UNIVERSIDAD SAN FRANCISCO DE QUITO

Colegio de Postgrados

Implementación de un sistema de Identificación de poses en imágenes de rostros

Paul Esteban Méndez Silva

Julio Ibarra, M.Sc, Director de Tesis

Tesis de grado presentada como requisito
para la obtención del título de Magíster en Matemáticas Aplicadas

Quito, diciembre 2013

Universidad San Francisco de Quito

Colegio de Postgrados

HOJA DE APROBACIÓN DE TESIS

Implementación de un sistema de identificación de poses en imágenes de rostros

Paul Esteban Méndez Silva

Julio Ibarra, M.Sc.
Director de la tesis _____

Carlos Jiménez M, Ph.D.
Director de la maestría en Matemáticas Aplicadas y
Miembro del Comité de Tesis _____

Eduardo Alba, Ph.D.
Miembro del Comité de Tesis _____

César Zambrano, Ph.D.
Decano del Colegio de Ciencias _____

Victor Viteri Breedy, Ph.D.
Decano del Colegio de Posgrados _____

Quito, diciembre 2013

© DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído la Política de Propiedad Intelectual de la Universidad San Francisco de Quito y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo de investigación quedan sujetos a lo dispuesto en la Política.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo de investigación en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Firma: _____

Nombre: Paul Esteban Méndez Silva

C. I.: 171658904-7

Fecha: Quito, diciembre 2013

AGRADECIMIENTOS

A todas las personas que hicieron posible este trabajo,

gracias por su guía y enseñanzas:

a mi director de tesis Julio Ibarra, al director de la maestría Carlos Jiménez

gracias a mis profesores:

Eduardo Alba, Jhon Skukalek, Luis Gordillo, David Hervas, …

A mis amigos y compañeros de la maestría:

Diego, Andrea, Víctor, Ricardo, Bolívar, …

Y de forma especial a mis padres y hermanos por su continuo apoyo.

RESUMEN

El reconocimiento de rostros constituye un aspecto central en los campos de la Visión Artificial y la Biométrica, con aplicaciones en seguridad, robótica, interfaces humano-computadora, cámaras digitales y entretenimiento. A pesar del gran esfuerzo dedicado a mejorar los algoritmos de reconocimiento facial, su éxito aún es limitado en aplicaciones de tiempo real, sujetas a diferentes condiciones de iluminación, pose, o expresión facial.

En trabajos recientes, el reconocimiento del rostro bajo diferentes poses se ha identificado como uno de los problemas no resueltos más prominentes en el reconocimiento facial y constituye un área de creciente interés en la comunidad de investigadores en visión artificial.

En este trabajo se presenta la implementación de un sistema de reconocimiento de poses (rotación horizontal respecto a la posición de la cámara), basado en redes neuronales de convolución. El algoritmo logra una tasa de reconocimiento del 90,1% sobre la base de datos de acceso público FERET, y del 88,6% sobre la base de datos FEI.

Adicionalmente se muestra su utilización como complemento de un sistema de reconocimiento basado en rostros propios (PCA) mejorando la tasa de reconocimiento de un 40,5% a un 61.4% sobre imágenes en varias poses tomadas de la base de datos FERET.

ABSTRACT

Face recognition is a central issue in the fields of Computer Vision and Biometrics, with application in security, robotics, human-computer interfaces, digital cameras and entertainment. Despite the large effort dedicated to improving facial recognition algorithms most of them fail in produce good results in real time applications under different lighting conditions, pose, and facial expression.

In recent surveys of face recognition techniques, pose variation has been identified as one of the prominent unsolved problems in the research of face recognition and it is an area of growing interest in the computer vision community.

This work presents the development of a pose estimation system, based on convolutional neural networks. The algorithm achieves a recognition rate of 90.1% on the public access database FERET, and 88.6% on the FEI database.

Additionally it is shown its use as a complement to a recognition system based on Eigenfaces (PCA) where it allows an increase in the recognition rate from 40.5% to 61.4% over images of faces in different poses taken from the FERET database.

TABLA DE CONTENIDO

TABLAS

FIGURAS

Capítulo 1

INTRODUCCIÓN

El procesamiento automático de imágenes para extraer su contenido semántico es una tarea que ha adquirido mucha importancia en los últimos años debido en gran medida al auge de la fotografía digital y sus medios de distribución sobre todo el Internet. La necesidad de organizar esta información automáticamente y extraer datos a partir de ella requiere de técnicas eficientes de análisis y algoritmos de reconocimiento de patrones capaces de extraer la información semántica de imágenes capturadas en una variedad de ambientes. Dentro de este contexto, los rostros son especialmente valiosos dado que representan una parte importante de la información semántica contenida en una fotografía.

El reconocimiento facial constituye un área muy activa en los campos de la Visión Artificial y la Biométrica, con aplicaciones en seguridad, robótica, interfaces humano-computadora, cámaras digitales y entretenimiento.

Como una de las técnicas más importantes en la biométrica, el reconocimiento facial se caracteriza por su naturaleza no intrusiva, que permite llevar a cabo el reconocimiento tanto para sujetos cooperadores donde la información se puede obtener en ambientes controlados; como en el caso de sujetos no cooperadores donde el reconocimiento se debe llevar a cabo en situaciones arbitrarias, sin conocimiento del sujeto.

Esta última aplicación ha cobrado especial importancia en los últimos años, dado el creciente interés en la seguridad en aeropuertos y la lucha contra el terrorismo.

Por otro lado, el requerimiento de esta generalidad de ambientes y situaciones para el reconocimiento de rostros, viene acompañado de serios retos. A pesar del gran esfuerzo dedicado a mejorar los algoritmos de reconocimiento facial, todavía no se ha logrado desarrollar un sistema que pueda producir buenos resultados en tiempo real y bajo diferentes condiciones ambientales. En general el reconocimiento facial sigue siendo un área de activa investigación.

1.1. Antecedentes

El reconocimiento facial envuelve dos etapas:

- La Detección Facial, donde se explora una imagen en busca de uno o más rostros.
- El Reconocimiento Facial, donde el rostro detectado y procesado es comparado con una base de datos de rostros conocidos a fin de determinar la identidad de la persona.

1.1.1. Detección de Rostros

La detección de rostros es la etapa base para los algoritmos de análisis de rostros incluyendo el alineamiento, modelado, retocado, reconocimiento, autenticación, estimación de pose, rastreo, estimación de edad y muchos otros (C. Zhang & Zhang, 2010). El objetivo de los

algoritmos de detección de rostros es determinar si una imagen contiene o no rostros y devolver su posición y tamaño.

El área de los algoritmos de detección de rostros ha experimentado un progreso significativo en la década pasada. En particular desde el trabajo de Viola y Jones (2001) que hizo posible la detección facial en aplicaciones en tiempo real tales como las cámaras digitales y el software de organización de fotografías.

1.1.2. Reconocimiento de Rostros

El reconocimiento de rostros es una de las aplicaciones más exitosas del análisis de imágenes y la visión artificial. Esta área ha recibido un especial interés en la última década tanto por su amplia gama de aplicaciones de nivel comercial y aplicaciones de seguridad.

Muchos sistemas han emergido que son capaces de lograr tasas de reconocimiento sobre el 90 % de precisión bajo condiciones controladas. Sin embargo su efectividad disminuye si varían las condiciones de iluminación, pose, expresión entre otras.

El reconocimiento facial bajo condiciones variables es mucho menos confiable que su detección y ha sido un área de investigación activa desde 1990. A pesar de que cada año se desarrollan nuevas y más sofisticadas técnicas, estas están aún lejos de ser totalmente confiables. (X. Zhang & Gao, 2009)

1.1.3. Reconocimiento de rostros en diferentes Poses

El reconocimiento a través de diferentes poses se refiere al reconocimiento por computadora de imágenes de rostros en diferentes ángulos. Este es de gran interés en muchas aplicaciones de reconocimiento pero de forma más importante en aquellas que usan sujetos indiferentes o no cooperativos.

Dado la complejidad de las estructuras en 3D de los rostros humanos, las variaciones en el ángulo del rostro presentan serios retos a los sistemas de reconocimiento facial. Las variaciones de imágenes de rostros humanos bajo transformaciones en 3D pueden resultar mayores a las que los sistemas convencionales de reconocimiento de rostros pueden tolerar. A menudo las variaciones causadas por los cambios en el ángulo del rostro son más grandes que las magnitudes de las variaciones de las características innatas del rostro.

El reto de los algoritmos de reconocimiento de rostros es por tanto, extraer las características innatas libres de variaciones por rotación.

En recientes trabajos de investigación, la variación en el ángulo del rostro o pose se ha identificado como uno de los problemas no resueltos más prominentes en la investigación del reconocimiento facial y ha despertado un creciente interés en la comunidad de investigadores en Visión Artificial y reconocimiento de patrones (X. Zhang & Gao, 2009)

1.1.4. Redes Neuronales de convolución para la clasificación de imágenes

Las redes neuronales de convolución permiten crear sistemas capaces de extraer características locales de las imágenes de entrada, con invariancia interna con respecto a traslaciones, rotaciones y distorsiones. Esto se logra a través de la convolución de diferentes muestras de las imágenes utilizando filtros cuyas salidas son repetidamente sub-muestreadas y re filtradas en una arquitectura jerárquica de varias capas cuyos vectores de salida permiten finalmente la clasificación. (Ciresan, Meier, Masci, Gambardella, & Schmidhuber, 2011) Todos los parámetros se optimizan a través de la minimización del error de clasificación sobre un conjunto de entrenamiento

Las redes neuronales de convolución o las redes Neuronales Profundas (*Deep Neuronal Network*), se han usado en los últimos años de forma exitosa para el reconocimiento de imágenes tomadas bajo condiciones de iluminación, rotación y escala variables.

Ciresan, Giusti, Gambardella, y Schmidhuber (2012) reportaron el uso de una DNN en el reconocimiento de imágenes de microscopía electrónica del cerebro humano donde se alcanzó resultados que superan cualquier otra aproximación utilizada anteriormente para el reconocimiento de la anatomía cerebral.

Un año antes Ciresan, Meier, Masci, y Schmidhuber (2011), utilizaron una implementación en GPU de este tipo de redes alcanzando una tasa de reconocimiento del 99.15 % superior a la tasa de reconocimiento promedio del ser humano (98.98 %) sobre imágenes de señales de tránsito.

Estos trabajos junto a las redes neuronales de convolución diseñadas por LeCun para el reconocimiento de escritura a mano posicionan a las redes neuronales de convolución como una alternativa interesante para afrontar el problema central que motiva este trabajo: la estimación de poses en imágenes de rostros.

Capítulo 2

CONTEXTO Y MARCO TEÓRICO

2.1. Detección de Rostros

El objetivo de los algoritmos de detección de rostros es determinar si una imagen contiene o no rostros y devolver su posición y tamaño.

Algunos de los desafíos que deben superar los algoritmos de detección de rostros son los siguientes:

- Oclusión: Si los rostros están parcialmente obstruidos, como puede ser por el uso de gafas.
- Expresiones Faciales.
- Pose o rotación fuera del plano (rotación horizontal de la cabeza con respecto al eje de la cámara).

- Pose o rotación en el plano (rotación vertical de la cabeza con respecto al eje de la cámara).
- Iluminación.
- Otros condicionantes al momento de capturar la imagen como la resolución de la cámara, etc.

En general las principales aproximaciones que se han utilizado para resolver el problema de la detección de rostros se pueden resumir en los siguientes grupos:

- Métodos basados en el conocimiento: Tratan de definir reglas utilizando nuestro conocimiento del rostro humano.
- Métodos basados en atributos: Tratan de identificar atributos invariantes que permitan reconocer un rostro. Suelen ser muy sensibles a los cambios de iluminación y a problemas de oclusión.
- Comparación con plantillas: Se compara las imágenes con plantillas de rostros. Los principales problemas en este caso son las variaciones en escala y pose.
- Métodos basados en la apariencia: Se obtiene plantillas de rostros a partir de los propios datos. Para el aprendizaje se pude usar técnicas estadísticas o de aprendizaje automático.

El área de los algoritmos de detección de rostros ha experimentado un progreso significativo en la década pasada. En particular desde el trabajo de Viola y Jones (2001) que hizo posible la detección facial en aplicaciones en tiempo real tales como las cámaras digitales y el software de organización de fotografías. Desde entonces se han publicado diferentes mejoras al algoritmo básico en una o más componentes del mismo.

2.2. Detector de Viola-Jones

El detector de Viola-Jones está basado en varias ideas importantes que juntas hacen posible el reconocimiento en tiempo real, estas son: las imágenes integrales, el clasificador con aprendizaje *AdaBoost* y la estructura en cascada (Viola & Jones, 2004)

2.2.1. Imágenes Integrales

La imagen integral es un algoritmo para calcular eficientemente la suma de valores de un subconjunto rectangular de una grilla. Viola y Jones utilizaron las imágenes integrales para calcular atributos semejantes a los atributos de Haar. Una imagen integral se construye de acuerdo al modelo de la ecuación [1]

$$I(x,y) = \sum_{x' \le x, y' \le y} i(x', y') \tag{1}$$

Donde $I(x,y)$ es la imagen integral calculada en el pixel ubicado en (x,y) y $i(x',y')$ es la imagen original. Usando la imagen integral se puede calcular eficientemente cualquier suma de un área recangular ABCD, como:

$$\sum_{(x,y)\in ABCD} i(x,y) = I(D) + I(A) - I(B) - I(C) \tag{2}$$

Estas sumas de áreas rectangulares permiten calcular atributos semejantes a atributos Haar como los que se presentan en la siguiente figura, donde el valor del atributo es igual a la diferencia entre la sumatoria de los valores del recuadro sombreado y del recuadro en blanco.

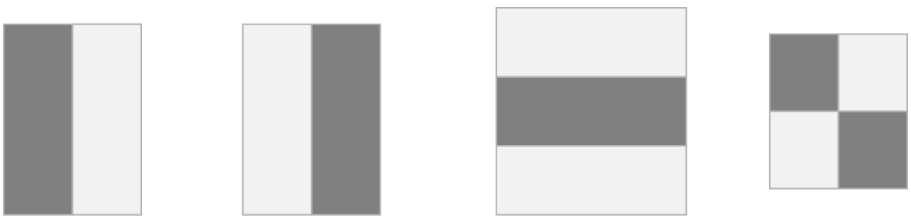

Figura 1 Ilustración de atributos rectangulares semejantes a atributos de Haar

En la práctica cada recuadro representa regiones de una imagen que contienen una grilla de valores por pixel. Estos valores generalmente corresponden a la tonalidad del gris en una imagen en escala de grises, o a valor de un vector de color en una representación como la RGB.

2.2.2. Aprendizaje AdaBoost

El "*Boosting*" es un método utilizado para encontrar con mucha precisión hipótesis combinando muchas hipótesis más "débiles", cada una con una precisión moderada.

El algoritmo *AdaBoost* (Boosting adaptativo), se considera como el primer paso hacia algoritmos de *boosting* con suficiente eficiencia para su uso en aplicaciones prácticas. El algoritmo se describe a continuación:

Dado un conjunto de ejemplos de entrenamiento $S = \{(x_i, y_i), i = 1, \dots, N\}$ donde $x_i \in \chi$, o dominio de las instancias y $y_i \in Y$ o dominio de las etiquetas. En problemas de clasificación binaria $Y = \{-1,1\}$. Si T es l número total de clasificadores débiles h_j a ser entrenados, para $t = 1, \dots, T$:

1. Normalizar los pesos:

$$w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}} \tag{3}$$

De forma que w_t, formen una distribución de probabilidad.

2. Para cada atributo j se entrena un clasificador débil h_j el cual se restringe a usar un solo atributo. El error se evalúa con respecto a w_t, $\epsilon = \sum_i w_i |h_j(x_i) - y_i|$.

3. Se escoge el clasificador h_t, con el menor error ϵ_t.

4. Se actualiza los pesos $w_{t+1,i} = w_{t,i} \beta_t^{1-\epsilon_j}$, donde $\epsilon_i = 0$, si el ejemplo se clasificó correctamente, y $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.

El clasificador final más fuerte se logra con la siguiente función:

$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & en\ otro\ caso \end{cases} \quad (4)$$

Donde $\alpha_t = \log \frac{1}{\beta_t}$.

Los clasificadores débiles se diseñan de forma que al final se seleccione un solo atributo rectangular que separe de la mejor manera los ejemplos positivos y negativos. Para cada atributo el clasificador débil determina un umbral óptimo, tal que él mínimo número de ejemplos se clasifique de forma incorrecta. Por tanto un clasificador débil $h_j(x)$ consiste de un atributo f_j, un umbral θ_j y una paridad p_j que indica la dirección de la desigualdad:

$$h_j(x) = \begin{cases} 1 & si\ p_j f_j < p_j \theta_j \\ 0 & en\ otro\ caso \end{cases} \quad (5)$$

En la práctica un solo atributo no puede llevar a cabo la clasificación con un error bajo. Los atributos seleccionados en las primeras rondas del proceso de "boosting" tienen tasas de error entre 0.1 y 0.3, conforme el proceso se vuelve más difícil en las rondas siguientes las tasas de error se elevan hasta ubicarse entre 0.4 y 0.5. (Viola & Jones, 2001).

Algoritmo 1. Algoritmo AdaBoost

1. **Inicializar** los pesos $w_i^t = \frac{1}{N}, i = 1,2, \dots, N$
2. Para $t = 1$ a T:
    a. Normalizar los pesos:

$$w^t = \frac{w^t}{\sum_{j=1}^{n} w_j^t}$$

b. Se entrena un clasificador débil h^t en la distribución w^t el cual se restringe a usar un solo atributo.

c. Evaluar el error con respecto a w^t, $\varepsilon_t = \sum_{i=1}^{N} w_i^t |h_t(x_i) - y_i|$.

d. Calcular $\beta_t = \varepsilon_t/(1 - \varepsilon_t)$

e. Se actualiza los pesos

$$w_i^{t+1} = w_i^t \beta_t^{1-h_i(x_i)\neq y_i}$$

3. Calcular $\alpha_t = \log\frac{1}{\beta_t}$

4. **Salida:** está dada por,

$$h(x) = \arg\max_{y\in Y} \sum_{i=1}^{T} (\alpha_t)||h_t(x) = y||$$

Usualmente para problemas de clasificación y regresión se utilizan ciertas variantes del algo- ritmo *AdaBoost* original, que pueden conducir a mejores resultados (Li et al., 2002), una de estas es el algoritmo *RealBoost* (ver algoritmo 2).

Algoritmo 2. Algoritmo RealBoost

Entrada: Secuencia de N ejemplos $(x_1, y_1), \ldots, (x_N, y_N)$

Una distribución D sobre los ejemplos de Clasificadores Débiles h_t

1. Inicializar los pesos:

$$w_{i,y}^t = \frac{D(i)|y - y_i|}{Z}$$

para $i = 1, \dots, N$, $y \in Y$, donde:

$$Z = \sum_{i=1}^{N} D(i) \int_0^1 |y - y_i| dy$$

2. Para $t = 1$ a T:

   a. Establecer

$$p^t = \frac{w^t}{\sum_{j=1}^{n} \int_0^2 w_{i,y}^t \, dy}$$

   b. Se entrena un clasificador débil h^t en la distribución p^t.

   c. Evaluar el error con respecto a p^t, $\varepsilon_t = \sum_{i=1}^{N} | \int_{y_1}^{h_i(x_i)} p_{y,y}^t \, dy |$.

   d. Calcular $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$

   e. Se actualiza los pesos

$$w_{i,y}^{t+1} = \begin{cases} w_{i,y}^t & si \; y_i \leq y \leq h_i(x_i) \\ w_{i,y}^t & \beta_t \end{cases}$$

3. Calcular $\alpha_t = \log \frac{1}{\beta_t}$

4. **Salida:** está dada por,

$$h(x) = \inf \left\{ y \in Y : \sum_{t: h_t(x) \leq y} \alpha_t \geq \frac{1}{2} \sum_t \alpha_t \right\}$$

2.2.3. Estructura en Cascada

La estructura en cascada es el componente crítico en el clasificador de Viola-Jones. El algoritmo construye una cascada de clasificadores que logran una tasa creciente de detección mientras reducen drásticamente el tiempo de cómputo.

La clave del algoritmo es la construcción de clasificadores usando *boosting* que son pequeños y por tanto eficientes. En cada etapa estos clasificadores se encargan de muestrear la imagen dividiéndola en diferentes sub-ventanas donde buscan la presencia de un rostro. C. Zhang y Zhang (2010)

El proceso global de clasificación de sub-ventanas se genera utilizando un árbol de decisión degenerado ("cascada"), donde cada nodo representa una etapa de la cascada, y ejecuta una decisión binaria que escoge si una determinada ventana se conservará para la siguiente ronda o será rechazado inmediatamente. El número de clasificadores débiles en cada nodo usualmente se incrementa conforme una sub-ventana va pasando más nodos.

Los nodos en la cascada se construyen entrenando clasificadores con el algoritmo *AdaBoost* y ajustando el umbral para minimizar los falsos negativos. En general un menor umbral lleva a una tasa de detección más alta con más falsos positivos.

La estructura en cascada refleja el hecho de que dentro de una imagen una gran parte de las sub-ventanas no contienen rostros y por tanto es necesario descartar tantos ejemplos negativos como sea posible en las etapas iniciales. Mientas tanto los ejemplos positivos que desencadenarán una evaluación por cada uno de los clasificadores de la cascada, resultan ser poco frecuentes.

De forma semejante a un árbol de decisión, los clasificadores siguientes se entrenan usando los ejemplos que pasaron a través de las etapas previas. Como resultado de esto la tarea de clasificar rostros es más difícil para cada nueva etapa.

El proceso de entrenamiento de la cascada envuelve equilibrar dos tipos de requerimientos. En muchos casos los clasificadores con más atributos lograrán mejores tasas de detección y menores tasas de falsos positivos. Al mismo tiempo estos clasificadores requieren más tiempo de cómputo. En principio se busca una combinación de etapas de clasificación, atributos por etapa y umbrales de los clasificadores que permitan minimizar el número de atributos evaluados, conservando una buena tasa de detección. Sin embargo, en la práctica, esta es una tarea en extremo difícil.

