 6.3: Pantalla de análisis

En la pantalla de análisis podemos observar los resultados y analizar otra imagen haciendo uso del botón *Analyze another image* que nos devolverá a la interfaz de la Figura 6.1.