Por lo general se utiliza un proceso simplificado, en el cual cada etapa de la cascada reduce la tasa de falsos positivos e incrementa la tasa de detección. Cada etapa se entrena añadiendo atributos hasta que se alcanzan las tasas de detección y de falsos positivos deseadas para esa etapa en particular (estas tasas se evalúan sobre un conjunto de prueba). Posteriormente, se añaden etapas adicionales hasta que los objetivos globales para las tasas de detección y falsos positivos se alcanzan. (C. Zhang & Zhang, 2010)

Dado que la detección de rostros es un evento de poca frecuencia, usualmente se requiere de una gran cantidad de ejemplos negativos para entrenar un detector de rostros eficiente. (Viola & Jones, 2004)

2.3. Reconocimiento de Rostros

Los principales desafíos en el área del reconocimiento de rostros se pueden enmarcar en las siguientes áreas:

- Localización y escala del rostro: La imagen del rostro capturada en la etapa de detección puede tener diferentes tamaños y estar ubicada en diferentes posiciones relativas a la imagen inicial.
- Reconocimiento de características del rostro tales como posición de los ojos, boca o nariz.
- Reconocimiento de rostros bajo distintas expresiones faciales.
- Estimación de la pose o ángulo de rotación del rostro en el plano o fuera del plano.
- Iluminación: Los cambios en iluminación involucran variaciones considerables de la apariencia de los rostros. En general su influencia se puede analizar desde dos perspectivas. La primera, la iluminación global del ambiente que tiene influencia en el brillo de la imagen resultante, resulta más fácil de manejar. La segunda, la dirección de la luz presenta más complicaciones ya que generalmente sigue una función altamente no lineal y puede causar que el rostro presente sombras solo en ciertas partes de la imagen.

El punto inicial para numerosos estudios y líneas de investigación en el área del reconocimiento de rostros ha sido (y sigue siendo), los algoritmos basados en el análisis de componentes principales (PCA). Los algoritmos basados en PCA son populares debido a la facilidad de su implementación y sus razonablemente altos niveles de eficiencia. Por esta razón se emplean a menudo como un punto de referencia para comparar la eficiencia de nuevos algoritmos (Moon & Phillips, 2001)

2.3.1. Reconocimiento de rostros basado en el análisis de componentes principales (PCA)

PCA es un método estadístico utilizado para reducir la dimensionalidad de un conjunto de datos manteniendo la mayor parte de su variabilidad.

El algoritmo más conocido de reconocimiento de rostros basado en componentes principales se conoce generalmente como el algoritmo de rostros propios o *Eigenfaces*.

En el reconocimiento facial usando rostros propios, como en muchas otras técnicas de reconocimiento, se debe pre procesar un conjunto de entrenamiento usando un proceso computacionalmente intensivo para crear un conjunto de rostros propios mutuamente ortogonales que definen un espacio de rostros propios. (Turk & Pentland, 1991)

Espacio Propio de Rostros El espacio propio se calcula identificando los vectores propios de la matriz de covarianza derivada del conjunto de imágenes de entrenamiento. Los vectores propios correspondientes a los valores propios diferentes de cero de la matriz de covarianza, forman una base ortonormal que rota y/o refleja las imágenes en un espacio de N dimensiones. Específicamente cada imagen se guarda en un vector de tamaño N

$$x^i = \left[x_1^i, \dots, x_N^i\right]^T \tag{6}$$

Las imágenes se centran en la media substrayendo la media de cada vector de imagen.

$$\bar{x}^i = x^i - m \tag{7}$$

Donde $m = \frac{1}{p} P \sum_{i=1}^{p} x_i$ Estos vectores se combinan como columnas para formar una matriz X de tamaño $N \times P$, donde P es el número de imágenes.

La matriz X se multiplica por su transpuesta para calcular la matriz de covarianza:

$$\Omega = \overline{\bar{X}\bar{X}^t} \tag{8}$$

La matriz de covarianza tiene un máximo de P valores propios asociados con valores diferentes de cero, asumiendo $P \leq N$. Los vectores propios se ordenan de acuerdo a su valor propio asociado, desde el más alto al más bajo. El vector propio asociado al valor propio más alto, contiene la mayor varianza de las imágenes, el segundo vector propio, es el de segunda mayor varianza y así sucesivamente.

Reconocimiento de Rostros sobre el espacio propio El reconocimiento de imágenes a través del espacio propio de proyección toma tres pasos básicos:

Primero se crea el espacio propio, usando las imágenes de entrenamiento para esto las imágenes son centradas, y se obtiene la matriz de vectores propios de la varianza dada por la ecuación [8]. Por facilidad usualmente este proceso se efectúa sobre imágenes en escala de grises, representadas como una matriz de $m \times n$ píxeles, que luego se convierten en los vectores columna de la matriz X.

A continuación cada imagen centrada $\bar{x}^i$ correspondiente a un individuo conocido es proyec- tada en el espacio propio y se obtiene sus coordenadas que son registradas para ser utilizadas en la comparación con las coordenadas de las imágenes de prueba. Para proyectar una imagen se calcula el producto punto de la imagen con cada uno de los vectores propios ordenados:

$$\bar{x}^i = V^t \bar{x}^i \tag{9}$$

De donde el producto punto de la imagen y el primer vector propio será el primer valor en el nuevo vector. El nuevo vector contendrá por tanto, un número de entradas igual al de vectores propios a utilizarse. Un rostro propio es el resultado de esta proyección de la imagen del rostro sobre el espacio de rostros propios de dimensiones reducidas. Esta proyección hace al reconocimiento de rostros más rápido y robusto, reduciendo al mismo tiempo de forma considerable la información que se debe almacenar para cada rostro. Finalmente para la identificación de las imágenes, cada imagen de prueba es centrada y proyectada en el espacio propio definido por V. La imagen de prueba proyectada $\overline{y_l}$ se compara entonces con cada una de las imágenes del conjunto de entrenamiento $\overline{x_l}$, usando usualmente una métrica como la distancia Euclidiana. La imagen más cercana es entonces utilizada para identificar la imagen de prueba. En sistemas donde puede necesitar evaluarse rostros de individuos desconocidos, se utiliza un umbral T. Si cada par de imágenes comparadas produce una distancia mayor al umbral, entonces el candidato se clasifica como desconocido.

2.3.2. Otros algoritmos de reconocimiento de rostros

En esta sección se hace un breve recuento de algunos de los algoritmos más conocidos, desarrollados para el reconocimiento de rostros.

Reconocimiento usando atributos propios. Este método se puede ver como una representación modular o en capas del rostro, donde una descripción general del rostro es aumentada con la representación de sus atributos o características. Es una extensión del método de rostros propios, en la cual para cada uno de estos atributos se construye un espacio de atributos. Por lo general comprende las siguientes etapas:

- *Extracción de atributos*: Después de que los atributos faciales se extraen de la imagen de prueba, se calcula una medida de similitud entre estos atributos y los atributos correspondientes en el modelo almacenado.
- *Clasificación*: Un enfoque simple es calcular una medida acumulada de similitud en términos de las contribuciones de las medidas de cada uno de los atributos evaluados. Una vez determinada esta medida acumulada, el rostro se clasifica según el modelo que maximice esta medida. Este método puede presentar una tasa de reconocimiento del 98 % sobre imágenes de rostros frontales. (Nefian, 1996)

Análisis de Discriminante Lineal. También conocido como algoritmo de Discriminante Lineal de Fisher (FLD), este trabaja usando clases o métodos específicos por individuo para reducir la dimensionalidad de cada imagen en el espacio de rostros, a diferencia de los rostros propios donde se aplica la misma transformación a cada imagen de entrada.

Este método reporta tasas de error de entre 7.4 y 0.6 % sobre 160 imágenes frontales de la base de datos de Yale, bajo variaciones en iluminación y expresiones faciales. (Belhumeur, Hespanha, & Kriegman, 1997)

Análisis de Componentes Independientes. Mientras el PCA elimina la correlación de los datos de entrada usando estadísticos de segundo orden, el análisis de componentes independientes (ICA), minimiza conjuntamente dependencias de segundo orden y de órdenes mayores. Como resultado el ICA provee una representación de los datos más robusto que la obtenida con el PCA. Draper, Baek, Bartlett, y Beveridge (2003)

Sobre un conjunto de entrenamiento de 425 imágenes de la base de datos FERET, y un conjunto de prueba de 421 imágenes de la misma base de datos, este método permitió alcanzar tasas de reconocimiento cercanas al 90 %, para imágenes de rostros frontales con diferentes expresiones faciales. (Bartlett, 2002)

Descomposición de valor singular. Este método utiliza el teorema de la descomposición de valor singular de una matriz, para luego utilizar los vectores resultantes de esta descomposición en un proceso en dos etapas:

- Se extrae un conjunto de vectores discriminantes óptimos (usando por ejemplo el método de análisis de discriminante lineal).
- Cuando se analiza una imagen de prueba se proyecta los vectores resultantes de su descomposición en el espacio definido por los vectores discriminantes y se efectúa la clasificación, utilizando la distancia euclidiana en este espacio.

Máquinas de Vectores de Soporte. Las máquinas de vectores de soporte o SVM por sus siglas en inglés, son un método conceptualmente simple de reconocimiento por clases segmentadas en espacios de muchas dimensiones. A diferencia de los métodos basados en PCA, que reducen la dimensionalidad de cada punto bajo consideración antes de efectuar la clasificación, las SVM tratan cada píxel de una imagen como un punto en un espacio. A

continuación las SVM comparan cada punto candidato a un conjunto sucesivo de pares conocidos como clases para determinar su clase experimental.

Dado que el SVM puede solo distinguir entre dos clases al mismo tiempo, es necesario eliminar iterativamente las categorías posibles para un punto de prueba determinado. Si se tiene k categorías, esto define $k(k\ -\ 1)/2$ comparaciones posibles. (Guo, Li, & Chan, 2000)

Sobre 295 imágenes de la base de datos M2VTS compuesta por rostros frontales con diferentes grados de iluminación, se ha reportado para este método una tasa mínima de clasificación errónea del 2 %. (Jonsson, Matas, Kittler, & Li, 2000)

Modelo oculto de Márkov. Los modelos ocultos de Márkov se han utilizado de forma exitosa para el análisis y el reconocimiento de patrones de una sola dimensión tales como el procesamiento de señales o el reconocimiento de voz. (Nefian & Hayes, 1998)

El reconocimiento facial se ha desarrollado usualmente como un proceso de dos dimensiones o una segmentación unidimensional de características del rostro. En este último método cada imagen se segmenta en características importantes tales como pelo, frente, ojos, nariz y boca; y estas características se alimentan a un modelo oculto de Márkov donde se calcula las verosimilitudes $P[O_{prueba}|\lambda_i]$. El modelo con la más alta verosimilitud revela la identidad del rostro.

Se ha reportado con este sistema una tasa de reconocimiento de 84 % sobre imágenes de rostros frontales de la base de datos AT&T, y un conjunto de prueba de 200 imágenes (Nefian, 1999).

Redes Neuronales de Convolución. Las redes neuronales de convolución se explicarán con más detalle en secciones posteriores bajo el contexto del reconocimiento de poses. Sin embargo estas también se han utilizado directamente para el reconocimiento facial. En este

contexto, el método ha reportado muy buenos resultados sobre un conjunto de prueba de 200 imágenes frontales de la base de datos ORL para 40 individuos diferentes, con una tasa de reconocimiento del 97,5 % (Lawrence, Giles, Tsoi, & Back, 1997).

2.4. Algoritmos de estimación de poses

Comparado a la detección y el reconocimiento facial la estimación de la pose de un rostro presenta pocos sistemas evaluados de forma rigurosa. Según Murphy-Chutorian y Trivedi (2009), los sistemas de estimación de poses presentes en la actualidad pueden enmarcarse en las siguientes categorías:

- Métodos basados en plantillas de apariencia: Comparan una nueva imagen con un conjunto de ejemplos con el fin de encontrar la vista más general.
- Métodos basados en arreglos de detectores: entrenan una serie de detectores, cada uno especializado en una pose en particular.
- Métodos de regresión no lineales.
- Modelos Flexibles: ajustan un modelo flexible de la cabeza de un individuo a un plano en dos dimensiones.
- Métodos Geométricos: Usan la localización de características tales como los ojos, la nariz o la boca para determinar la posición de la cabeza.
- Métodos de rastreo: Recuperan la variación global de la posición de la cabeza analizando diferentes cuadros de una secuencia de video.
- Métodos híbridos: combinan dos o más de las aproximaciones anteriores.

Dentro de los métodos de estimación de poses por regresiones no lineales, una aproximación importante son las redes neuronales. Las ventajas de las redes neuronales son

numerosas. Estos sistemas son muy rápidos y solo requieren imágenes recortadas y etiquetadas para su entrenamiento. Resultan mucho más robustas que los métodos basados en características ya que utilizan para su entrenamiento toda la imagen, eliminando los errores introducidos al estimar la posición de características del rostro (Stiefelhagen, 2004). Adicionalmente algunas de estas han reportado estimaciones de pose bastante precisas en la práctica. (Murphy-Chutorian & Trivedi, 2009)

2.5. Reconocimiento de rostros en diferentes poses

Existen diferentes acercamientos al problema del reconocimiento de rostros bajo variaciones de pose. Uno de estos es el modelo de apariencia activa propuesto en el trabajo de Cootes y Taylor (2004), el cual deforma un modelo de rostro genérico para ajustar la imagen de entrada y luego usa los parámetros de control como vectores de atributos para alimentar un clasificador. Otro posible acercamiento es alimentar un espacio propio con diferentes vistas y efectuar el reconocimiento sobre este espacio. (Reddy, 2012)

Una tercera aproximación involucra el uso de modelos 3D como plantillas para realizar la comparación de las entradas, propuesta por Blanz y Vetter (2003), Bronstein, Bronstein, Gordon, y Kimmel (2004) y, Weyrauch, Heisele, Huang, y Blanz (2004).

Otros enfoques pueden verse como modificaciones de estas aproximaciones o enmarcarse en los mismos principios fundamentales. En consecuencia los enfoques de la detección de rostros bajo un cambio de pose, se puede clasificar en los siguientes tres grupos:

2.5.1. Algoritmos generales

Estos métodos intentan extraer patrones de clasificación o características de imágenes de rostros en 2D y reconocer las imágenes de rostros de entrada basados en estos patrones contra las imágenes en la base de datos. Los ejemplos más conocidos son el análisis de componentes

principales (PCA) (Reddy, 2012), el análisis de discriminadores de Fisher (LDA) (Chen, Man, & Nefian, 2005), PCA modular, Redes neuronales (Lawrence, Giles, Tsoi, & Back, 1998), entre otros.

Dentro de estos algoritmos se pueden distinguir dos tipos de aproximaciones, las aproximaciones holísticas que buscan extraer patrones de la imagen en su conjunto y aproximaciones locales, que buscan aislar regiones del rostro y extraer características de estas regiones. A pesar de que estos algoritmos locales pueden aliviar en algo el problema de la variación de poses, esto solo se logra de forma limitada debido a que también existen distorsiones de la imagen a nivel local.

Experimentos con algoritmos holísticos han mostrado que estos pueden tener una tolerancia a rotaciones de hasta 20◦, con una precisión del 63 % en el caso de PCA al 70 % para el algoritmo LEM (mapas de bordes lineales) (X. Zhang & Gao, 2009).

Las ventajas de estos algoritmos son su relativamente simple implementación y su velocidad que facilita su uso en aplicaciones de reconocimiento en tiempo real.

2.5.2. Manejo de variaciones de pose usando técnicas 2D

Las técnicas de manejo en 2D de la variación en pose del rostro, tratan de utilizar múltiples vistas de un mismo rostro tanto a través del uso múltiples imágenes reales de cada individuo como a través de la utilización de algoritmos que transforman imágenes conocidas para generar imágenes virtuales de poses desconocidas que ayuden en la tarea del reconocimiento. El rendimiento de estos algoritmos es bastante similar al de los algoritmos de reconocimiento frontal, debido a que su única diferencia es trabajar en subespacios de diferentes poses. Sus limitaciones radican en que necesitan un número relativamente grande de imágenes por individuo.

Estas técnicas se pueden subdividir a su vez en los siguientes grupos:

Emparejamiento baso en vistas reales. Estos algoritmos se basan en preparar una galería de posiciones para cada individuo que se desea identificar. Algunos algoritmos representativos de este grupo son el esquema en mosaico (Singh, Vatsa, Ross, & Noore, 2007) y el emparejamiento de plantillas (Beymer, 1994).

Transformación de pose en el espacio de imagen. Una alternativa a la recolección de imágenes en varias poses de un mismo individuo es la generación de poses virtuales. Esto se puede realizar utilizando algoritmos como el de deformación paralela y los modelos de apariencia activa (Wang, Shan, Gao, Cao, & Yin, 2002).

Transformación de pose en el espacio de características. La tolerancia a la variación en pose se puede obtener también en el espacio de las características. Un espacio de características posible se obtiene a través de núcleos o *kernels*, como en los algoritmos de *kernels* con PCA, o el análisis de discriminador de Fisher en *Kernel* (Bach & Jordan, 2003). Estos algoritmos buscan mapear imágenes de rostros en espacios de características no lineales y buscar la invariancia a la pose del rostro sobre este espacio (Yang, 2001).

Las técnicas usadas en la síntesis de vistas virtuales de posibles poses a partir de un número limitado de vistas asumen continuidad de la imagen en las transformaciones de pose y pueden manejar efectivamente variaciones usualmente limitadas a 45◦ (Georghiades, Belhumeur, & Kriegman, 2001)

Variaciones mayores implican discontinuidades en el espacio de imágenes 2D que no pueden manejarse de forma confiable. Bajo esas circunstancias las aproximaciones en 3D suelen generar mejores resultados. (X. Zhang & Gao, 2009)

2.5.3. Manejo de variaciones en Pose usando modelos en 3D

Estos métodos se pueden clasificar en 3 sub categorías:

- Métodos basados en formas genéricas.
- Reconstrucciones 3D basadas en características.
- Reconstrucciones 3D basadas en imágenes.

Dentro de estos algoritmos, la estrategia más simple es utilizar una forma 3D genérica de la cabeza para modelar las variaciones del rostro en diferentes poses. A pesar de que estos métodos presentan cierta eficiencia, también presentan problemas debido a las desviaciones que existen en la forma de los rostros entre diferentes individuos. (Blanz & Vetter, 1999)

Una mejor aproximación se puede lograr con modelos personalizados para cada individuo que pueden construirse a partir de un conjunto de características o de un conjunto de imágenes. La reconstrucción basada en características requiere la localización puntual de ciertas características del rostro o utilizan información de bordes. (Ju, 2010)

La tercera aproximación consigue una reconstrucción más detallada del rostro en tres dimensiones usando imágenes completas pero en general requiere de procedimientos más complejos y potencialmente más intensivos desde el punto de vista computacional (Moghaddam, Pfister, & Machiraju, 2004)

2.6. Redes Neuronales

El término Red Neuronal tiene sus orígenes en los intentos de encontrar modelos matemáticos para representar el procesamiento de información en sistemas biológicos. El modelo más exi- toso en el contexto de reconocimiento son las redes neuronales de retro-alimentación, también conocidas como perceptrones de múltiples capas.

2.6.1. Perceptrón de capas múltiples

El perceptrón multicapa es una red neuronal artificial capaz de aproximar funciones no lineales arbitrariamente complejas. Estas ganaron particular interés a partir de 1986 con la publicación del algoritmo de propagación hacia atrás (Rumelhart, Hinton, & Williams, 1986).

La estructura general de un MLP consiste en una capa de unidades de entrada, una o más capas ocultas y una capa de unidades de salida, donde cada unidad con excepción de las entradas se implementan como perceptrones.

Normalmente las unidades de una capa solo se conectan directamente con las unidades de la siguiente capa.

Esta estructura se le conoce también como Red de retroalimentación, dado que la activación de las neuronas se propaga de una capa a la siguiente hasta alcanzar la capa de salida.

El algoritmo de propagación hacia atrás requiere que la función de activación de los perceptrones sea derivable, las elecciones comunes suelen ser funciones lineales, sigmoidea o la tangente hiperbólica.

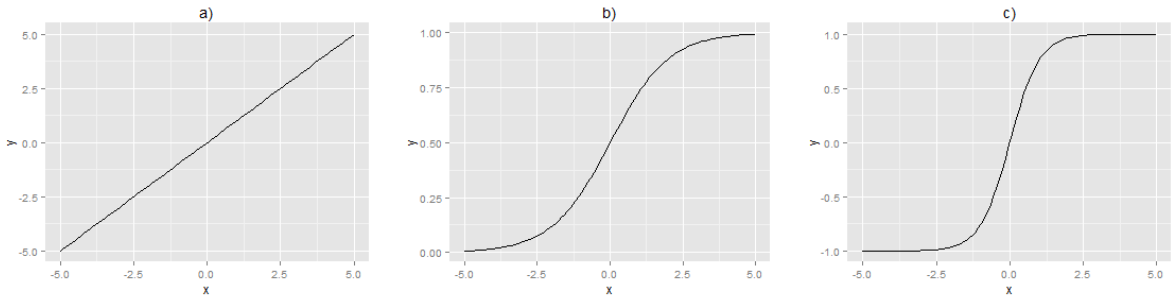

Figura 2. Funciones de activación comunes: a) lineal, b) sigmoidea, c) tangente hiperbólica

2.6.2. Propagación hacia atrás de errores

La propagación hacia atrás de errores es el algoritmo más conocido para el entrenamiento de redes neuronales. Este es conceptualmente simple y computacionalmente eficiente (LeCun, Bottou, Orr, & Muller, 1998).

La propagación hacia atrás de errores permite determinar el error en la salida de una capa anterior dada la salida de la capa actual. Es un algoritmo de aprendizaje supervisado que define una función de error E y utiliza una técnica de descenso de gradiente en el espacio de los pesos para minimizar E.

Como una deficiencia inherente al uso del descenso de gradiente, el algoritmo BP no garantiza encontrar un mínimo global.

En el caso más simple el algoritmo utiliza una tasa de aprendizaje o η constante. En métodos más sofisticados η toma la forma de una matriz diagonal, o el estimado del inverso de la matriz Hessiana de la función de costo (LeCun et al., 1998).

2.6.3. Estimación del valor de la tasa de aprendizaje η

Asumiendo que la función de costo es aproximadamente cuadrática, el valor óptimo de η se puede obtener de la expansión de Taylor de la función de costo, alrededor del peso actual:

$$E(W) = E(W_C) + \frac{(W - W_C)dE(W_C)}{dW} + \frac{1}{2}(W - W_C)^2 \frac{d^2E(W_C)}{dW^2} + \cdots \qquad (10)$$

Si $E(W)$ es cuadrática, el segundo término es una constante y el resto de términos desaparecen. Si se diferencia la ecuación anterior con respecto al peso actual y se despeja el peso mínimo se obtiene:

$$W_{min} = W_c - \frac{\left(\frac{d^2E(W_C)}{dW^2}\right)^{-1} dE(W_c)}{dW} \tag{11}$$

Por tanto el valor de η que permite obtener el valor óptimo del peso en un solo paso estaría dado por:

$$\eta_{opt} = \left(\frac{d^2E(W_C)}{dW^2}\right)^{-1} \tag{12}$$

En consecuencia cuando se trabaja con múltiples dimensiones η_{opt} es el inverso de la matriz Hessiana.

2.6.4. Extensiones del algoritmo de propagación hacia atrás

2.6.4.1. Momento

Es este caso se añade un término de "momento" con el fin de tomar en cuenta los cambios previos realizados sobre el peso:

$$w_{ji_t} = w_{ji_{t-1}} - \frac{\eta \partial E_n}{\partial w_{ji}} + \alpha \Delta w_{ji_{t-1}} \tag{13}$$

Donde $\alpha \in [0, 1]$ es la tasa del momento. El objetivo de este término es mejorar la velocidad de convergencia del algoritmo. Si existen actualizaciones consecutivas de un peso en la misma dirección, los cambios serán mayores. Si el peso oscila los cambios se reducirán.

2.6.4.2. Decaimiento de Peso

Este método añade un término $\frac{\alpha}{2}\sum_x w_x^2$ a la función de error,

$$E_n = \frac{1}{2}\sum_X (y_{nk} - t_{nk})^2 + \frac{\alpha}{2}\sum_X w_x^2 \tag{14}$$

Donde w_x son todos los pesos de la red neuronal.

El objetivo de este término es penalizar los pesos altos para reducir el sobre ajuste a los datos en la red neuronal (Werbos, 1988).

2.6.4.3. Método estocástico de Levenberg Marquadt

Este es un algoritmo que permite optimizar el valor de η utilizando la matriz Hessiana diagonal obtenida por propagación hacia atrás, logrando una convergencia más rápida en comparación al método de propagación hacia atrás estándar.

En su artículo acerca de propagación hacia atrás eficiente, LeCun afirma que este método permite alcanzar la convergencia alrededor de tres veces más rápido que el algoritmo estándar. Bajo este algoritmo se puede ajustar el parámetro de la tasa de aprendizaje utilizando la siguiente ecuación:

$$\eta_{ki} = \frac{\eta}{\langle \frac{\partial^2 E}{\partial w_{ki}} + \mu \rangle} \tag{15}$$

Donde μ es un parámetro que previene que el valor de η_{ki} explote.

En la práctica las segundas derivadas se pueden calcular con un subconjunto del conjunto de entrenamiento, a fin de minimizar su impacto sobre la velocidad del algoritmo.

2.6.5. Propagación hacia atrás de la matriz Hessiana

Asumiendo una función de costo cuadrática se tiene:

$$E(w) = \frac{1}{2}\sum\nolimits_k \big(t_k - f(w, a_k)\big)^t (t_k - f(w, a_k)) \tag{16}$$

Donde el gradiente está dado por:

$$\frac{\partial E(w)}{\partial w} = -\sum_k \left(t_k - f(w, a_k)\right)^t \frac{\partial f(w, a_k)}{\partial w} \tag{17}$$

Y el Hessiano es igual a:

$$H(w)$$

$$= \sum_k \left[\frac{\partial f(w, a_k)}{\partial w}\right]^t \frac{\partial f(w, a_k)}{\partial w} + \sum_k \left(t_k - f(w, a_k)\right)^t \frac{\partial^2 f(w, a_k)}{\partial w \partial w} \tag{18}$$

Al eliminar el segundo término de la ecuación anterior se obtiene una aproximación del Hessiano usando el cuadrado del Jacobiano:

$$H(w) = \sum_k \left[\frac{\partial f(w, a_k)}{\partial w}\right]^t \frac{\partial f(w, a_k)}{\partial w} \tag{19}$$

Esta aproximación es equivalente a asumir que la red es una función lineal de los parámetros w. Para la ecuación del costo en una red neuronal si se descarta el término que contiene h'' se obtiene la aproximación de Gauss Newton del Hessiano:

$$\frac{\partial^2 E}{\partial a_k^2} = \frac{\partial^2 E}{\partial z_k^2}\left(h'(a_k)\right)^2 \tag{20}$$

De la misma forma si se deriva con respecto a los pesos se obtiene la siguiente fórmula de propagación hacia atrás:

$$\frac{\partial^2 E}{\partial w_{ki}^2} = \frac{\partial^2 E}{\partial a_k^2} z_i^2 \tag{21}$$

$$\frac{\partial^2 E}{\partial z_{ki}^2} = \sum_k \frac{\partial^2 E}{\partial a_k^2} w_{ki}^2 \tag{22}$$

Donde para la capa final de la red neuronal $\frac{\partial^2 E}{\partial a_k^2} = 1$ para todo k.

2.6.6. Redes Neuronales de Convolución

Clásicamente las redes muticapa de perceptrones se han utilizado para clasificar patrones de los cuales se extraen previamente atributos con algún algoritmo adicional. Sin embargo, la extracción manual de estos atributos es a menudo empírica y por tanto sub-óptima. Una solución posible a este problema es alimentar directamente los datos a la red neuronal y permitir que el algoritmo de entrenamiento, encuentre los mejores atributos ajustando los pesos correspondientes.

El problema de esta última aproximación es que las dimensiones de los datos de entrada resultan ser muy altas, y por tanto la red final cuenta con un número muy grande de conexiones y de parámetros libres. Esto aumenta considerablemente la cantidad de muestras que se necesitan en el conjunto de entrenamiento, constituyéndose en una gran limitante (Lecun, Bottou, Bengio, & Haffner, 1998).

Otra desventaja de las redes neuronales convencionales es que sus capas de entrada tienen tamaño constante y por tanto se requiere que los patrones de entrada estén igualmente alineados y normalizados a esta ventana de entrada. En la práctica eso resulta ser una tarea complicada; por tanto es deseable construir la red de forma que presente cierta invariancia a pequeñas traslaciones y distorsiones locales.

Finalmente, las arquitecturas convencionales de redes no toman en cuenta la correlación existente entre datos de entrada cercanos. Esta correlación por otro parte suele ser común en

algunos problemas de reconocimiento, y por tanto sería preferible extraer atributos locales y combinarlos antes de realizar el reconocimiento.

Las redes neuronales de convolución se diseñaron con los problemas anteriores en mente, estas buscan extraer atributos locales, de forma que puedan dar resultados robustos frente a traslaciones y distorsiones de los patrones de entrada. Esto se logra en gran medida gracias a tres ideas importantes: mapas receptivos locales, pesos compartidos (o replicación de pesos), y en ocasiones sub muestreo espacial.

El principio de pesos compartidos reduce drásticamente el número de parámetros libres e incrementa su capacidad de generalización comparado a las arquitecturas de redes neuronales que no implementan esta propiedad (Bouvrie, 2006).

Los mapas receptivos locales pueden extraer características visuales elementales tales como bordes, puntos terminales, o esquinas. Adicionalmente, estos detectores que resultan útiles en una parte de la imagen, probablemente pueden ser útiles a lo largo de toda la imagen. El recorrido de los detectores por la imagen, se logra mediante mapas de neuronas formados con pesos compartidos. En cada posición, diferentes unidades en diferentes mapas calculan diferentes tipos de características. Una implementación secuencial de estas unidades, barrerá la imagen de entrada con un mapa de neuronas que tienen un receptor local y almacenan sus estados como pesos compartidos en el mapa de características. Esta operación es equivalente a la convolución con un *kernel*.

En redes neuronales de convolución típicas, se alternan operaciones de convolución y sub muestreo, con una etapa final de capas genéricas y totalmente conectadas.

Las capas de convolución están alternadas con capas de submuestreo para reducir el tiempo de cálculos y para construir gradualmente invariancia a la posición y la configuración espacial.

Simultáneamente se desea que este factor de submuestreo sea pequeño para conservar especificidad.

Las capas de submustreo están constituidas por mapas de atributos, y el modelo individual de las neuronas corresponde al perceptrón básico, con una función de activación sigmoidea

Un hecho importante en el desarrollo de las redes neuronales fue el uso del algoritmo de propagación hacia atrás, para entrenar las redes. La red presentada por LeCun, Jackel, y Bottou (1995) fue la primera red neuronal de convolución entrenada por propagación hacia atrás y aplicada al problema del reconocimiento de escritura a mano.

La siguiente figura presenta la topología de la CNN propuesta por LeCun, a la que se refiere en trabajos posteriores como LeNet-1. En esta red, la capa de entrada tiene un tamaño de 28×28 y recibe una imagen en escala de grises conteniendo el dígito a reconocer.

Las intensidades de los píxeles se normalizan entre -1 y +1. La primera capa oculta $H1$ consiste de cuatro mapas de atributos $y_j^{(1)}$ cada uno conteniendo 25 pesos w_{j0}, que constituyen un kernel de 5×5 y un sesgo b_j. Por tanto el valor de los mapas de atributos se obtienen al convolucionar el mapa de entrada con el *kernel* y aplicando la función de activación h al resultado:

$$y_k = h\left(\sum_{u,v \in K} w_{j0}(u,v) \cdot y(x+u, y+v) + b_j\right) \tag{23}$$

Donde $K = (u,v) \in N | 0 \leq u < 5$ y $0 \leq v < 5$. Por tanto los mapas de atributos resultantes tienen dimensiones 24×24.

A continuación cada mapa de convolución es seguido por un mapa de submuestreo $y_j^{(2)}$, el cual ejecuta algún tipo de reducción de dimensiones del mapa de convolución. Las capas 3 y 4 , cada una contienen 12 mapas de convolución y mapas de submuestreo de dimensiones 8×8 y

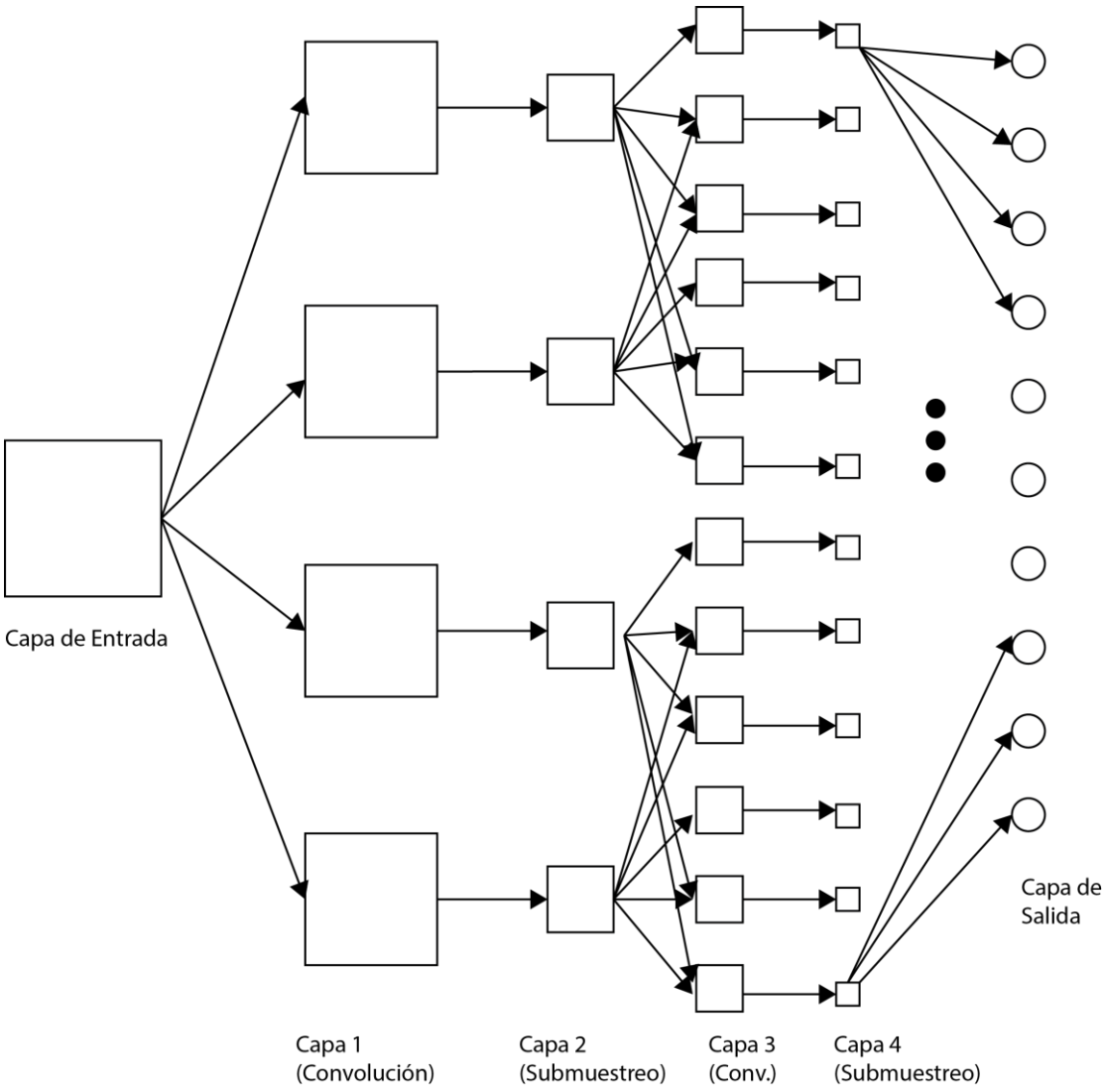

Figura 3. Arquitectura propuesta por LeCun et al para reconocimiento de dígitos (LeNet1)

4×4 respectivamente. La función usada es exactamente la misma que la de las capas precedentes, con la excepción de que los mapas de atributos son de dimensiones 3×3.

Finalmente la capa de salida contiene un conjunto de 10 neuronas, conectadas completamente a los mapas de submuestreo de la capa 4, y representa los 10 dígitos que se desea reconocer (LeCun et al., 1995).

El dígito reconocido es aquel que corresponde a la neurona con la respuesta más alta.

En total esta red tiene 4635 unidades y 98442 conexiones, pero debido a la compartición de pesos, solo presenta 2578 parámetros independientes que entrenar.

2.6.6.1. Capas de Convolución

Una capa de convolución está parametrizada por el número de mapas, el tamaño del *kernel* y los factores de salto. Cada capa tiene M mapas de igual tamaño (M_x, M_Y). Un kernel de tamaño (K_X, K_Y) se desplaza sobre la imagen de entrada. Los factores de salto S_X, S_Y, definen cuantos pixeles saltará el *kernel* en cada desplazamiento, entre convoluciones. Por tanto el tamaño del mapa de salida queda definido por las siguientes ecuaciones:

$$M_x^n = \frac{M_x^{n-1} - K_x^n}{S_x^n} + 1; M_y^n = \frac{M_y^{n-1} - K_y^n}{S_y^n} + 1; \tag{24}$$

Donde el índice n indica el número de capa.

Cada mapa en la capa L^n se conecta como máximo con M^{n-1} mapas en la capa L^{n-1}. Los pesos de las neuronas en cada mapa son compartidos. (Ciresan, Meier, Masci, Gambardella, & Schmidhuber, 2011) Las activaciones resultantes de los M^n mapas de salida son dadas por la suma de las M^{n-1} respuestas convolucionales obtenidas al pasar a través de una función de activación no lineal:

$$Y_j^n = f\left(\sum_{i=1}^{M^{n-1}} Y_i^{n-1} * W_{ij}^n + b_j^n\right) \tag{25}$$

Donde n indica el número de capa, Y es el mapa de tamaño $Mx \times My$ y W_{ij} es un filtro de tamaño $Kx \times Ky$ conectando el mapa de entrada i con el mapa de salida j, b_n^j es el sesgo del mapa de salida j y * es la operación de convolución en dos dimensiones.

2.6.6.2. Entrenamiento de redes neuronales de convolución

El entrenamiento de una red neuronal de convolución es muy similar al de otros tipos de redes neuronales. Tanto el procesamiento hacia adelante como el procesamiento hacia atrás requieren solo de ligeros ajustes en el algoritmo original.

Normalmente las ecuaciones de procesamiento hacia atrás ya presentadas se deben ajustar para las capas de convolución sumando el aporte de las derivadas en cada mapa de atributos para cada peso compartido.

Para el entrenamiento, generalmente se utiliza un conjunto grande de imágenes, y se evalúa la tasa de error obtenida en cada iteración con otro conjunto de imágenes de prueba para garantizar que no se produzca un sobre ajuste.

En ciertos casos se puede añadir cierto ruido a las imágenes de entrenamiento para aumentar la robustez de la red neuronal frente a pequeños desplazamientos o cambios de escala (Simard, Steinkraus, & Platt, 2003).

Capítulo 3

METODOLOGÍA Y DISEÑO

3.1. Recursos y herramientas

3.1.1. OpenCV

OpenCV (*Open source computer vision*) es una librería de visión artificial de código abierto, originalmente diseñada por Intel en 1999. La librería está escrita en C y C++ y tiene versiones para sistemas Linux, Windows, Mac OS X y algunas plataformas móviles. Existen también interfaces para su uso con Python, Ruby, Matlab entre otros lenguajes.

OpenCV está diseñado con el propósito de ser altamente eficiente con un fuerte enfoque en aplicaciones en tiempo real. Uno de los objetivos de *OpenCV* es proveer de una infraestructura para investigaciones en el campo de la visión artificial que sea fácil de usar y que permita desarrollar aplicaciones de visión artificial de forma rápida y eficiente. La librería *OpenCV* contiene alrededor de 500 funciones que abarcan diversas áreas de la visión artificial

incluyendo el manejo eficiente de imágenes y matrices, reconocimiento de objetos, análisis de imágenes médicas, seguridad, interfaz de usuario, calibración de cámara, visión estéreo y visión robótica. (Bradski & Kaehler, 2008)

OpenCV también contiene una extensa librería de Aprendizaje Automático de propósito general. Esta sub librería está enfocada en el reconocimiento estadístico de patrones y clústeres.

3.1.2. Bases de datos de imágenes

3.1.2.1. FERET

El programa (FERET) es administrado por la Agencia (DARPA) (Defense Advanced Research Projects Agency) y (NIST) (National Institute of Standards and Technology). La base de datos consiste en imágenes faciales recogidas entre diciembre de 1993 y agosto de 1996. En 2003 se publicó una versión de alta resolución, 24 bits de color, de estas imágenes. El conjunto de datos incluye 2413 imágenes faciales, representando a 856 persona. Las imágenes a color se encuentran en formato ppm, con una resolución de 256×384 pixeles. (Phillips, Wechsler, Huang, & Rauss, 1998)

3.1.2.2. FEI

La base de datos de rostros FEI, es una base de datos de libre acceso para investigación y actividades académicas, que consiste en 2800 imágenes tomadas entre junio de 2005 y marzo de 2006 en el laboratorio de inteligencia artificial de São Bernardo do Campo, São Paulo, Brasil. Contiene 14 imágenes a color de 640×480 pixeles (10 posiciones e imágenes frontales con diferentes expresiones y condiciones de iluminación) por cada uno de los 200 individuos participantes, para el total de 2800 imágenes. Las imágenes recogen un número igual de mujeres y hombres (Pesquisa, Leonel, & Junior, 2005)

3.2. Implementación

El rostro humano puede cambiar significativamente debido a diferentes condiciones de iluminación, expresiones faciales, oclusión, rotaciones, etc. Recientemente se han efectuado un gran número de investigaciones encaminadas a desarrollar sistemas de reconocimiento que reconozcan rostros bajo varias de estas condiciones y en especial bajo rotaciones fuera del plano (X. Zhang & Gao, 2009)

En este trabajo se propone desarrollar en primera instancia un sistema de estimación de poses capaz de clasificar imágenes de rostros humanos en siete poses básicas definidas según su ángulo de rotación horizontal, con respecto al eje de la cámara.

En una segunda instancia se evaluará el sistema de estimación de poses como parte de un prototipo de un sistema de reconocimiento de rostros que pueda trabajar dentro del rango de rotaciones cubierto por las siete poses básicas. El sistema de reconocimiento se implementará como una extensión del algoritmo de reconocimiento por rostros propios o reconocimiento por *Eigenfaces*.

3.2.1. Sistema de estimación de poses

Los rostros utilizados para la etapa de entrenamiento del sistema de estimación de poses se obtuvieron de dos bases de datos de imágenes de rostros de uso público: la base de datos FERET (Phillips et al., 1998) y la base de datos FEI (Pesquisa et al., 2005)

Después de una primera etapa de preprocesamiento de las imágenes, estas imágenes se alimentan a una red neuronal de convolución de seis capas (entrada, tres capas de convolución, una capa de submuestreo y una capa de salida tradicional con conexión completa) encargada de categorizar las imágenes en una de siete posiciones básicas: frontal cuarto de perfil, medio perfil y perfil completo, izquierdo y derecho respectivamente.

Con el fin de evaluar si los parámetros de la red neuronal de convolución no se ajustaron solo a las particularidades de un conjunto de prueba, se buscó la configuración y los parámetros óptimos de la red utilizando solamente la base de datos FERET y posteriormente se utilizó estos mismos parámetros para entrenar la red con las imágenes de la base de datos FEI y evaluar los resultados sobre un segundo conjunto de prueba.

Para el ajuste de parámetros se utilizó 2000 imágenes de entrenamiento y 100 imágenes de prueba tomadas de la base de datos FERET. Posteriormente se evaluó el error mínimo que se puede alcanzar tras 5 repeticiones del proceso de entrenamiento y evaluación de la red, para un mismo conjunto de entrenamiento de 2400 imágenes y 200 de prueba, tomadas de la base FEI.

Con el fin de evaluar el impacto del tamaño de imagen alimentado a la red, se entrenó la red neuronal para tamaños de 33, 41, 65 y 81 pixeles, con conjuntos de 2400 imágenes de la base de datos FEI, y se evaluó con conjuntos de 200 imágenes de la misma base de datos. Para cada uno de las 30 evaluaciones realizadas por cada tamaño, los conjuntos de entrenamiento y prueba se generaron aleatoriamente.

La proporción de las imágenes frontales en las bases de datos utilizadas es aproximadamente de 3 a 4 veces mayor a la de imágenes en otros ángulos.

El algoritmo utilizado para el entrenamiento fue el algoritmo de propagación hacia atrás junto al método estocástico diagonal de Levenberg Marquadt para la optimización de la tasa de aprendizaje η.

La arquitectura de la red neuronal está basada en el trabajo de reconocimiento de dígitos con redes neuronales de convolución desarrollado por LeCun y las sugerencias para optimizar redes neuronales del artículo “Propagación hacia atrás eficiente” del mismo autor (LeCun et al., 1998). La implementación se basa en el trabajo sobre redes neuronales de convolución para

reconocimiento de dígitos de O'Neill (2006), cuyo código se encuentra disponible bajo licencia de código abierto MIT X11.

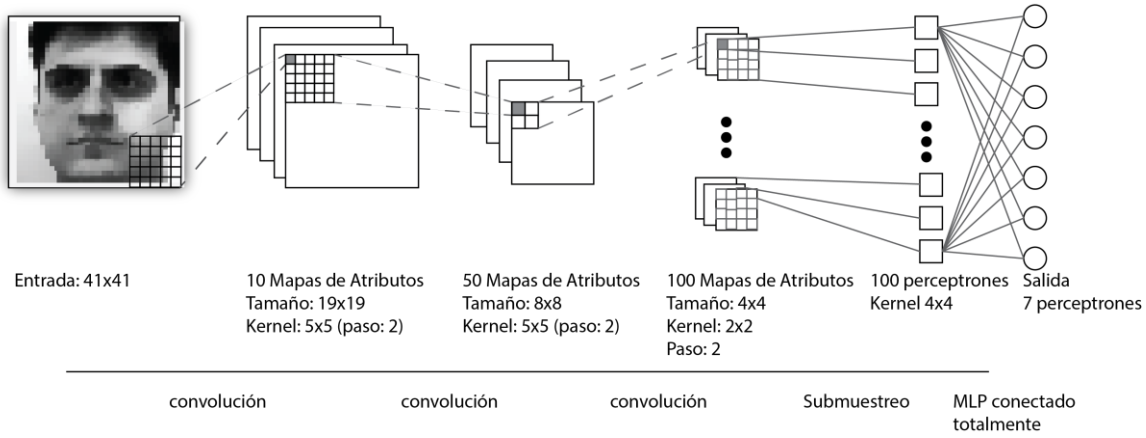

Figura 4. Arquitectura de la red neuronal de convolución de seis capas utilizada para la estimación de pose horizontal

3.2.2. Sistema de reconocimiento

Para cada una de estas rotaciones se entrenó un sistema de reconocimiento por rostros propios que es el encargado del reconocimiento final. Para este sistema se utilizó como conjunto de aprendizaje 2600 imágenes de la base de datos FERET y un conjunto de prueba de 200 imágenes. El proceso de entrenamiento y evaluación se repitió 30 veces, con conjuntos generados de forma aleatoria.

3.2.3. Pre-procesamiento de las imágenes

La implementación de los algoritmos de pre-procesamiento de las imágenes y de reconocimiento por rostros propios se basa en código proporcionado en los tutoriales de la librería *OpenCV* y otras fuentes de información similares. En todos los casos el código se encuentra disponible bajo licencia BSD o compatible, que permite su libre uso y modificación con fines académicos.

La clasificación y preparación previa de los conjuntos de imágenes se realizó utilizando scripts desarrollados en Python. La implementación del sistema de reconocimiento y la red neuronal de convolución se realizó en C++, utilizando la librería para sistemas de visión artificial *OpenCV* (Bradski & Kaehler, 2008)

Dado que el presente trabajo está enfocado en la etapa de reconocimiento, no en la detección, se utilizó como entrada los rostros detectados y enmarcados por el conocido algoritmo de detección de rostros en cascada desarrollado por Viola y Jones (2001) e implementado en la librería *OpenCV*.

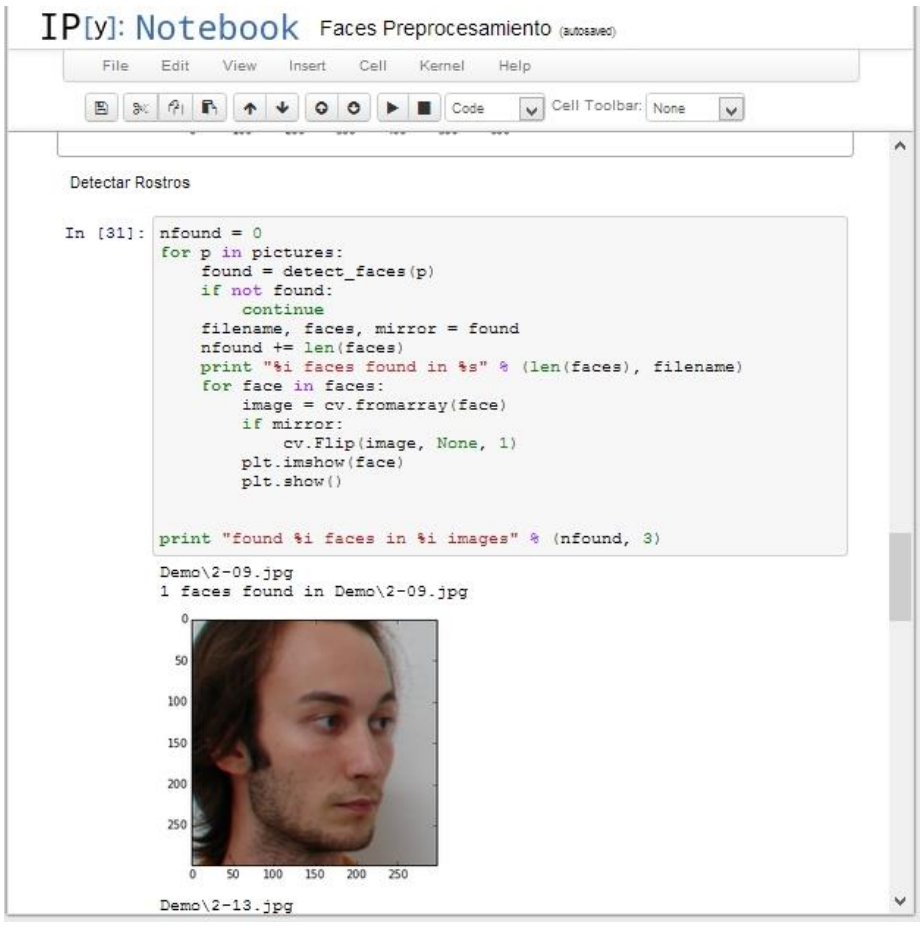

Figura 5. Cuaderno de IPython utilizado para la preparación y clasificación previa de imágenes

Adicionalmente es necesario señalar que este trabajo se centrará solamente en cambios en el ángulo de rotación horizontal fuera del plano con respecto al eje de la cámara.

Las imágenes de dimensiones 256×384 para la base de datos FEI y 640×480 para la base de datos FERET se normalizaron a imágenes en escala de grises con tamaños estándar de 33×33, 41×41, 65×65 y 81×81 píxeles cambiando la escala y recortando la imagen según fue necesario para conservar la proporción original del rostro.

A continuación, se realizó una ecualización de histograma para mejorar el contraste y el brillo de las imágenes. Esta etapa ayuda a reducir la variación debido a condiciones diferentes de

Figura 6. Imagen antes (a) y después (b) de la detección y el procesamiento (tomada de la base de datos FEI)

iluminación, y es importante para mejorar el desempeño de algoritmos basados en extraer características del rostro, como es el caso del algoritmo PCA Moon y Phillips (2001)

Las imágenes resultantes se convierten en arreglos unidimensionales para facilitar la alimentación de los datos a las siguientes etapas.

Figura 7. Ejemplo de imágenes de perfil completo, medio perfil, cuarto de perfil y frontal tomadas de la base de datos FEI

Finalmente, para preparar los datos de entrenamiento para el sistema de estimación de pose horizontal y el sistema de reconocimiento por rostros propios específicos para cada vista, se separaron las imágenes en siete conjuntos según las siguientes categorías: frontal, cuarto de perfil izquierdo y derecho, medio perfil izquierdo y derecho y perfil completo izquierdo y derecho. Las imágenes de perfil corresponden aproximadamente a ángulos de rotación horizontal respecto a la posición frontal de $\pm 22.5^{o}$, $\pm 67.5^{o}$ y $\pm 90^{o}$ para el cuarto de perfil, medio perfil y perfil completo respectivamente.

Capítulo 4

EXPERIMENTACIÓN Y DESARROLLO

El desarrollo de la red neuronal para la estimación de pose se realizó a lo largo de varios meses, y constituye el principal componente de este trabajo. Los componentes de reconocimiento y detección facial, se desarrollaron empleando algoritmos más simples que utilizan en gran medida funciones de las librerías *OpenCV*. Aunque estos algoritmos funcionan bien para propósitos de evaluación del sistema de estimación de pose, es claro que pueden ser sujeto de sensibles mejoras en trabajos posteriores.

Para las pruebas se utilizaron como plataformas de desarrollo una computadora portátil con sistema operativo Microsoft Windows, procesador Intel I5 a 2.5 GHz, 4Gb de memoria RAM y

compilador MinGW 4.8 de 32 bits y una computadora portátil MacBook Pro, con procesador Intel I7 a 2.3 GHz, y 8 Gb de memoria RAM, con compilador CLANG - LLVM 3.3 de 64 bits.

4.1. Ajuste de parámetros iniciales

Los parámetros iniciales tales como las dimensiones de las imágenes alimentadas a la red neuronal y el pre-procesamiento de las mismas se desarrollaron tomando como base trabajos previos, tanto en reconocimiento de rostros como estimación de poses. Al definir estos parámetros se buscó un balance entre la cantidad de información, y los requerimientos de memoria y tiempo de procesamiento. Los trabajos de referencia, presentados en la introducción teórica, se utilizaron también para definir el tamaño y proporción de los conjuntos de prueba y entrenamiento.

A fin de construir los conjuntos de prueba y entrenamiento se realizó primero una prueba para determinar las imágenes que podían ser detectadas correctamente por el algoritmo de detección de rostros. Las tasas de detección alcanzadas para las dos bases de datos empleadas se resumen en la tabla 1

Tabla 1. Tasas de detección según la base de datos usada

Base de datos	Imágenes Totales	Imágenes detectadas	Tasa de detección
FERET	2807	2790	99.4 %
FEI	2800	2760	98.6 %

Utilizando las imágenes detectadas correctamente y que por tanto pueden ser sujetas al pre-procesamiento indicado, se elaboró conjuntos de entrenamiento de aproximadamente 2000 y 2400 imágenes y conjuntos de prueba de aproximadamente 100 y 200 imágenes, para la primera etapa de evaluación. Para una segunda etapa de evaluación de la tasa de reconocimiento según el tamaño de imagen de entrada se utilizaron conjuntos de entrenamiento de aproximadamente

2600 imágenes y de prueba de 200 imágenes de la base de datos FEI, seleccionadas de forma aleatoria.

Para la evaluación del sistema de reconocimiento, se utilizaron conjuntos de entrenamiento de aproximadamente 2600 imágenes y de prueba de 200 imágenes extraídas de la base de datos FERET, de forma aleatoria.

4.2. Ajuste de la Red Neuronal de Convolución

La red neuronal de convolución utilizada para la estimación de poses se desarrolló a partir de un prototipo de red neuronal simple con una capa oculta y conexiones completas. Ese prototipo se fue mejorando gradualmente buscando aumentar tanto su rendimiento como su tasa de estimación correcta de poses.

Tomando como entrada imágenes con el pre-procesamiento descrito en la sección anterior, la primera versión de la red neuronal, alcanzó tasas de error cercanas al 40 %, al aumentar una capa de convolución, esta tasa de error mejoró hasta situarse alrededor del 30 %.

Para esta primera etapa encaminada a definir la estructura de la red neuronal se marcó como objetivo alcanzar un error cercano o inferior al 20 %, para después proceder a un ajuste más fino de los parámetros.

La estructura final de seis capas descrita en la sección anterior, permitió alcanzar errores cercanos al 15 % sin ningún procedimiento de refinación de sus parámetros., por tanto se aceptó como base para un ajuste más exhaustivo de sus parámetros.

Con el fin de evaluar si los parámetros y la configuración de la red neuronal de convolución no se ajustan solo a las particularidades de un conjunto de prueba, se realizó las optimizaciones anteriormente descritas utilizando solamente la base de datos FERET y posteriormente se

utilizó estos mismos parámetros para entrenar la red con las imágenes de la base de datos FEI y evaluar los resultados sobre un segundo conjunto de prueba.

4.2.1. Ajuste de parámetros

A diferencia de trabajos con redes neuronales estándar en los que se utiliza un algoritmo adicional para la extracción de las características a ser alimentadas al sistema, la red neuronal de convolución permite delegar la tarea de seleccionar las características importantes a las capas iniciales de convolución de la red. En estas, la importancia de las características se refleja en los pesos del *kernel* (núcleo) de cada mapa de características.

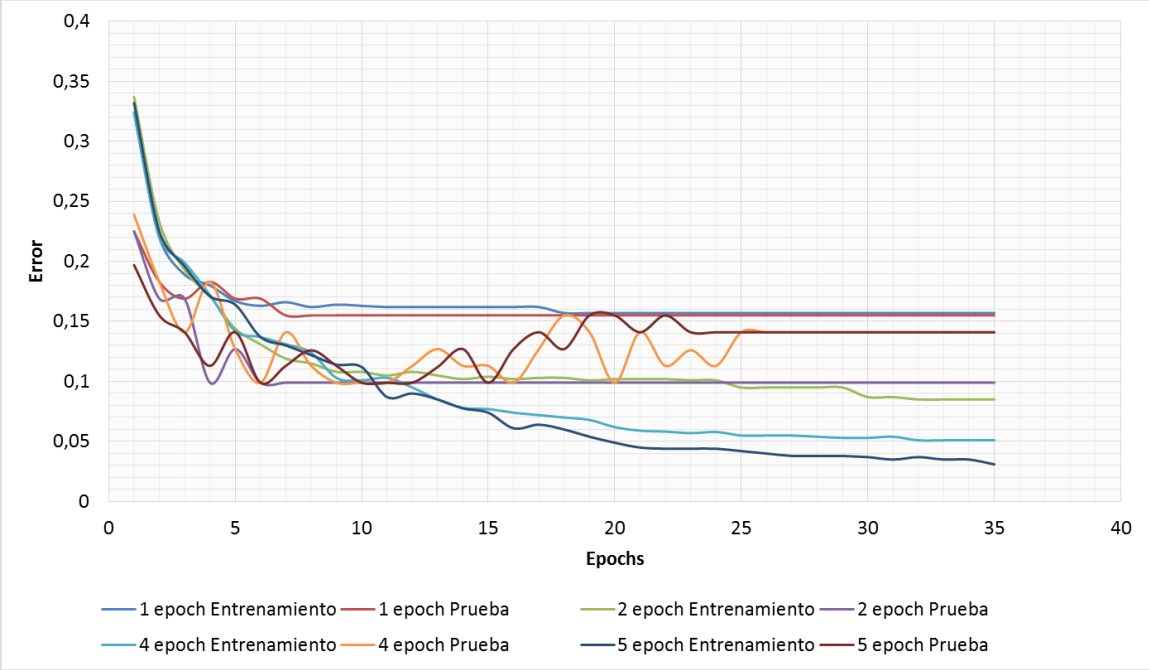

Figura 8. Relación entre la tasa de actualización de η y la tasa de reconocimiento en imágenes de la base de datos FERET

Como se describió en el capítulo anterior el parámetro η o tasa de aprendizaje se modifi- ca dinámicamente a fin de reducir el sobre aprendizaje. Por tanto las primeras pruebas se destinaron a determinar los valores iniciales para este parámetro. En la figura 8 se muestra la dependencia entre la frecuencia de actualización del parámetro η y la tasa de error en la

estimación de poses, para un η inicial de 0.0005, tanto sobre el conjunto de entrenamiento como sobre el conjunto de prueba.

A pesar de que una velocidad de ajuste más lento (mas *epochs* entre cada actualización) disminuye el error en el conjunto de entrenamiento, esta disminución ocurre en el conjunto de prueba solo hasta una tasa de 2 *epochs* (propagaciones hacia adelante completas sobre todas las imágenes del conjunto de entrenamiento). Adicionalmente se puede concluir que por debajo de una tasa de actualización de 5 *epochs*, el sistema se estabiliza antes de las 10 iteraciones.

En la figura 9, se muestra la dependencia de la tasa de error en la estimación de poses con respecto al valor inicial del parámetro η.

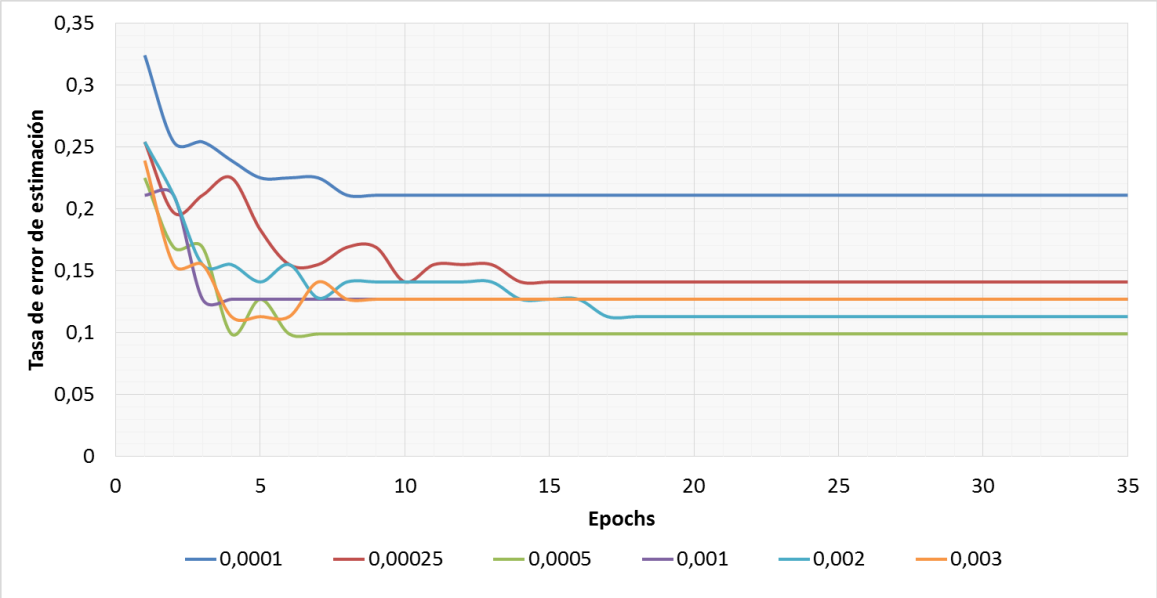

Figura 9. Efecto del η inicial sobre el reconocimiento en el conjunto de prueba de la base de datos FERET

Se puede apreciar en la figura 11 que la función de error presenta dos mínimos con respecto a η en el intervalo de 0.0001 a 0.005. El primer mínimo presenta valores uniformes en la tasa de error para ambos conjuntos. El segundo mínimo muestra algo de sobre-ajuste ya que el mínimo en el conjunto de entrenamiento es mucho menor al mínimo en el conjunto de prueba.

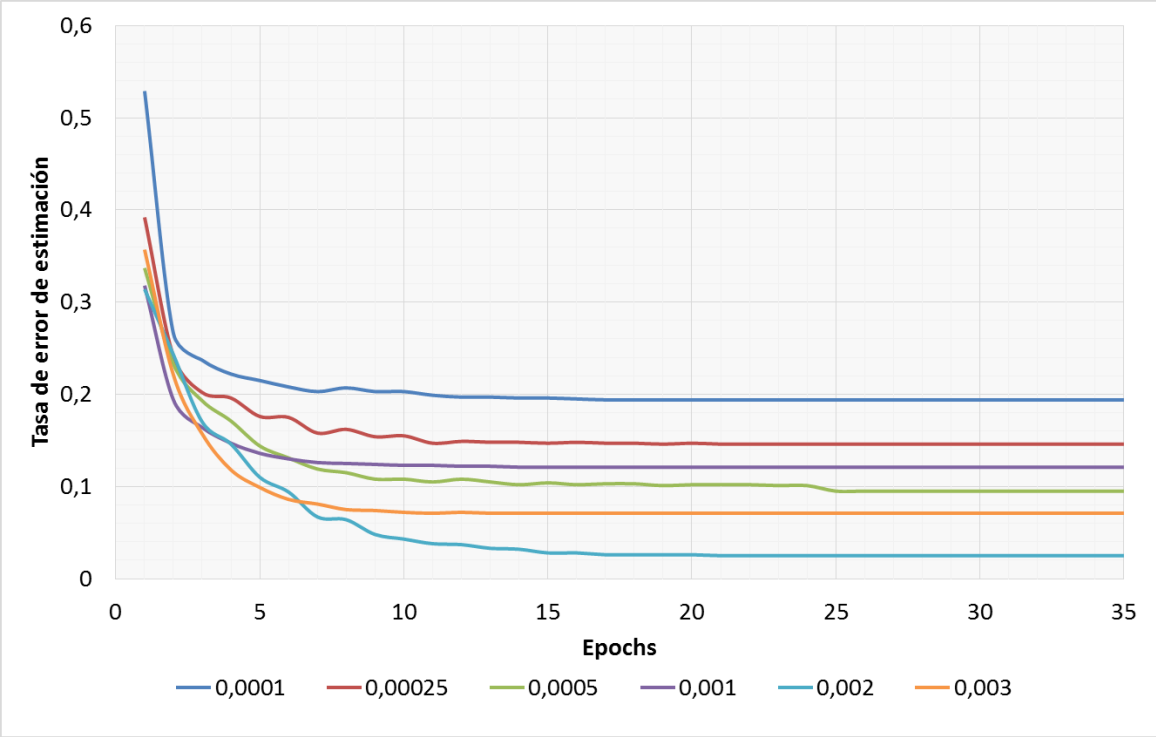

Figura 10. Efecto del η inicial sobre el reconocimiento en el conjunto de entrenamiento de la base de datos FERET

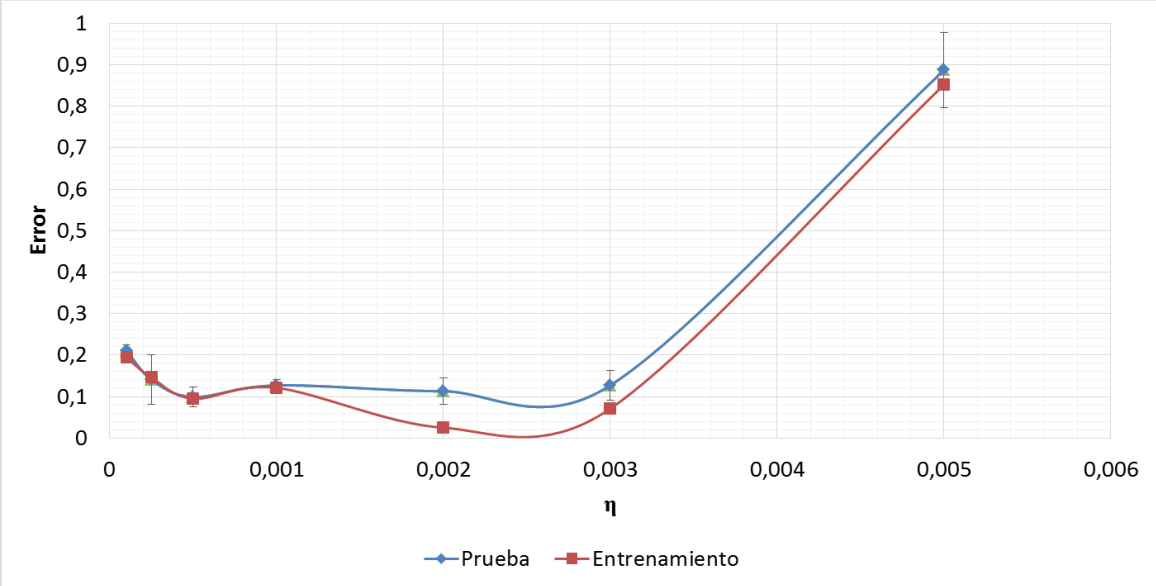

Figura 11. Relación entre el η inicial y la tasa de error

Por otro lado valores mayores de η producen un mayor error en el reconocimiento en ambos conjuntos, y valores aún mayores conducen a un sistema que no converge en un mínimo (el error aumenta en cada iteración).

Aunque los dos mínimos producen tasas de error similares sobre ambos conjuntos, se decidió utilizar como valor óptimo para el parámetro η una tasa inicial de 0.0005, dado que presentaba una tasa de error ligeramente menor y menor desviación estándar, como se aprecia en la figura 11.

Adicionalmente se condujeron experimentos que demostraron similares resultados al acoplar el sistema de estimación al de reconocimiento, una vez entrenado con cualquiera de los dos mínimos.

Capítulo 5

RESULTADOS

El sistema de estimación de poses y el sistema de reconocimiento se desarrollaron de forma modular, para facilitar su portabilidad en inclusión en otros sistemas, así como su modificación para su utilización en pruebas futuras. Gracias a la disponibilidad para diversas plataformas de las librerías OpenCV, todos los programas utilizados en este trabajo pueden ser compilados tanto en sistemas Windows, como MacOS y Linux de 32 y 64 bits.

5.1. Sistema de estimación de pose

En una primera evaluación el sistema de estimación de pose se evaluó tanto sobre la base de datos FERET como la base de datos FEI. Para la base de datos FERET se formó de forma aleatoria un conjunto de entrenamiento constituido por 2000 imágenes y un conjunto de prueba

formado por 100 imágenes distribuidas en 10 para cada perfil completo, 15 para medio perfil y cuarto de perfil y 30 imágenes frontales. La tasa de error alcanzada en esta base de datos (se reporta la mejor de 5 intentos) fue del 9.86 %, sin embargo si se contabiliza solamente los errores que difieren de la pose real con una distancia mayor a 1, el error fue del 3.37 %.

Tabla 2. Porcentaje de error en la estimación de poses según la base de datos

Base de Datos	Imágenes de Entrenamiento	Imágenes de Prueba	Errores Totales	Errores Mayores a una posición
FERET	2000	100	9.86%	3.37%
FEI	2400	190	11.4%	2.17%

Para la base de datos FEI se utilizó un conjunto de entrenamiento de 2400 imágenes y un conjunto de prueba de 190 imágenes, distribuidas en 10 imágenes para cada perfil completo, 15 para medio perfil, 30 para cuarto de perfil y 80 imágenes frontales. Utilizando un método similar al del caso anterior, sobre esta base de datos se alcanzó una tasa de error del 11.4%. Nuevamente si se contabiliza solo errores con una distancia mayor a 1 con respecto a la pose correcta, el error es de 2.17%.

5.2. Sistema de Reconocimiento de rostros

El sistema de reconocimiento de rostros con detección previa de pose, se evaluó sobre conjuntos de entrenamiento de 2590 imágenes y conjuntos de prueba de 200 imágenes seleccionas aleatoriamente. Los datos reportados son el resultado del promedio de 30 repeticiones. La tasa de reconocimiento alcanzada por el algoritmo de rostros propios con y sin etapa de clasificación por pose se presenta en la tabla 3.

En la tabla 4 se presenta la tasa de reconocimiento en función de la pose a la que pertenecía la imagen de prueba, para el reconocimiento con el algoritmo simple de rostros propios y usando el sistema con estimación de pose.

Tabla 3. Tasa de reconocimiento alcanzada con y sin estimación de poses (entre paréntesis se presenta el error estándar)

Método de Reconocimiento	Tasa de Reconocimiento
Rostros Propios	40.5% (0.7)
Rostros Propios + estimación de pose	61.4% (1.1)

Para cada caso el conjunto de prueba estuvo constituido aproximadamente de 20 imágenes de perfil, 30 de medio perfil, 30 de cuarto de perfil, izquierdo y derecho; y 40 frontales.

Tabla 4. Tasa de reconocimiento alcanzada con y sin estimación de poses (entre paréntesis se presenta el error estándar), para cada perfil.

Perfil	Rostros Propios [%]	Rostros Propios + CNN [%]
Perfil izquierdo	26.1 (2.4)	39.7 (3.9)
Medio perfil izquierdo	25.4 (2.6)	46.9 (2.2)
Cuarto perfil izquierdo	54.1 (3)	53.9 (2.7)
Frontal	62.8 (2.1)	78.6 (2.3)
Cuarto perfil derecho	32.4 (2.0)	61.8 (2.2)
Medio perfil derecho	39.4 (3.6)	61.7 (3.8)
Perfil derecho	26.7 (1.8)	42.7 (3.9)

El añadir el reconocimiento de perfil como una etapa previa al reconocimiento de rostros permitió disminuir la tasa de error para todos los perfiles excepto las imágenes de medio perfil izquierdo. Una explicación para estos resultados puede ser el que la clasificación previa de las imágenes ayuda a reducir el ruido introducido al mezclar poses en el conjunto de entrenamiento.

La mejor tasa de reconocimiento se registró sobre las imágenes frontales donde usualmente se ha reportado buenos resultados para el algoritmo de rostros propios. La menor tasa de reconocimiento para los dos métodos evaluados corresponde al perfil completo, donde en general se dispone de menos características del rostro para distinguir entre diferentes individuos.

De acuerdo a investigaciones previas el algoritmo de reconocimiento por PCA, puede lograr el reconocimiento aún con variaciones en ángulo horizontal cercanas a 20◦ (X. Zhang & Gao, 2009), lo cual explicaría que el reconocimiento con este método sea aceptable para las imágenes de cuarto de perfil, pero no para las de perfil completo.

En general, el sistema de reconocimiento propuesto se puede utilizar para el reconocimiento de individuos cooperadores de los cuales sea posible capturar el conjunto de imágenes necesarias para entrenar el sistema. El componente de estimación de pose, puede trabajar, aunque posiblemente con menor precisión, sin imágenes previas del individuo, pero un conjunto completo de imágenes para todos los perfiles son indispensables si se quiere efectuar el reconocimiento.

Aun cuando se logró mejorar el reconocimiento de forma significativa con la adición de la etapa de clasificación de pose, este sistema debe considerarse como una aproximación simple cuyo principal objetivo es evaluar el impacto que puede tener la inclusión de la clasificación previa según perfil, en el reconocimiento. En este sentido el sistema propuesto puede ser sujeto de múltiples mejoras, en casi todas las etapas del proceso. También es necesario una mayor investigación respecto a los efectos de la iluminación y expresiones faciales tanto en el sistema de estimación de poses como en el de reconocimiento.

A pesar de que el entrenamiento de la red neuronal es un proceso que toma un tiempo considerable, una vez entrenada para el reconocimiento de poses, su utilización sobre imágenes de prueba es suficientemente eficiente como para su inclusión en aplicaciones de tiempo real. En la figura 12 se puede ver una captura de una aplicación prototipo para la estimación de poses en tiempo real, basado en el algoritmo propuesto.

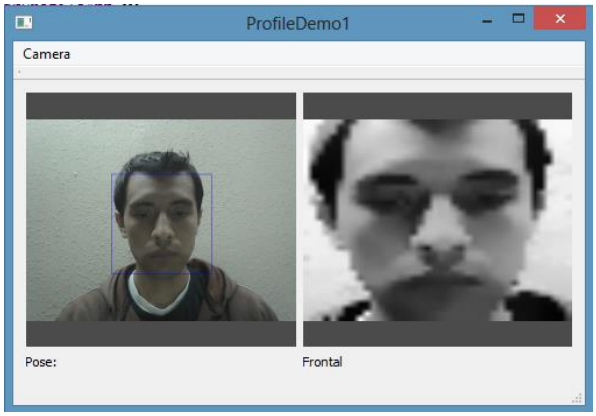

Figura 12. Prototipo de aplicación para la estimación de pose en tiempo real

Aunque el prototipo presenta resultados aceptables bajo condiciones óptimas de iluminación, una implementación completa de un sistema de reconocimiento en tiempo real, requiere un mayor trabajo tanto en el componente de detección como el de reconocimiento de pose, sobre todo con respecto a su robustez a diferentes condiciones de iluminación y expresión facial.

5.3. Tiempo de ejecución y tamaño de las imágenes de entrada

En la tabla 5 se resume los resultados alcanzados por la red neuronal, bajo distintos tamaños de imagen de entrada. Para cada tamaño de imagen se entrenó la red con un total de 2560 imágenes y se evaluó con un conjunto de prueba de 200 imágenes seleccionadas aleatoriamente durante cada una de las 30 iteraciones efectuadas.

Para cada tamaño se ajustaron las dimensiones de los mapas utilizados en las capas de convolución de forma proporcional.

Los parámetros tales como la tasa de aprendizaje inicial y el número de mapas en cada capa, se conservaron iguales.

Tabla 5. Tiempo de ejecución y tasa de error de la red neuronal para diferentes tamaños de imagen, todos los tiempos se miden en segundos y los errores corresponden al promedio de 30 repeticiones (junto a las tasas de error se encuentra el error estándar)

	Tiempo			**%Error**	
Tamaño	Preprocesamiento (100 imágenes)	Entrenamiento (20 Epochs)	Pruebas (100 imágenes)	Entrena-miento	Prueba
33 × 33	162.8	2036.8	1.02	11.1 (0.2)	15.9 (0.4)
41 × 41	161.5	3502.9	1.75	7.7 (0.4)	15.5 (0.4)
65 × 65	154.8	10172.1	5.27	5.9 (0.1)	17.7 (0.4)
81 × 81	162.1	16316.4	8.46	1.8 (0.4)	20.5 (1.7)

En general se puede apreciar que la tasa de error sobre el conjunto de entrenamiento disminuye conforme aumenta el tamaño de la imagen. Sin embargo, aunque la reducción en el error alcanzado sobre el conjunto de entrenamiento es considerable para tamaños de imagen mayores, el error aumenta sobre el conjunto de prueba, probablemente debido a que al alimentar a la red imágenes más grandes aumenta el número de pesos por ajustar y por tanto se acentuó el problema del sobre-ajuste.

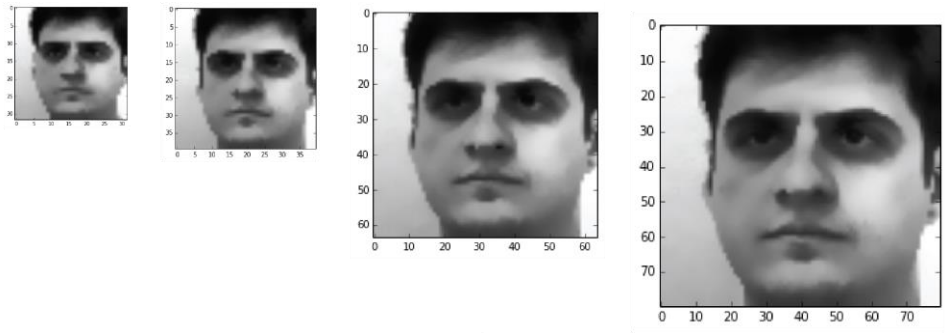

Figura 13. Ejemplos de imágenes en varios tamaños: 33×33, 41×41, 65×65 y 81×81

Otro aspecto interesante que muestran los resultados anteriores es que el tiempo necesario para cada ronda de entrenamiento crece solo linealmente. Por tanto una alternativa promete- dora para trabajos futuros sería el introducir métodos adicionales de control del sobre-ajuste (ver recomendaciones al final) con el fin de aprovechar la reducción en las tasas de error que se puede alcanzar al aumentar el tamaño de imagen.

Tabla 6. Tiempos de entrenamiento y prueba para el algoritmo de rostros propios con diferentes tamaños de imágenes

Tamaño	Entrenamiento [s]	Prueba [s]
33 × 33	104,2	1,20
41 × 41	414,9	2,08
65 × 65	2017,3	5,98
81 × 81	2102.5	8.56

Como referencia en la tabla 6, se muestran los tiempos de entrenamiento y prueba para el algoritmo de rostros propios, aplicado sobre las mismas imágenes de la base de datos EFI y compilado y ejecutado bajo el mismo entorno. Aunque el tiempo de entrenamiento es considerablemente mayor para el caso de la red neuronal. Se puede ver que una vez que esta ha sido entrenada su tiempo de ejecución para la estimación sobre el conjunto de prueba no difiere mucho del tiempo requerido para el reconocimiento por el algoritmo de rostros propios. Por

tanto se puede estimar razonable el consumo extra de recursos ocasionado por su adición como una etapa extra en un sistema de reconocimiento.

Capítulo 6

CONCLUSIONES Y RECOMENDACIONES

6.1. Conclusiones

En el pasado, la mayor parte de las investigaciones realizadas en el área del reconocimiento de rostros se han centrado en el reconocimiento sobre imágenes frontales.

Sin embargo, en los últimos años el interés se ha extendido al trabajo sobre imágenes fuera de ambientes controlados, donde el objetivo central es aumentar la robustez de los algoritmos empleados frente a distintas condiciones de resolución, escala, iluminación, expresión facial y pose.

El reconocimiento de pose tanto horizontal como vertical resulta importante en sistemas de reconocimiento con sujetos cooperadores, pero más aún para sistemas que buscan el

reconocimiento en sujetos no cooperadores, donde es imprescindible desarrollar todo el potencial del reconocimiento de rostros como una técnica biométrica de naturaleza no intrusiva.

En el presente trabajo se ha buscado explorar la utilidad de las redes neuronales de convolución como un método para la estimación de poses horizontales, alcanzando como resultado un algoritmo que permite una tasa de estimación correcta de poses aceptable para las dos bases de datos de prueba estudiadas.

De la misma forma se ha podido demostrar como este sistema permite mejorar el reconocimiento del algoritmo simple de rostros propios, al construir un sistema que utiliza la categorización en diferentes poses, como un paso previo al reconocimiento.

Aun cuando es necesario un estudio más amplio que incluya variaciones de pose verticales, y condiciones distintas de iluminación y expresiones faciales, este último resultado, sumado al prototipo de reconocimiento en tiempo real que se desarrolló al final de este trabajo, sientan las bases de un trabajo posterior que profundice en estas ideas y permita alcanzar un algoritmo de reconocimiento en tiempo real robusto a distintas poses.

Un aspecto que permitiría mejorar los resultados alcanzados durante el reconocimiento podría ser el encontrar algoritmos que permitan el escalado y localización uniforme de características del rostro (ojos, nariz, boca) bajo distintos ángulos de rotación.

Este paso es evidentemente más sencillo para aplicaciones de reconocimiento de rostros frontales donde normalmente se realiza de forma manual o de forma automática utilizando la posición y la distancia entre los ojos (Zhao, Chellappa, Phillips, & Rosenfeld, 2003)

En el presente trabajo esta corrección automática de tamaño y posición no fue posible, ya que los algoritmos de detección estándar basados en cascada utilizados para la detección de las características del rostro, tienen dificultades para detectar la posición de los ojos en rostros con

rotaciones horizontales pronunciadas. Bajo estas condiciones usualmente la nariz proyecta cierta sombra sobra los ojos, la cual disminuye considerablemente la precisión de su detección o la impide completamente. Más aún, para imágenes de rostros de perfil completo no es posible detectar ambos ojos y por tanto no es posible usar el procedimiento usual que emplea la distancia entre estos para escalar las imágenes.

La ausencia de este tratamiento previo de las imágenes puede ser una de las razones por las cuales la eficiencia del reconocimiento en rostros frontales es menor a la que se encuentra en otros trabajos que utilizan el algoritmo de rostros propios para el reconocimiento de rostros exclusivamente en posición frontal. En estos trabajos la precisión alcanzada llega incluso a superar el 95 % (C. Zhang & Zhang, 2010)

6.2. Recomendaciones

A pesar de su gran potencialidad los algoritmos de redes neuronales de múltiples capas presentan grandes retos.

En casi todos los usos prácticos de este tipo de algoritmos, la función objetivo es una función altamente no convexa en sus parámetros, con el potencial de presentar muchos mínimos locales. Esto introduce un serio problema ya que no todos los mínimos presentan tasas de error comparables. Como consecuencia en múltiples casos las técnicas usuales basadas en inicialización aleatoria de los parámetros, presentan un pobre desempeño.

En los últimos años se han presentado diferentes enfoques para solucionar este problema. La más importante de estas publicada por Hinton y Srivastava (2012) propone pre-entrenar cada capa con un algoritmo de aprendizaje no supervisado, que les permita aprender una transformación lineal de sus entradas que capture las variaciones principales. El pre-

entrenamiento es seguido por una etapa final donde la arquitectura se ajusta respecto a un criterio supervisado utilizando optimización basada en el gradiente.

Esta estrategia ha reportado mejoraras importantes en los algoritmos de aprendizaje de múltiples capas, sin embargo sus mecanismos subyacentes aún son objeto de estudio.

Otro problema presente al entrenar redes neuronales es el "sobre-ajuste". Este es usualmente más crítico, si el conjunto de entrenamiento es limitado, dado que los vectores de pesos tienden a usar dependencias entre detectores para ajustarse casi perfectamente al conjunto de entrenamiento, lo cual posteriormente implica un mal desempeño sobre el conjunto de prueba.

Recientemente se ha propuesto como solución el procedimiento conocido como *dropout*, que consiste en omitir aleatoriamente algunos de los detectores de atributos en cada iteración de entrenamiento. De esta forma se reduce las co-adaptaciones complejas en las cuales, un determinado detector de atributos se vuelve útil solo en el contexto de muchos otros detectores de atributos específicos.

Sobre la base de datos de imágenes de escritura a mano MNIST, el *dropout* permitió una reducción en la tasa de error para una red de convolución de 5 capas cercana al 19 %. Sobre la base de datos TIMIT para reconocimiento de voz, y una red neuronal de 4 capas conectadas completamente, se consiguió una reducción de la tasa de error sobre un conjunto de prueba de 22.7 % a 19.7 % (Srivastava, 2013)

En general los autores argumentan que independientemente de la arquitectura el *dropout* permite obtener mejoras moderadas en la tasa de error sobre el conjunto de prueba.

Trabajos posteriores han mejorado esta idea al utilizar el *dropout* como una técnica para promediar modelos y combatir el sobre-ajuste. En este contexto se puede ver el *dropout* en cada

actualización como la ejecución de un modelo diferente sobre un subconjunto diferente del conjunto de entrenamiento.

El modelo *maxout* basado en esta idea es una arquitectura de propagación hacia atrás, tal como una red de perceptrón multicapa o una red de convolución, que utiliza una nueva función de activación llamada unidad *maxout*, y se entrena utilizando el modelo *dropout* (Goodfellow, Warde-Farley, Mirza, Courville, & Bengio, 2013)

Una unidad de activación *maxout* está dada, para una entrada x, por la función:

$$h_i(x) = \max_{j \in [1,k]} z_{ij} \tag{26}$$

Donde,

$$z_{ij} = x^t W_{ij} + b_{ij} \tag{27}$$

En el contexto de una red de convolución, un mapa de atributos *maxout* se puede construir tomando el máximo entre k mapas de atributos afines.

Con este método sobre la base de datos MNIST se obtuvo una tasa de error de 0.94 %, que es el mejor resultado a la fecha para algoritmos sin pre-entrenamiento no supervisado. Sobre la base de datos de imágenes CIFAR-10, utilizando una red neuronal de convolución de 3 capas se obtuvo una tasa de error del 12.93 %, que mejora considerablemente la más baja alcanzada para este tipo de algoritmos del 14.05 % (Goodfellow et al., 2013)

Apéndice A

CÓDIGO FUENTE

El código de clasificación previa de las imágenes se desarrolló utilizando scripts de Python y cuadernos de *IPython*, su ejecución ha sido probada en sistemas con la versión 2.7.5 de Python, 2.4 de *PyOpenCV*, *NumPy* 1.7.1 e *IPython* 1.0, tanto en versiones de 32 como 64 bits.

El código necesario para los sistemas de estimación de poses y el prototipo de reconocimiento de rostros, se desarrolló utilizando la versión 2.4.6 de las librerías *OpenCV* y las librerías estándar de C++11. Cada módulo del sistema se encuentra disponible como un proyecto de *QtCreator*. Previa a la compilación de los proyectos, el archivo .pro debe ser modificado con la ubicación de las librerías *OpenCV* en el sistema.

A excepción del prototipo de aplicación de reconocimiento en tiempo real, el resto de proyectos no presentan una interfaz gráfica y por tanto no requieren de las librerías Qt. El prototipo de aplicación de reconocimiento en tiempo real, requiere las librerías Qt en su versión 5.0 o superior para su compilación.

Para más información acerca de las librerías utilizadas en este trabajo, su instalación y licencias, se puede consultar los siguientes enlaces:

- OpenCV. http://opencv.org/
- Qt. http://qt-project.org/products/licensing

A.1. Pre-procesamiento

A.1.1. Leer imágenes e iniciar detectores

```
/**
 * Initializes the cascade detectors used to crop faces.
 **/
void initDetectors(CascadeClassifier &faceCascade, CascadeClassifier
&faceCascade2)
{
    // Load the Face Detection cascade classifier xml file.
    try {
        faceCascade.load(g_faceCascadeFilename);
    } catch (cv::Exception &e) {}
    if ( faceCascade.empty() ) {
        cerr << "ERROR: No se pudo abrir el detector en cascada: [" <<
g_faceCascadeFilename << "]!" << endl;
        cerr << "Copie el archivo del directorio de instalacion de OpenCV
(ej: 'C:\\OpenCV\\data\\lbpcascades.xml')" << endl;
        exit(1);
    }
    cout << "Se abrió exitosamente el detector de rostros en cascada ["
<< g_faceCascadeFilename << "]." << endl;

    // Load the Face Profile Detection cascade classifier xml file.
    try {
        faceCascade2.load(g_faceProfileCascadeFilename);
    } catch (cv::Exception &e) {}
    if ( faceCascade2.empty() ) {
        cerr << "No se pudo abrir el detector en cascada: [" <<
g_faceProfileCascadeFilename << "]!" << endl;
```

```
        cerr << "Copie el archivo del directorio de instalacion de OpenCV
(ej: 'C:\\OpenCV\\data\\haarcascade_profileface.xml')" << endl;
        exit(1);
    }
    cout << "Se abrió exitosamente el detector de imágenes de perfil ["
<< g_faceProfileCascadeFilename << "]." << endl;
}

/**
 * Generic function used to read csv data files
 **/
static void read_csv_2(const string& filename, vector<Mat>& images,
vector<int>& labels, vector<int>& lprofiles, char separator = ';')
{
    std::ifstream file(filename.c_str(), ifstream::in);
    if (!file) {
        string error_message = "El nombre de archivo esta incorrecto o no
existe";
        CV_Error(CV_StsBadArg, error_message);
    }
    string line, path, classlabel, profilelabel;
    while (getline(file, line)) {
        stringstream liness(line);
        getline(liness, path, separator);
        getline(liness, classlabel, separator);
        getline(liness, profilelabel);
        if(!path.empty() && !classlabel.empty()) {
            images.push_back(imread(path, 0));
            labels.push_back(atoi(classlabel.c_str()));
            lprofiles.push_back(atoi(profilelabel.c_str()));
        }
    }
}

/**
 * @brief preprocessData
 */
void preprocessData()
{
    //Init detectors
    CascadeClassifier faceCascade;
    CascadeClassifier faceCascade2;
    initDetectors(faceCascade, faceCascade2);

    // Read Data
    // These vectors hold the images and corresponding labels.
    vector<Mat> imageSamples;
    vector<int> imageLabels;
    vector<int> imageProfileLabels;

    vector<Mat> procImages;
    vector<int> procLabels;
    vector<int> procPLabels;

    // Read in the data. This can fail if no valid
    // input filename is given.
    cout << "Leyendo imágenes..." << endl;
    try {
```

```
        read_csv_2(fn_csv, imageSamples, imageLabels, imageProfileLabels
);
    } catch (cv::Exception& e) {
        cerr << "Error al abrir el archivo \"" << fn_csv << "\". Razon: "
<< e.msg << endl;
        // nothing more we can do
        exit(1);
    }
    cout << "Imágenes cargadas: " << imageSamples.size() << endl;

    // Quit if there are not enough images
    if(imageSamples.size() < 1) {
        string error_message = "Se necesita al menos una imagen";
        CV_Error(CV_StsError, error_message);
    }

    cout << "Iniciando preprocesamiento de las imágenes" << endl;
    auto start = std::chrono::high_resolution_clock::now();
    for (int j=0; j < imageSamples.size();j++) {
        // Find a face and preprocess it to have a standard size and
contrast & brightness.
        Rect faceRect;  // Position of detected face.
        Mat preprocessedFace = getPreprocessedFace(imageSamples[j],
faceWidth, faceCascade, faceCascade2, preprocessLeftAndRightSeparately,
&faceRect);

        if (preprocessedFace.data) {
            procImages.push_back(preprocessedFace);
            procLabels.push_back(imageLabels[j]);
            procPLabels.push_back(imageProfileLabels[j]);

        }
    }
    cout << "Se procesaron " << procImages.size() << " imágenes" << endl;
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
    cout << "Tiempo empleado en preprocesamiento: " << elapsed << "
milliseconds" << endl;

    //Save preprocessed images data
    FileStorage fs(filename, FileStorage::WRITE);
    fs << "Images" << procImages;
    fs << "Labels" << procLabels;
    fs << "Profile" << procPLabels;
    fs.release();                                       // explicit close
    cout << "Imagenes preprocessadas almacenadas en archivo: " <<
filename << endl;

}
```

A.2. Detectar Rostros

```
/*****************************************************************
*****
*   Face Recognition using Eigenfaces or Fisherfaces
```

```
*******************************************************************************
*   Referencias: Shervin Emami, 5th Dec 2012
*                http://www.shervinemami.info/openCV.html
*******************************************************************************
****/

#include "detectObject.h"                                   // Easily detect
faces
#include "opencv/cv.h"


/**
 * Search for objects such as faces in the image using the given
parameters, storing the multiple cv::Rects into 'objects'.
 * Can use Haar cascades or LBP cascades for Face Detection, or even eye,
mouth, or car detection.
 * Input is temporarily shrunk to 'scaledWidth' for much faster
detection, since 200 is enough to find faces.
 */

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
void detectObjectsCustom(const Mat &img, CascadeClassifier &cascade,
vector<Rect> &objects, int scaledWidth, int flags, Size minFeatureSize,
float searchScaleFactor, int minNeighbors)
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
{
    // If the input image is not grayscale, then convert the BGR or BGRA
color image to grayscale.
    Mat gray;
    if (img.channels() == 3) {
        cvtColor(img, gray, CV_BGR2GRAY);
    }
    else if (img.channels() == 4) {
        cvtColor(img, gray, CV_BGRA2GRAY);
    }
    else {
        // Access the input image directly, since it is already
grayscale.
        gray = img;
    }

    // Possibly shrink the image, to run much faster.
    Mat inputImg;
    float scale = img.cols / (float)scaledWidth;
    if (img.cols > scaledWidth) {
        // Shrink the image while keeping the same aspect ratio.
        int scaledHeight = cvRound(img.rows / scale);
        resize(gray, inputImg, Size(scaledWidth, scaledHeight));
    }
    else {
        // Access the input image directly, since it is already small.
        inputImg = gray;
    }

    // Standardize the brightness and contrast to improve dark images.
```

```
    Mat equalizedImg;
    equalizeHist(inputImg, equalizedImg);

    // Detect objects in the small grayscale image.
    cascade.detectMultiScale(equalizedImg, objects, searchScaleFactor,
minNeighbors, flags, minFeatureSize);

    // Enlarge the results if the image was temporarily shrunk before
detection.
    if (img.cols > scaledWidth) {
        for (int i = 0; i < (int)objects.size(); i++ ) {
            objects[i].x = cvRound(objects[i].x * scale);
            objects[i].y = cvRound(objects[i].y * scale);
            objects[i].width = cvRound(objects[i].width * scale);
            objects[i].height = cvRound(objects[i].height * scale);
        }
    }

    // Make sure the object is completely within the image, in case it
was on a border.
    for (int i = 0; i < (int)objects.size(); i++ ) {
        if (objects[i].x < 0)
            objects[i].x = 0;
        if (objects[i].y < 0)
            objects[i].y = 0;
        if (objects[i].x + objects[i].width > img.cols)
            objects[i].x = img.cols - objects[i].width;
        if (objects[i].y + objects[i].height > img.rows)
            objects[i].y = img.rows - objects[i].height;
    }

    // Return with the detected face rectangles stored in "objects".
}


/** Search for just a single object in the image, such as the largest
face, storing the result into 'largestObject'.
 * Can use Haar cascades or LBP cascades for Face Detection, or even eye,
mouth, or car detection.
 * Input is temporarily shrunk to 'scaledWidth' for much faster
detection, since 200 is enough to find faces.
 * Note: detectLargestObject() should be faster than detectManyObjects().
 */
void detectLargestObject(const Mat &img, CascadeClassifier &cascade, Rect
&largestObject, int scaledWidth)
{
    // Only search for just 1 object (the biggest in the image).
    int flags = CASCADE_FIND_BIGGEST_OBJECT;// | CASCADE_DO_ROUGH_SEARCH;
    // Smallest object size.
    Size minFeatureSize = Size(20, 20);
    // How detailed should the search be. Must be larger than 1.0.
    float searchScaleFactor = 1.1f;
    // How much the detections should be filtered out. This should depend
on how bad false detections are to your system.
    // minNeighbors=2 means lots of good+bad detections, and
minNeighbors=6 means only good detections are given but some are missed.
    int minNeighbors = 4;
```

```
    // Perform Object or Face Detection, looking for just 1 object (the
biggest in the image).
    vector<Rect> objects;
    detectObjectsCustom(img, cascade, objects, scaledWidth, flags,
minFeatureSize, searchScaleFactor, minNeighbors);
    if (objects.size() > 0) {
        // Return the only detected object.
        largestObject = (Rect)objects.at(0);
    }
    else {
        // Return an invalid rect.
        largestObject = Rect(-1,-1,-1,-1);
    }
}

/** Search for many objects in the image, such as all the faces, storing
the results into 'objects'.
 * Can use Haar cascades or LBP cascades for Face Detection, or even eye,
mouth, or car detection.
 * Input is temporarily shrunk to 'scaledWidth' for much faster
detection, since 200 is enough to find faces.
 *  Note: detectLargestObject() should be faster than
detectManyObjects().
 */
void detectManyObjects(const Mat &img, CascadeClassifier &cascade,
vector<Rect> &objects, int scaledWidth)
{
    // Search for many objects in the one image.
    int flags = CASCADE_SCALE_IMAGE;

    // Smallest object size.
    Size minFeatureSize = Size(20, 20);
    // How detailed should the search be. Must be larger than 1.0.
    float searchScaleFactor = 1.1f;
    // How much the detections should be filtered out. This should depend
on how bad false detections are to your system.
    // minNeighbors=2 means lots of good+bad detections, and
minNeighbors=6 means only good detections are given but some are missed.
    int minNeighbors = 4;

    // Perform Object or Face Detection, looking for many objects in the
one image.
    detectObjectsCustom(img, cascade, objects, scaledWidth, flags,
minFeatureSize, searchScaleFactor, minNeighbors);
```

}

A.3. Procesar Rostros

```
/*****************************************************************************
*
*   Face Recognition using Eigenfaces or Fisherfaces
*****************************************************************************
*
*   Referencias: Shervin Emami, 5th Dec 2012
*                http://www.shervinemami.info/openCV.html
```

```
*****************************************************************************
****/

#include "detectObject.h"       // Easily detect faces or eyes (using LBP
or Haar Cascades).
#include "preprocessFace.h"     // Easily preprocess face images, for
face recognition.
#include "opencv/cv.h"

/**
 * Histogram Equalize seperately for the left and right sides of the
face.
 **/
void equalizeLeftAndRightHalves(Mat &faceImg)
{
    // It is common that there is stronger light from one half of the
face than the other.

    int w = faceImg.cols;
    int h = faceImg.rows;

    // 1) First, equalize the whole face.
    Mat wholeFace;
    equalizeHist(faceImg, wholeFace);

    // 2) Equalize the left half and the right half of the face
separately.
    int midX = w/2;
    Mat leftSide = faceImg(Rect(0,0, midX,h));
    Mat rightSide = faceImg(Rect(midX,0, w-midX,h));
    equalizeHist(leftSide, leftSide);
    equalizeHist(rightSide, rightSide);

    // 3) Combine the left half and right half and whole face together,
so that it has a smooth transition.
    for (int y=0; y<h; y++) {
        for (int x=0; x<w; x++) {
            int v;
            if (x < w/4) {          // Left 25%: just use the left face.
                v = leftSide.at<uchar>(y,x);
            }
            else if (x < w*2/4) {   // Mid-left 25%: blend the left face
& whole face.
                int lv = leftSide.at<uchar>(y,x);
                int wv = wholeFace.at<uchar>(y,x);
                // Blend more of the whole face as it moves further right
along the face.
                float f = (x - w*1/4) / (float)(w*0.25f);
                v = cvRound((1.0f - f) * lv + (f) * wv);
            }
            else if (x < w*3/4) {   // Mid-right 25%: blend the right
face & whole face.
                int rv = rightSide.at<uchar>(y,x-midX);
                int wv = wholeFace.at<uchar>(y,x);
                // Blend more of the right-side face as it moves further
right along the face.
                float f = (x - w*2/4) / (float)(w*0.25f);
                v = cvRound((1.0f - f) * wv + (f) * rv);
```

```
            }
            else {                  // Right 25%: just use the right
face.
                v = rightSide.at<uchar>(y,x-midX);
            }
            faceImg.at<uchar>(y,x) = v;
        }// end x loop
    }//end y loop
}

/**
  * Create a grayscale face image that has a standard size and contrast &
brightness.
  * If 'doLeftAndRightSeparately' is true, it will process left & right
sides seperately,
  * so that if there is a strong light on one side but not the other, it
will still look OK.
  * Performs Face Preprocessing as a combination of:
  *  - smoothing away image noise using a Bilateral Filter,
  *  - standardize the brightness on both left and right sides of the
face independently using separated Histogram Equalization,
  * Returns either a preprocessed face square image or NULL (ie: couldn't
detect the face).
  * If a face is found, it can store the rect coordinates into
'storeFaceRect'.
*/
Mat getPreprocessedFace(Mat &srcImg, int desiredFaceWidth,
CascadeClassifier &faceCascade, CascadeClassifier &faceCascade2, bool
doLeftAndRightSeparately, Rect *storeFaceRect)
{
    // Use square faces.
    int desiredFaceHeight = desiredFaceWidth;

    // Mark the detected face region as invalid, in case they aren't
detected.
    if (storeFaceRect)
        storeFaceRect->width = -1;

    // Find the largest face (as frontal).
    Rect faceRect;
    Rect faceRectm;
    Rect faceRect1;
    Rect faceRect2;
    Rect faceRect1m;
    Rect faceRect2m;
    bool useMirror = false;
    // Frontal detection
    detectLargestObject(srcImg, faceCascade, faceRect1);
    // Profile detection
    detectLargestObject(srcImg, faceCascade2, faceRect2);

    if (faceRect1.width >= faceRect2.width) {
        faceRect = faceRect1;
    } else {
        faceRect = faceRect2;
    }

    //Mirror detection
```

```
    Mat mirrorImg;
    cv::flip(srcImg, mirrorImg,1);
    // Frontal detection
    detectLargestObject(mirrorImg, faceCascade, faceRect1m);
    // Profile detection
    detectLargestObject(mirrorImg, faceCascade2, faceRect2m);

    if (faceRect1m.width >= faceRect2m.width) {
        faceRectm = faceRect1m;
    } else {
        faceRectm = faceRect2m;
    }
    if (faceRectm.width > faceRect.width) {
        useMirror = true;
        faceRect = faceRectm;
    }

    // Check if a face was detected.
    if (faceRect.width > 0) {
        // Give the face rect to the caller if desired.
        if (storeFaceRect) {
            *storeFaceRect = faceRect;
        }
        Mat faceImg;
        if (useMirror) {
            cv::flip(mirrorImg(faceRect), faceImg,1); // Get the detected
face image
        } else {
            //faceImg = mirrorImg(faceRect);    // Get the detected face
image.
            cv::flip(mirrorImg(faceRect), faceImg,1);
        }

        // If the input image is not grayscale, then convert the BGR or
BGRA color image to grayscale.
        Mat gray;
        if (faceImg.channels() == 3) {
            cvtColor(faceImg, gray, CV_BGR2GRAY);
        }
        else if (faceImg.channels() == 4) {
            cvtColor(faceImg, gray, CV_BGRA2GRAY);
        }
        else {
            // Access the input image directly, since it is already
grayscale.
            gray = faceImg;
        }


        // Scale the image to the desired size
        Mat warped = Mat(desiredFaceHeight, desiredFaceWidth, CV_8U,
Scalar(128)); // Clear the output image to a default grey.
        resize(gray, warped, warped.size());
        //imshow("warped", warped);

        // Give the image a standard brightness and contrast, in case it
was too dark or had low contrast.
        if (!doLeftAndRightSeparately) {
```

```
            // Do it on the whole face.
            equalizeHist(warped, warped);
        }
        else {
            // Do it seperately for the left and right sides of the face.
            equalizeLeftAndRightHalves(warped);
        }
        //imshow("equalized", warped);

        // Use the "Bilateral Filter" to reduce pixel noise by smoothing
the image, but keeping the sharp edges in the face.
        Mat filtered = Mat(warped.size(), CV_8U);
        bilateralFilter(warped, filtered, 0, 20.0, 2.0);
        //imshow("filtered", filtered);

        // Return Output image
        Mat dstImg = Mat(warped.size(), CV_8U, Scalar(128)); // Clear the
output image to a default gray.
        dstImg = filtered;
        return dstImg;
    }
    // Return a Null matriz if the face was not detected
    return Mat();

}
```

A.4. Red Neuronal de Convolución

A.4.1. Parámetros Globales

```
#include <math.h>

#pragma once

#define UINT unsigned int

const UINT g_imageSize = 40;
const UINT g_vectorSize = 41;
const UINT g_cOutputSize = 7;
const UINT g_countHessianSample = 100;
const bool g_useDropout = false;
const bool g_useWeightDecay = true;
const double g_regLambda = 0.01;

#define RGB_TO_BGRQUAD(r,g,b) (RGB((b),(g),(r)))

#define GAUSSIAN_FIELD_SIZE (21)

#define RANDOM_PLUS_MINUS_ONE  ( (double)(2.0 * rand())/RAND_MAX - 1.0 )     //in [-1,
#define RANDOM_ZERO_ONE                         ( (double) rand() /
(RAND_MAX + 1) )              //in [0, 1) range
#define RANDOM_BINOM                    ( (int) 2.0 * rand() / (RAND_MAX +
1) )           //in [0, 1) range

const double dmuParameter = 0.10;  //since we divide by this, update can
never be more than 10x current eta
```

```
const double PI = 2.0 * acos(0.0);
```

A.4.2. Clase Principal

```
////////////////////////////////////////////////////////////////////////
////////////
//              Convolutional Neuronal Network
//
////////////////////////////////////////////////////////////////////////
////////////
// Referencias:  Mike O'Neill, Neural Network for Recognition of
Handwritten Digits
//               2006

#include "CNN.h"
#include <stdlib.h>
#include <ctime>
#include <iostream>
#include <string>
#include <fstream>

double ACTFUN(double x) {return 1.7159*tanh(0.66666667*x);}
// derivative of the sigmoid as a function of the sigmoid's output
double DACTFUN(double x) {return 0.66666667/1.7159*(1.7159+(x))*(1.7159-
(x));}

/**
 * @brief Class constructor
 */
ConvNN::ConvNN()
{
    int i;
    m_testTime = false;
    // Number of Layers
    m_numberOfLayers = 6;
    // Initialize vector of Layers
    m_Layer = new Layer[m_numberOfLayers];
    m_Layer[0].previousLayer = NULL;

    // Connect Layers
    for (i=1; i<m_numberOfLayers; i++) m_Layer[i].previousLayer =
&m_Layer[i-1];

    //void Layer::Construct (    type,                   nFeatureMap,
featureSize, kernelSize, stepFactor)
    m_Layer[0].construct       (       Layer::INPUT_LAYER,         1,
41,          0,          0     );
    m_Layer[1].construct       (       Layer::CONVOLUTIONAL,          10,
19,          5,          2     );
    m_Layer[2].construct       (       Layer::CONVOLUTIONAL,          20,
8,          5,          2      );
    m_Layer[3].construct       (       Layer::CONVOLUTIONAL,       10,
4,          2,          2      );
    m_Layer[4].construct       (       Layer::FULLY_CONNECTED,     15,
1,          4,          1      );
    m_Layer[5].construct       (       Layer::FULLY_CONNECTED,     7,
1,          1,          1      );
```

```
}

/**
 * @brief Class destructor
 */
ConvNN::~ConvNN(void)
{
    //saveWeights("weights_updated.txt");
    for(int i=0; i<m_numberOfLayers; i++) m_Layer[i].deleteFeatureMap();
}

/**
 * @brief Load random weigths for test
 */
void ConvNN::loadWeightsRandom()
{
    int i, j, k, m;

    srand((unsigned)time(0));

    for ( i=1; i<m_numberOfLayers; i++ ) {
        for( j=0; j<m_Layer[i].m_numberOfFeatureMaps; j++ ) {
            m_Layer[i].m_FeatureMap[j].bias = 0.05 *
RANDOM_PLUS_MINUS_ONE;

            for(k=0; k<m_Layer[i].previousLayer->m_numberOfFeatureMaps;
k++)
                for(m=0; m < m_Layer[i].m_KernelSize *
m_Layer[i].m_KernelSize; m++)
                    m_Layer[i].m_FeatureMap[j].kernel.at<double>(k,m) =
0.05 * RANDOM_PLUS_MINUS_ONE;
        }
    }
}

/**
 * @brief load weigths from filename
 * @param fileName
 */
void ConvNN::loadWeights(const char *fileName)
{
    int i, j, k, m, n;

    FILE *f;
    if((f = fopen(fileName, "r")) == NULL) return;

    for ( i=1; i<m_numberOfLayers; i++ ) {
        for( j=0; j<m_Layer[i].m_numberOfFeatureMaps; j++ ) {
            fscanf(f, "%lg ", &m_Layer[i].m_FeatureMap[j].bias);

            for(k=0; k<m_Layer[i].previousLayer->m_numberOfFeatureMaps;
k++)
                for(m=0; m < m_Layer[i].m_KernelSize *
m_Layer[i].m_KernelSize; m++)
                    fscanf(f, "%lg ",
&m_Layer[i].m_FeatureMap[j].kernel.at<double>(k,m));
        }
    }
```

```
    fclose(f);
}

/**
 * @brief Save weigths in filename
 * @param fileName
 */
void ConvNN::saveWeights(const char *fileName)
{
    int i, j, k, m;

    FILE *f;
    if((f = fopen(fileName, "w")) == NULL) return;

    for ( i=1; i<m_numberOfLayers; i++ ) {
        for( j=0; j<m_Layer[i].m_numberOfFeatureMaps; j++ ) {
            fprintf(f, "%lg ", m_Layer[i].m_FeatureMap[j].bias);

            for(k=0; k<m_Layer[i].previousLayer->m_numberOfFeatureMaps;
k++)
                for(m=0; m < m_Layer[i].m_KernelSize *
m_Layer[i].m_KernelSize; m++) {
                    fprintf(f, "%lg ",
m_Layer[i].m_FeatureMap[j].kernel.at<double>(k,m));
                }
        }
    }
    fclose(f);
}

/**
 * @brief Forward propagation of the network
 * @param input
 * @param output
 * @return
 */
int ConvNN::calculate(std::vector<double> &input, std::vector<double>
&output, bool testTime)
{
    m_testTime = testTime;
    int i, j;

    //copy input to first layer
    for (i = 0; i < m_Layer[0].m_numberOfFeatureMaps; i++) {
        m_Layer[0].m_FeatureMap[0].xValue = input;
    }

    //forward propagation
    //calculate x_i values of neurons in each layer
    for (i = 1; i < m_numberOfLayers; i++) {
        //Clear previous values
        for (j = 0; j < m_Layer[i].m_numberOfFeatureMaps; j++) {
            m_Layer[i].m_FeatureMap[j].clearXValues();
        }

        //forward propagation from layer[i-1] to layer[i]
        m_Layer[i].calculate();
    }
```

```
    //copy last layer values to output
    for (i = 0; i < m_Layer[m_numberOfLayers-1].m_numberOfFeatureMaps;
i++) {
        output[i] = m_Layer[m_numberOfLayers-
1].m_FeatureMap[i].xValue[0];
    }

    //get index of highest scoring output feature
    j = 0;
    for (i = 1; i < m_Layer[m_numberOfLayers - 1].m_numberOfFeatureMaps;
i++) {
        if (output[i] > output[j]) j = i;
    }
    return j;
}

/**
 * @brief Backpropagation starts here
 * @param targetOutput
 * @param eta
 */
void ConvNN::backPropagate(std::vector<double> &targetOutput, double eta)
{
    int i;

    //derivative of the error in last layer
    //calculated as difference between actual and target output
    // dEn/dxn = x_n - t_n
    for (i = 0; i < m_Layer[m_numberOfLayers - 1].m_numberOfFeatureMaps;
i++) {
        m_Layer[m_numberOfLayers - 1].m_FeatureMap[i].dError_dx[0] =
                m_Layer[m_numberOfLayers - 1].m_FeatureMap[i].xValue[0] -
targetOutput[i];
    }

    // Save MSE for network evaluation
    double mse=0.0;
    for (i = 0; i < m_Layer[m_numberOfLayers-1].m_numberOfFeatureMaps;
i++) {
        mse += m_Layer[m_numberOfLayers-1].m_FeatureMap[i].dError_dx[0] *
m_Layer[m_numberOfLayers-1].m_FeatureMap[i].dError_dx[0];
    }

    //backpropagate through rest of the layers
    for( i = m_numberOfLayers - 1; i > 0; i--) {
        m_Layer[i].backPropagate(1, eta);
    }
}

/**
 * @brief Back propagation of the diagonal hessian
 */
void ConvNN::calculateHessian()
{
    int i, j, k, m;

    //2nd derivative of the error wrt Xn in last layer is always 1
```

```
    //Xn is the output after applying SIGMOID (z_j in Chapter 2)
    for(i=0; i<m_Layer[m_numberOfLayers-1].m_numberOfFeatureMaps; i++) {
        m_Layer[m_numberOfLayers-1].m_FeatureMap[i].dError_dx[0] = 1.0;
    }

    //backpropagate through rest of the layers
    for(i=m_numberOfLayers-1; i>0; i--) {
        m_Layer[i].backPropagate(2, 0);
    }

    //average over the number of samples used
    for(i=1; i<m_numberOfLayers; i++) {
        for(j=0; j<m_Layer[i].m_numberOfFeatureMaps; j++) {
            m_Layer[i].m_FeatureMap[j].diagHessianBias /=
g_countHessianSample;

            for(k=0; k<m_Layer[i].previousLayer->m_numberOfFeatureMaps;
k++) {
                for(int m=0; m < m_Layer[i].m_KernelSize *
m_Layer[i].m_KernelSize; m++) {

m_Layer[i].m_FeatureMap[j].diagHessian.at<double>(k,m) /=
g_countHessianSample;
                }
            }
        }
    }
}

/**
 * @brief Construct layer of given type
 * @param type
 * @param nFeatureMap
 * @param featureSize
 * @param kernelSize
 * @param samplingFactor
 */
void Layer::construct(Layer::LayerType type, int nFeatureMap, int
featureSize, int kernelSize, int samplingFactor)
{
    m_type = type;
    m_numberOfFeatureMaps = nFeatureMap;
    m_FeatureSize = featureSize;
    m_KernelSize = kernelSize;
    m_SamplingFactor = samplingFactor;

    m_FeatureMap = new FeatureMap[m_numberOfFeatureMaps];

    for(int j=0; j<m_numberOfFeatureMaps; j++) {
        m_FeatureMap[j].pLayer = this;
        m_FeatureMap[j].constructLayer();
    }
}

/**
 * @brief Clear featureMap data
 */
void Layer::deleteFeatureMap()
```

```
{
    for(int j=0; j<m_numberOfFeatureMaps; j++)
m_FeatureMap[j].deleteData();
}

/**
 * @brief forward propagation in a layer
 */
void Layer::calculate()                //forward propagation
{
    for (int i = 0; i < m_numberOfFeatureMaps; i++) {
        //initialize feature map to bias
        std::fill(m_FeatureMap[i].xValue.begin(),
m_FeatureMap[i].xValue.end(), m_FeatureMap[i].bias);

        //calculate x value of each feature map using outputs of previous
layer feature maps
        for (int j = 0; j < previousLayer->m_numberOfFeatureMaps; j++) {
            m_FeatureMap[i].calculate(
                        previousLayer->m_FeatureMap[j].xValue,              //output
                        j
                        );
        }

        // Calculate output appling activation function: Apply sigmoid
function to each element in xValue
        std::transform (m_FeatureMap[i].xValue.begin(),
m_FeatureMap[i].xValue.end(), m_FeatureMap[i].xValue.begin(), ACTFUN);
    }
}


/**
 * @brief backpropagation in a layer
 * @param dOrder
 * @param etaLearningRate
 */
void Layer::backPropagate(int dOrder, double etaLearningRate)
{
    std::vector<double> temp;
    //Calculate dEn/dYn = DSIGMOID(Xn)^2 * dEn/dXn
    for (int i=0; i < m_numberOfFeatureMaps; i++) {
        temp.resize(m_FeatureMap[i].xValue.size());
        //temp = DSIGMOID(m_FeatureMap[i].xValue)
        std::transform (m_FeatureMap[i].xValue.begin(),
m_FeatureMap[i].xValue.end(), temp.begin(), DACTFUN);
        //temp = temp^2
        if (dOrder == 2) std::transform(temp.begin(), temp.end(),
temp.begin(), temp.begin(), std::multiplies<float>());
        // dError_dx = dError_dx*temp
        std::transform(m_FeatureMap[i].dError_dx.begin(),
m_FeatureMap[i].dError_dx.end(), temp.begin(),
                       m_FeatureMap[i].dError_dx.begin(),
std::multiplies<float>());
    }

    //clear dError wrt weights
```

```
    for (int i=0; i < m_numberOfFeatureMaps; i++)
m_FeatureMap[i].clear_dE_dwt();

    //clear dError wrt Xn in previous layer.
    //This is input to the previous layer for backpropagation
    for (int i=0; i < previousLayer->m_numberOfFeatureMaps; i++)
        previousLayer->m_FeatureMap[i].clear_dE_dx();

    //Backpropagate
    for (int i = 0; i < m_numberOfFeatureMaps; i++) {
        //dEn/dwn = sum(xi*dEn/dYn) for the bias
        for(int j = 0; j < m_FeatureSize * m_FeatureSize; j++)
            m_FeatureMap[i].dErr_dwtb += m_FeatureMap[i].dError_dx[j];

        //calculate effect of this feature map on each feature map in the
previous layer
        for(int j=0; j<previousLayer->m_numberOfFeatureMaps; j++) {
            // dE_{n-1}/dX_{n-1} = Sum(w*dEn/dyn) and dEn/dwn =
sum(xi*dEn/dYn) for other weigths
            m_FeatureMap[i].backPropagate(
                        previousLayer->m_FeatureMap[j].xValue,              //input f
                        j,
                        previousLayer->m_FeatureMap[j].dError_dx,           //dErr_wr
                        dOrder
                        );
        }
    }

    //update weights (for backporpagation) or diagonal hessian (for 2nd
order backpropagation)
    double epsilon, divisor;

    for (int i=0; i<m_numberOfFeatureMaps; i++) {
        if (dOrder == 1) {
            divisor = std::max(0.0, m_FeatureMap[i].diagHessianBias) +
dmuParameter;
            epsilon = etaLearningRate / divisor;
            if (g_useWeightDecay) {
                m_FeatureMap[i].bias -= (epsilon *
m_FeatureMap[i].dErr_dwtb+epsilon * g_regLambda * m_FeatureMap[i].bias);
            } else {
                m_FeatureMap[i].bias -= epsilon *
m_FeatureMap[i].dErr_dwtb;
            }
        }
        else {
            m_FeatureMap[i].diagHessianBias += m_FeatureMap[i].dErr_dwtb;
        }

        for (int j=0; j<previousLayer->m_numberOfFeatureMaps; j++) {
            for (int k=0; k < m_KernelSize * m_KernelSize; k++) {
                if(dOrder == 1) {
                    divisor = std::max(0.0,
m_FeatureMap[i].diagHessian.at<double>(j,k)) + dmuParameter;
                    epsilon = etaLearningRate / divisor;
                    if (g_useWeightDecay) {
```

```
                        double delta = epsilon *
(m_FeatureMap[i].dErr_dwt.at<double>(j,k) + g_regLambda
*m_FeatureMap[i].kernel.at<double>(j,k));
                        m_FeatureMap[i].kernel.at<double>(j,k) -= delta;
                    } else {
                        m_FeatureMap[i].kernel.at<double>(j,k) -= epsilon
* m_FeatureMap[i].dErr_dwt.at<double>(j,k);
                    }
                }
                else {
                    m_FeatureMap[i].diagHessian.at<double>(j,k) +=
m_FeatureMap[i].dErr_dwt.at<double>(j,k);
                }
            }
        }
    }
}

/**
 * @brief Initialize FeatureMaps shared data
 */
void FeatureMap::constructLayer( )
{
    m_testTime = false;
    if (pLayer->m_type == Layer::INPUT_LAYER) m_nFeatureMapPrev = 0;
    else m_nFeatureMapPrev = pLayer->previousLayer-
>m_numberOfFeatureMaps;

    int FeatureSize = pLayer->m_FeatureSize;
    int KernelSize  = pLayer->m_KernelSize;

    //neuron values
    xValue.resize(FeatureSize * FeatureSize);

    //error in neuron values
    dError_dx.resize(FeatureSize * FeatureSize);

    //shared weights kernel
    kernel = Mat::zeros(m_nFeatureMapPrev, KernelSize * KernelSize,
CV_64F);
    for (int i = 0; i < m_nFeatureMapPrev; i++) {
        //initialize
        bias = 0.05 * RANDOM_PLUS_MINUS_ONE;
        for (int j = 0; j < KernelSize * KernelSize; j++)
kernel.at<double>(i,j) = 0.05 * RANDOM_PLUS_MINUS_ONE;
    }

    //diagHessian
    diagHessian = Mat::zeros(m_nFeatureMapPrev, KernelSize * KernelSize,
CV_64F);

    //derivative of error wrt kernel weights
    dErr_dwt = Mat::zeros(m_nFeatureMapPrev, KernelSize * KernelSize,
CV_64F);
}

/**
 * @brief clear FeatureMap data
```

```
 */
void FeatureMap::deleteData()
{
    xValue.clear();
    dError_dx.clear();
    kernel.release();
    diagHessian.release();
    dErr_dwt.release();
}

/**
 * @brief Clear outputs
 */
void FeatureMap::clearXValues()
{
    std::fill(xValue.begin(), xValue.end(), 0.0);
}

/**
 * @brief Clear error differentials (dE/dx)
 */
void FeatureMap::clear_dE_dx()
{
    std::fill(dError_dx.begin(), dError_dx.end(), 0.0);
}

/**
 * @brief Clear Diagonal Hessian data
 */
void FeatureMap::clearDiagHessian()
{
    diagHessianBias = 0;
    diagHessian = Mat::zeros(diagHessian.size(), diagHessian.type());
}

/**
 * @brief clear error differentials (dE/dwt)
 */
void FeatureMap::clear_dE_dwt()
{
    dErr_dwt = Mat::zeros(dErr_dwt.size(), dErr_dwt.type());
    dErr_dwtb = 0;
}


/**
 * @brief Feature Map forward propagation
 * @param xValuePrevFeatureMap previous layer feature map output
 * @param idxPrevFeatureMap index of feature map in previous layer
 * Calculate c_n = Sum(x_{n-1}*w_n)
 * c_n are convolutions y_n is sum(c_n)
 */
void FeatureMap::calculate(std::vector<double> &xValuePrevFeatureMap, int
idxPrevFeatureMap, bool testTime)
{
    m_testTime = testTime;
    int numberOfInputs = pLayer->previousLayer->m_FeatureSize; //feature
size in previous layer
```

```
    int kernelSize = pLayer->m_KernelSize;
    int stepSize = pLayer->m_SamplingFactor;

    int k = 0;
    for(int row0 = 0; row0 <= numberOfInputs - kernelSize; row0 +=
stepSize)
        for(int col0 = 0; col0 <= numberOfInputs - kernelSize; col0 +=
stepSize) {
            xValue[k++] += convolute(xValuePrevFeatureMap,
numberOfInputs, row0, col0, kernel, idxPrevFeatureMap, kernelSize);
        }

}

/**
 * @brief Convolute a patch of size
 * @param input
 * @param size
 * @param r0
 * @param c0
 * @param weight
 * @param idxPrevFeatureMap
 * @param kernelSize
 * @return
 */
double FeatureMap::convolute(std::vector<double> &input, int size, int
r0, int c0, Mat &weight, int idxPrevFeatureMap, int kernelSize)
{
    int i, j, k = 0;
    double summ = 0;

    for (i = r0; i < r0 + kernelSize; i++) {
        for(j = c0; j < c0 + kernelSize; j++) {
            if (g_useDropout) {
                if (m_testTime) {
                    summ += 0.5 * input[i * size + j] *
weight.at<double>(idxPrevFeatureMap,k++);
                } else {
                    summ += RANDOM_BINOM * input[i * size + j] *
weight.at<double>(idxPrevFeatureMap,k++);
                }
            } else {
                summ += input[i * size + j] *
weight.at<double>(idxPrevFeatureMap,k++);
            }
        }
    }
    return summ;
}

/**
 * @brief backPropagate FeatureMap
 * @param xValueFeatureMapPrev feature map outputs in previous layer
 * @param idxFeatureMapPrev index of feature map in previous layer
 * @param dError_dxFeatureMapPrev dError_dx neuron values in the FM in
prev layer
 * @param dOrder order of the differential
 * Calculates dE_{n-1}/dx_{n-1} = sum(w*dE_n/dx_n)
```

```
 */
void FeatureMap::backPropagate(vector<double> &xValueFeatureMapPrev, int
idxFeatureMapPrev,
                               vector<double> &dError_dxFeatureMapPrev,
int dOrder )
{
    int inputSize = pLayer->previousLayer->m_FeatureSize;    //size of FM
in previous layer
    int kernelSize = pLayer->m_KernelSize;
        //kernel size
    int stepSize = pLayer->m_SamplingFactor;          //subsampling
factor

    int row0, col0, k;

    k = 0;
    for (row0 = 0; row0 <= inputSize - kernelSize; row0 += stepSize) {
        for (col0 = 0; col0 <= inputSize - kernelSize; col0 += stepSize)
{
            for (int i = 0; i < kernelSize; i++) {
                for (int j = 0; j < kernelSize; j++) {

                    //Calculates dE_{n-1}/dx_{n-1} = sum(w*dE_n/dx_n)
                    double weigth_ij =
kernel.at<double>(idxFeatureMapPrev, i * kernelSize + j);
                    if(dOrder == 1)
                        dError_dxFeatureMapPrev[(row0 + i) * inputSize +
(j + col0)] += dError_dx[k] * weigth_ij;
                    else
                        dError_dxFeatureMapPrev[(row0 + i) * inputSize +
(j + col0)] += dError_dx[k] * weigth_ij * weigth_ij;

                    //get dError wrt kernel wights
                    //dEn/dwn = sum(dEn/dYn*xi)
                    double x_ij = xValueFeatureMapPrev[(row0 + i) *
inputSize + (j + col0)];
                    if(dOrder == 1)
                        dErr_dwt.at<double>(idxFeatureMapPrev, i *
kernelSize + j) += dError_dx[k] * x_ij;
                    else
                        dErr_dwt.at<double>(idxFeatureMapPrev, i *
kernelSize + j) += dError_dx[k] * x_ij * x_ij;
                }
            }
            k++;
        }
    }
}
```

A.4.3. Estimación y Entrenamiento

```
/**
 * @brief makeTestAndTrainSets
 */
void makeTestAndTrainSets()
{
```

```
    //clean variables
    test_data.clear();
    test_labels.clear();
    train_data.clear();
    train_labels.clear();

    vector<Mat> procImages;
    vector<int> procLabels;
    vector<int> procPLabels;

    // Read stored preprocessed images
    FileStorage fs;
    cout << "Leyendo archivo de imagenes preprocesadas: " << filename <<
endl;
    fs.open(filename, FileStorage::READ);

    if (!fs.isOpened()) {
        cerr << "Error al abrir " << filename << endl;
        return;
    }

    fs["Images"] >> procImages;
    fs["Labels"] >> procLabels;
    fs["Profile"] >> procPLabels;
    cout << "Datos cargados correctamente..." << endl;

    //pick a random test set of test size.
    //g_testSetSize;
    vector<int> indexes(procImages.size());
    int i;
    for(i=0; i < indexes.size(); i++) indexes[i] = i;
    randomizeIndexes(indexes);

    //make sets
    for (int j = 0;j < procImages.size();j++) {
        vector<int>::iterator it;
        it = std::find(indexes.begin(),indexes.end(),j);
        auto pos = std::distance(indexes.begin(), it);
        if (pos < g_testSetSize) {
            test_data.push_back(procImages[j]);
            test_labels.push_back(procPLabels[j]);
        } else{
            train_data.push_back(procImages[j]);
            train_labels.push_back(procPLabels[j]);
        }
    }
    cout << "Tamaño del conjunto de prueba: " << test_data.size() << endl;
    cout << "Tamaño del conjunto de entrenamiento: " << train_data.size() <<
endl;
}

/**
 * Controls backpropagation of the convolutional neuronal network
 **/
void trainNetwork(int realzton)
{
    cout << "================================================" << endl;
    cout << "Realización " << realzton << endl;
```

```
    cout << "================================================" << endl;
    //file name
    string file_name = real_root + to_string(realzton) + ".txt";
    string weigth_name = weigthFileName + to_string(realzton)+ ".txt";

    //vectors to save data
    vector<double> timeForEpoch;
    vector<double> trainError;
    vector<double> testError;
    vector<double> testTime;
    vector<int> oneStepError;

    //initialize variables
    auto train_start = std::chrono::high_resolution_clock::now();
    bool m_statusTraining = true;
    int errorCount = 0;
    int oneStepCount = 0;
    int m_iNumImage = 0;
    int m_cLabel = -1;
    vector<double> desiredOutputVector(g_cOutputSize);
    const char *outFileName;
    outFileName = weigth_name.c_str();

    //indices of training samples, to be used for random picking
    vector<int> indexes(g_countTrainingSample);
    int i;
    for(i=0; i < g_countTrainingSample; i++) indexes[i] = i;

    int j = 0;
    double epochs = 1;
    while(m_statusTraining)
    {
        cout << "Epoch: " << epochs << endl;
        auto start = std::chrono::high_resolution_clock::now();
        calculateHessian(train_data, train_labels);
        randomizeIndexes(indexes);
        errorCount = 0;
        int i;
        oneStepCount = 0;
        for(i=0; i < g_countTrainingSample; i++) {
            m_iNumImage = indexes[i];
            int  m_iOutput  =  calculate(m_iNumImage,  m_cLabel,  train_data,
train_labels); //forward propagation

            if (m_iOutput != m_cLabel) {
                errorCount++;
                if (abs(m_iOutput - m_cLabel) == 1) oneStepCount++;
            }

            //false recognition count
            //cout << "Image: " << i << " Real: " << m_cLabel << " pred: " <<
m_iOutput << endl;
            //cout  << "Error  Count:  "  <<  errorCount  <<  "  of:  "  <<  i  <<
"images" << endl;

            //backward propagation
            std::fill(desiredOutputVector.begin(), desiredOutputVector.end(),
-1);
```

```
            desiredOutputVector[m_cLabel] = 1;
            g_cnn.backPropagate(desiredOutputVector, g_etaLearningRate);

            if(!m_statusTraining) break;
        }
        //recognition history (count of false recognitions in each epoch)
        auto end = std::chrono::high_resolution_clock::now();
        auto                          elapsed                          =
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
        cout << "Tiempo empleado: " << elapsed << " milliseconds" << endl;
        cout  <<  "Error  Entrenamiento:  "  <<  errorCount  <<  "  :  "  <<
errorCount/double(g_countTrainingSample) << "%" << endl;
        //store data
        timeForEpoch.push_back(elapsed);
        trainError.push_back(errorCount/double(g_countTrainingSample));
        oneStepError.push_back(oneStepCount);

        //adjust learning rate after g_decayAfterEpochs epochs
        if ((fmod(epochs, g_decayAfterEpochs) == 0.0) && (epochs != 0)) {
            g_etaLearningRate *= g_decayingRate;
        }

        //save weights
        if(outFileName != NULL) g_cnn.saveWeights(outFileName);
        testNetwork(testError, testTime);

        //Save in file
        FileStorage fs(file_name, FileStorage::WRITE);
        fs << "TrainTime" << timeForEpoch;
        fs << "TrainError" << trainError;
        fs << "TestTime" << testTime;
        fs << "TestError" << testError;
        fs << "OneStepCount" << oneStepError;
        fs.release();

        //set a target for training completion
        if(errorCount        <        int(g_countTrainingSample*g_errorRate))
m_statusTraining = false;
        if (epochs >= g_epochs_limit) m_statusTraining = false;
        epochs += 1;

    }
    cout  <<  "Entrenamiento  Finalizado  pesos  almacenados  en  archivo  "  <<
weigthFileName <<"..." << endl;
    auto train_end = std::chrono::high_resolution_clock::now();
    auto                          train_elapsed                          =
std::chrono::duration_cast<std::chrono::milliseconds>(train_end            -
train_start).count();
    cout << "Tiempo total empleado: " << train_elapsed << " milliseconds" <<
endl;

    //Save in file
    FileStorage fs(file_name, FileStorage::WRITE);
    fs << "TrainTime" << timeForEpoch;
    fs << "TrainError" << trainError;
    fs << "TestTime" << testTime;
    fs << "TestError" << testError;
```

```
    fs << "TotalTime" << (double)train_elapsed;
    fs.release();
}

/**
 * Use test data set to determine the performance of the network
 **/
void testNetwork(vector<double> &testError, vector<double> &testTime)
{
    int countError = 0;
    int m_iNumImage = 0;
    int i;
    int realLabel = -1;
    const char *outFileName;
    outFileName = weigthFileName.c_str();
    //load weights
    if(outFileName != NULL) g_cnn.loadWeights(outFileName);
    cout << "Archivo de pesos: " << weigthFileName << " cargado
exitosamente..." << endl;
    auto start = std::chrono::high_resolution_clock::now();
    for(i=0; i < g_countTestingSample; i++) {
        m_iNumImage = i;
        int predictedLabel = calculate(m_iNumImage, realLabel, test_data,
test_labels, true); //forward propagation
        //cout << "Imagen: " << i << " etiqueta: " << realLabel << " se
identifico como: " << predictedLabel << endl;
        //update false detections so far
        if (predictedLabel != (int) realLabel) {
            ++countError;
            //cout << " Error count: " << countError << endl;
        }
    }
    cout << "Tasa de Error en Test: " <<
(countError/(double)g_countTestingSample) << endl;
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end
- start).count();
    cout << "Tiempo empleado en datos de prueba: " << elapsed << "
milliseconds" << endl;
    //Save Data
    testError.push_back(countError/(double)g_countTestingSample);
    testTime.push_back(elapsed);
}

/**
 * Randomize vector of intengers
 **/
void randomizeIndexes(vector<int> &idx)
{
    int i, j;

    srand((unsigned)time(0));

    for (i = 0; i < g_countTrainingSample; i++) {
        j = int((double)rand()/(double)RAND_MAX*double(g_countTrainingSample-
1));
        int temp = idx[i];
        idx[i] = idx[j];
```

```
        idx[j] = temp;
    }
}

/**
 * Forward propagation of the diagonal hessian
 **/
void calculateHessian(vector<Mat> &data, vector<int> &labels)
{
    int i;

    //clear all
    for(i = 0; i < g_cnn.m_numberOfLayers; i++) g_cnn.m_Layer[i].clearAll();

    //Use random g_sampleCountForHessian samples for calculating diagHessian
    //for random sequence to change every time, it's seed with current time
    srand((unsigned)time(0));

    for(i = 0; i < g_countHessianSample; i++) {
        int imageIndex = (int) floor((double)rand() / (double) RAND_MAX *
(g_countTrainingSample - 1))
                + (int) ((double)rand() / RAND_MAX);

        //forward propagation
        int currentLabel = -1;
        calculate(imageIndex, currentLabel, data, labels);

        //backpropagation of 2nd derivative
        g_cnn.calculateHessian();
    }
}

/**
 * Forward propagation of the data on the network
 **/
int calculate(int index, int &currentLabel, vector<Mat> &data, vector<int>
&labels, bool testTime)
{
    //open the data set and read character and label
    Mat preprocessedFace = data[index];
    currentLabel = labels[index];

    // Get size of preprocessed Images
    int sz = preprocessedFace.cols*preprocessedFace.rows;

    //Reshape image in a column vector
    Mat ptvec = preprocessedFace.reshape(1, sz);
    cv::Mat first_col(ptvec.col(0));
    std::vector<Scalar> v(first_col.begin<uchar>(), first_col.end<uchar>());

    // Prepare temporal vectors
    vector<double> inputVector(g_vectorSize*g_vectorSize, 0.0);
    vector<double> outputVector(g_cOutputSize*g_cOutputSize, 0.0);

    //copy gray scale image to a double input vector in -1 to 1 range
    // one is white, -one is black
    int ii, jj;
    for (ii = 0; ii < inputVector.size(); ++ii) inputVector[ii] = 1.0;
```

```
    for (ii = 0; ii < g_imageSize; ++ii) {
        for (jj=0; jj < g_imageSize; ++jj) {
            int idxVector = 1 + jj + g_vectorSize * (1 + ii);
            int idxImage = jj + g_imageSize * ii;
            inputVector[idxVector] = double(255 - v[idxImage].val[0])/128.0 -
1.0;
        }
    }
    int i_output = -1;
    //call forward propagation function of CNN
    if (testTime) {
        i_output = g_cnn.calculate(inputVector, outputVector, true);
       } else {
        i_output = g_cnn.calculate(inputVector, outputVector);
    }
    return i_output;
}
```

A.5. Reconocimiento de Rostros

A.5.1. Reconocimiento por rostros propios

```
/*****************************************************************************
*****
*   Face Recognition using Eigenfaces or Fisherfaces
*****************************************************************************
*****
*   Referencias: Shervin Emami, 5th Dec 2012
*                http://www.shervinemami.info/openCV.html
*   Requires OpenCV v2.4.1 or later
*****************************************************************************
****/

#include "recognition.h"     // Train the face recognition system and
recognize a person from an image.

/**
 * @brief Start training from the collected faces
 * The face recognition algorithm can be one of these and perhaps more,
depending on your version of OpenCV, which must be atleast v2.4.1:
 * "FaceRecognizer.Eigenfaces":  Eigenfaces, also referred to as PCA
(Turk and Pentland, 1991).
 * "FaceRecognizer.Fisherfaces": Fisherfaces, also referred to as LDA
(Belhumeur et al, 1997).
 * "FaceRecognizer.LBPH":        Local Binary Pattern Histograms (Ahonen
et al, 2006).
 * @param preprocessedFaces
 * @param faceLabels
 * @param facerecAlgorithm
 * @return
 */
Ptr<FaceRecognizer> learnCollectedFaces(const vector<Mat>
preprocessedFaces, const vector<int> faceLabels, const string
facerecAlgorithm)
{
    Ptr<FaceRecognizer> model;
```

```
    cout << "Learning the collected faces using the [" <<
facerecAlgorithm << "] algorithm ..." << endl;

    // Make sure the "contrib" module is dynamically loaded at runtime.
    // Requires OpenCV v2.4.1 or later (from June 2012), otherwise the
FaceRecognizer will not compile or run!
    bool haveContribModule = initModule_contrib();
    if (!haveContribModule) {
        cerr << "ERROR: The 'contrib' module is needed for FaceRecognizer
but has not been loaded into OpenCV!" << endl;
        exit(1);
    }

    // Use the new FaceRecognizer class in OpenCV's "contrib" module:
    // Requires OpenCV v2.4.1 or later (from June 2012), otherwise the
FaceRecognizer will not compile or run!
    model = Algorithm::create<FaceRecognizer>(facerecAlgorithm);
    if (model.empty()) {
        cerr << "ERROR: The FaceRecognizer algorithm [" <<
facerecAlgorithm << "] is not available in your version of OpenCV. Please
update to OpenCV v2.4.1 or newer." << endl;
        exit(1);
    }

    // Do the actual training from the collected faces. Might take
several seconds or minutes depending on input!
    model->train(preprocessedFaces, faceLabels);

    return model;
}

/**
 * @brief loadSavedModel
 * @param filename
 * @param facerecAlgorithm
 * @return
 */
Ptr<FaceRecognizer> loadSavedModel (const string filename, const string
facerecAlgorithm) {
    Ptr<FaceRecognizer> model;
    model = Algorithm::create<FaceRecognizer>(facerecAlgorithm);
    model->load(filename);
    return model;
}

/**
 * @brief Convert the matrix row or column (float matrix) to a
rectangular 8-bit image that can be displayed or saved.
 * Scales the values to be between 0 to 255.
 * @param matrixRow
 * @param height
 * @return
 */
Mat getImageFrom1DFloatMat(const Mat matrixRow, int height)
{
    // Make it a rectangular shaped image instead of a single row.
    Mat rectangularMat = matrixRow.reshape(1, height);
```

```
    // Scale the values to be between 0 to 255 and store them as a
regular 8-bit uchar image.
    Mat dst;
    normalize(rectangularMat, dst, 0, 255, NORM_MINMAX, CV_8UC1);
    return dst;
}

/**
 * @brief Generate an approximately reconstructed face by back-projecting
the eigenvectors & eigenvalues of the given (preprocessed) face.
 * @param model
 * @param preprocessedFace
 * @return
 */
Mat reconstructFace(const Ptr<FaceRecognizer> model, const Mat
preprocessedFace)
{
    // Since we can only reconstruct the face for some types of
FaceRecognizer models (ie: Eigenfaces or Fisherfaces),
    // we should surround the OpenCV calls by a try/catch block so we
don't crash for other models.
    try {

        // Get some required data from the FaceRecognizer model.
        Mat eigenvectors = model->get<Mat>("eigenvectors");
        Mat averageFaceRow = model->get<Mat>("mean");

        int faceHeight = preprocessedFace.rows;

        // Project the input image onto the PCA subspace.
        Mat projection = subspaceProject(eigenvectors, averageFaceRow,
preprocessedFace.reshape(1,1));
        //printMatInfo(projection, "projection");

        // Generate the reconstructed face back from the PCA subspace.
        Mat reconstructionRow = subspaceReconstruct(eigenvectors,
averageFaceRow, projection);
        //printMatInfo(reconstructionRow, "reconstructionRow");

        // Convert the float row matrix to a regular 8-bit image. Note
that we
        // shouldn't use "getImageFrom1DFloatMat()" because we don't want
to normalize
        // the data since it is already at the perfect scale.

        // Make it a rectangular shaped image instead of a single row.
        Mat reconstructionMat = reconstructionRow.reshape(1, faceHeight);
        // Convert the floating-point pixels to regular 8-bit uchar
pixels.
        Mat reconstructedFace = Mat(reconstructionMat.size(), CV_8U);
        reconstructionMat.convertTo(reconstructedFace, CV_8U, 1, 0);
        //printMatInfo(reconstructedFace, "reconstructedFace");

        return reconstructedFace;

    } catch (cv::Exception e) {
        //cout << "WARNING: Missing FaceRecognizer properties." << endl;
        return Mat();
```

```
    }
}

/**
 * @brief Compare two images by getting the L2 error (square-root of sum
of squared error).
 * @param A
 * @param B
 * @return
 */
double getSimilarity(const Mat A, const Mat B)
{
    if (A.rows > 0 && A.rows == B.rows && A.cols > 0 && A.cols == B.cols)
{
        // Calculate the L2 relative error between the 2 images.
        double errorL2 = norm(A, B, CV_L2);
        // Convert to a reasonable scale, since L2 error is summed across
all pixels of the image.
        double similarity = errorL2 / (double)(A.rows * A.cols);
        return similarity;
    }
    else {
        //cout << "WARNING: Images have a different size in
'getSimilarity()'." << endl;
        return 100000000.0;  // Return a bad value
    }
}



/**
 * @brief main function
 * @param argc
 * @param argv
 * @return
 */
int main(int argc, char *argv[])
{
    // Read Test and Train Images from files
    //preprocessData();

    //Init vectors used to save data
    vector<double> trainTime;
    vector<double> testTime;
    vector<double> error_p0;
    vector<double> error_p1;
    vector<double> error_p2;
    vector<double> error_p3;
    vector<double> error_p4;
    vector<double> error_p5;
    vector<double> error_p6;
    vector<double> error_t;
    vector<vector<int> > poses;

    //Initit evaluation
    int realizations = 30;
    for (int idx = 1; idx <= realizations; idx++){
```

```
        cout << "==================================================" <<
endl;
        cout << "Realización " << idx << endl;
        cout << "==================================================" <<
endl;

        //Read Train and Test Set
        makeTestAndTrainSets();

        // Init recognizer
        Ptr<FaceRecognizer> model;

        // Start training from the collected faces using Eigenfaces or a
similar algorithm.
        cout << "Entrenando reconocedor de rostros ..." << endl;
        //cout << pose_labels[0] << endl;
        auto start = std::chrono::high_resolution_clock::now();
        model = learnCollectedFaces(train_data, train_labels,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model->save(modelname);

        //load saved model
        //model = createEigenFaceRecognizer();
        //model->load(modelname);

        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
        cout << "Tiempo empleado en entrenar modelo: " << elapsed << "
milliseconds" << endl;
        trainTime.push_back(elapsed);
        // The following line predicts the label of a given
        // test image:
        cout << "Evaluando el modelo..." << endl;
        start = std::chrono::high_resolution_clock::now();
        int correctPredictions = 0;
        int pose0 = 0; int pose1 = 0; int pose2 = 0; int pose3 = 0;
        int pose4 = 0; int pose5 = 0; int pose6 = 0;
        int poseC0 = 0; int poseC1 = 0; int poseC2 = 0; int poseC3 = 0;
        int poseC4 = 0; int poseC5 = 0; int poseC6 = 0;
        poses.push_back(pose_labels);

        for (int j=0; j < test_data.size();j++) {
            // Run the face recognition system
            int identity = -1;
            Mat testSample = test_data[j];
            int testLabel = test_labels[j];
            int poseLabel = pose_labels[j];
            if (poseLabel == 0) pose0++;
            if (poseLabel == 1) pose1++;
            if (poseLabel == 2) pose2++;
            if (poseLabel == 3) pose3++;
            if (poseLabel == 4) pose4++;
            if (poseLabel == 5) pose5++;
            if (poseLabel == 6) pose6++;
```

```
            // Generate a face approximation by back-projecting the
eigenvectors & eigenvalues.
            Mat reconstructedFace;
            reconstructedFace = reconstructFace(model, testSample);

            // Verify whether the reconstructed face looks like the
preprocessed face, otherwise it is probably an unknown person.
            double similarity = getSimilarity(testSample,
reconstructedFace);
            //cout << "Semejanza: " << similarity << ";" << model-
>predict(testSample) <<  endl;

            if (similarity < UNKNOWN_PERSON_THRESHOLD) {
                // Identify who the person is in the preprocessed face
image.
                identity = model->predict(testSample);
            }

            //string result_message = format("Etiqueta predecida = %d /
Etiqueta Real = %d.", identity, testLabel);
            //cout << result_message << endl;
            if (identity == testLabel) {
                correctPredictions++;
                if (poseLabel == 0) poseC0++;
                if (poseLabel == 1) poseC1++;
                if (poseLabel == 2) poseC2++;
                if (poseLabel == 3) poseC3++;
                if (poseLabel == 4) poseC4++;
                if (poseLabel == 5) poseC5++;
                if (poseLabel == 6) poseC6++;

            }
        }
        double tasaAciertos = (double)correctPredictions /
(double)test_data.size();
        string result_message = format("Predicciones correctas = %d, tasa
= %f.", correctPredictions, tasaAciertos);
        string pose_0 = format("Correctas %f. de %d", (double)poseC0 /
(double)pose0, pose0);
        string pose_1 = format("Correctas %f. de %d",(double)poseC1 /
(double)pose1, pose1);
        string pose_2 = format("Correctas %f. de %d",(double)poseC2 /
(double)pose2, pose2);
        string pose_3 = format("Correctas %f. de %d", (double)poseC3 /
(double)pose3, pose3);
        string pose_4 = format("Correctas %f. de %d", (double)poseC4 /
(double)pose4, pose4);
        string pose_5 = format("Correctas %f. de %d", (double)poseC5 /
(double)pose5, pose5);
        string pose_6 = format("Correctas %f. de %d", (double)poseC6 /
(double)pose6, pose6);
        cout << result_message << endl;
        cout << pose_0 << endl;
        cout << pose_1 << endl;
        cout << pose_2 << endl;
        cout << pose_3 << endl;
        cout << pose_4 << endl;
        cout << pose_5 << endl;
```

```
        cout << pose_6 << endl;

        end = std::chrono::high_resolution_clock::now();
        elapsed =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
        cout << "Tiempo empleado en probar modelo: " << elapsed << "
milliseconds" << endl;
        //Update data
        testTime.push_back(elapsed);
        error_p0.push_back((double)poseC0 / (double)pose0);
        error_p1.push_back((double)poseC1 / (double)pose1);
        error_p2.push_back((double)poseC2 / (double)pose2);
        error_p3.push_back((double)poseC3 / (double)pose3);
        error_p4.push_back((double)poseC4 / (double)pose4);
        error_p5.push_back((double)poseC5 / (double)pose5);
        error_p6.push_back((double)poseC6 / (double)pose6);
        error_t.push_back(tasaAciertos);
        //Save data in file
        FileStorage fs(dataname, FileStorage::WRITE);
        fs << "TrainTime" << trainTime;
        fs << "TestTime" << testTime;
        fs << "ErrorP0" << error_p0;
        fs << "ErrorP1" << error_p1;
        fs << "ErrorP2" << error_p2;
        fs << "ErrorP3" << error_p3;
        fs << "ErrorP4" << error_p4;
        fs << "ErrorP5" << error_p5;
        fs << "ErrorP6" << error_p6;
        fs << "ErrorT" << error_t;
        fs << "Poses" << poses;
        fs.release();
    }

    return 0;
}
```

A.5.2. Reconocimiento con estimación previa de pose

```
/**
 * @brief main function
 * @param argc
 * @param argv
 * @return
 */
int main(int argc, char *argv[])
{
    // Read Test and Train Images from files
    //preprocessData();
    vector<vector<double> > train_errors;
    vector<double> train_error;
    vector<double> train_time;
    vector<double> test_time;
    for (int i = 0; i<7;i++){
        vector<double> pose;
        train_errors.push_back(pose);
```

```
    }
    //Initit evaluation
    int realizations = 30;
    for (int idx = 1; idx <= realizations; idx++){
        cout << "=====================================================" <<
endl;
        cout << "Realización " << idx << endl;
        cout << "=====================================================" <<
endl;

        // Read preprocessed images
        makeTestAndTrainSets();

        auto start = std::chrono::high_resolution_clock::now();
        // Init recognizers for each pose
        Ptr<FaceRecognizer> model_lp;
        Ptr<FaceRecognizer> model_lh;
        Ptr<FaceRecognizer> model_lq;
        Ptr<FaceRecognizer> model_f;
        Ptr<FaceRecognizer> model_rq;
        Ptr<FaceRecognizer> model_rh;
        Ptr<FaceRecognizer> model_rp;


        // Start training from the collected faces using Eigenfaces or a
similar algorithm.

        cout << "Entrenando reconocedor de rostros ..." << endl;
        cout << "Perfil Izquierdo..." << endl;
        model_lp = learnCollectedFaces(train_data_lp, train_labels_lp,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_lp->save(model_lp_name);
        cout << "Medio Perfil Izquierdo..." << endl;
        model_lh = learnCollectedFaces(train_data_lh, train_labels_lh,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_lh->save(model_lh_name);
        cout << "Cuarto Perfil Izquierdo..." << endl;
        model_lq = learnCollectedFaces(train_data_lq, train_labels_lq,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_lq->save(model_lq_name);
        cout << "Frontal..." << endl;
        model_f = learnCollectedFaces(train_data_f, train_labels_f,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_f->save(model_f_name);
        cout << "Cuarto Perfil Derecho..." << endl;
        model_rq = learnCollectedFaces(train_data_rq, train_labels_rq,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_rq->save(model_rq_name);
        cout << "Medio Perfil Derecho..." << endl;
        model_rh = learnCollectedFaces(train_data_rh, train_labels_rh,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_rh->save(model_rh_name);
```

```
        cout << "Perfil derecho..." << endl;
        model_rp = learnCollectedFaces(train_data_rp, train_labels_rp,
facerecAlgorithm);
        // save the model to eigenfaces.yaml
        model_rp->save(model_rp_name);

        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
        cout << "Tiempo empleado en entrenar el modelo: " << elapsed << "
milliseconds" << endl;

        train_time.push_back(elapsed);

        //load saved model
        model_lp = loadSavedModel(model_lp_name, facerecAlgorithm);
        model_lh = loadSavedModel(model_lh_name, facerecAlgorithm);
        model_lq = loadSavedModel(model_lq_name, facerecAlgorithm);
        model_f = loadSavedModel(model_f_name, facerecAlgorithm);
        model_rq = loadSavedModel(model_rq_name, facerecAlgorithm);
        model_rh = loadSavedModel(model_rh_name, facerecAlgorithm);
        model_rp = loadSavedModel(model_rp_name, facerecAlgorithm);

        // Load the network
        const char *outFileName;
        outFileName = cnnw_name.c_str();
        g_cnn.loadWeights(outFileName);

        cout << "Evaluando el modelo..." << endl;
        start = std::chrono::high_resolution_clock::now();
        int correctPredictions = 0;
        vector<int> correct_p;
        vector<int> total_p;
        for (int i = 0; i < 7; i++) {
            correct_p.push_back(0);
            total_p.push_back(0);
        }
        for (int j=0; j < test_data.size();j++) {
            // Run the face recognition system
            int identity = -1;
            Mat testSample = test_data[j];
            int testLabel = test_labels[j];

            Ptr<FaceRecognizer> model;
            // Get model
            int realLabel;
            int predictedLabel = calculate(j, realLabel, test_data,
test_plabels); //forward propagation
            total_p[predictedLabel] = total_p[predictedLabel] + 1;
            switch(predictedLabel) {
            case 0:
                model = model_lp;
                break;
            case 1:
                model = model_lh;
                break;
            case 2:
```

```
                model = model_lq;
                break;
            case 3:
                model = model_f;
                break;
            case 4:
                model = model_rq;
                break;
            case 5:
                model = model_rh;
                break;
            case 6:
                model = model_rp;
                break;
            default:
                break;
            }

            // Generate a face approximation by back-projecting the
eigenvectors & eigenvalues.
            Mat reconstructedFace;
            reconstructedFace = reconstructFace(model, testSample);

            // Verify whether the reconstructed face looks like the
preprocessed face, otherwise it is probably an unknown person.

            double similarity = getSimilarity(testSample,
reconstructedFace);


            //if (similarity < UNKNOWN_PERSON_THRESHOLD) {
                // Identify who the person is in the preprocessed face
image.
                identity = model->predict(testSample);
            //}
            //if (realLabel == 0 or realLabel == 6){
            //    cout << "rpose: " <<realLabel << " ; " <<
predictedLabel << " rid: "<< testLabel << " ; "<< identity << endl;
            //}

            if (identity == testLabel) {
                correctPredictions++;
                correct_p[predictedLabel] = correct_p[predictedLabel] +
1;
            }
        }
        double tasaAciertos = (double)correctPredictions /
(double)test_data.size();
        train_error.push_back(tasaAciertos);
        string result_message = format("Predicciones correctas = %d, tasa
= %f.", correctPredictions, tasaAciertos);
        cout << result_message << endl;
        for (int i = 0;i < 7;i++) {
            double tasaAciertos = 0;
            if ((double)total_p[i] > 0) {
                tasaAciertos = (double)correct_p[i] / (double)total_p[i];
            }
            train_errors[i].push_back(tasaAciertos);
```

```
            string result_message = format("Predicciones correctas pose
%i = %d de %d, tasa = %f.", i, correct_p[i], total_p[i], tasaAciertos);

            cout << result_message << endl;
        }
        end = std::chrono::high_resolution_clock::now();
        elapsed =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
        cout << "Tiempo empleado en evaluar modelo: " << elapsed << "
milliseconds" << endl;
        test_time.push_back(elapsed);
        //Save Data
        FileStorage fs(dataname, FileStorage::WRITE);
        fs << "TrainTime" << train_time;
        fs << "TestTime" << test_time;
        fs << "TotalError" << train_error;
        fs << "Pose1" << train_errors[0];
        fs << "Pose2" << train_errors[1];
        fs << "Pose3" << train_errors[2];
        fs << "Pose4" << train_errors[3];
        fs << "Pose5" << train_errors[4];
        fs << "Pose6" << train_errors[5];
        fs << "Pose7" << train_errors[6];
        fs.release();
    }
    return 0;
}
```

REFERENCIAS

Bach, F., & Jordan, M. (2003). Kernel independent component analysis. The Journal of Machine Learning Research, 3 , 1–48. Recuperado de: http://dl.acm.org/citation.cfm?id=944920

Bartlett, M. (2002). Face recognition by independent component analysis. Neural Networks, IEEE, 13 (6), 1450–1464. Recuperado de: http://ieeexplore.ieee.org/xpls/absall.jsp?arnumber=1058079

Belhumeur, P., Hespanha, J., & Kriegman, D. (1997, July). Eigenfaces vs. Fisherfaces: recognition using class specific linear projection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19 (7), 711–720. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=598228, doi: 10.1109/34.598228

Beymer, D. (1994). Face recognition under varying pose. Computer Vision and Pattern Recognition (1461).

Blanz, V., & Vetter, T. (1999). A morphable model for the synthesis of 3D faces. Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99, 187–194. doi: 10.1145/311535.311556

Blanz, V., & Vetter, T. (2003, September). Face recognition based on fitting a 3D morphable model. IEEE Transactions on Pattern Analysis and Machine Intelligence, 25 (9), 1063–1074. doi: 10.1109/TPAMI.2003.1227983

Bouvrie, J. (2006). Notes on Convolutional Neural Networks.

Bradski, G., & Kaehler, A. (2008). Learning OpenCV: Computer Vision in C++ with the OpenCV Library (1st ed. ed.). O'Reilly Media.

Bronstein, A., Bronstein, M., Gordon, E., & Kimmel, R. (2004). Fusion of 2D and 3D data in three-dimensional face recognition. In 2004 international conference on image processing, 2004. icip '04. (Vol. 1, pp. 87–90). IEEE. doi: 10.1109/ICIP.2004.1418696

Chen, L., Man, H., & Nefian, A. V. (2005, June). Face recognition based on multi-class mapping of Fisher scores. Pattern Recognition, 38 (6), 799–811. doi: 10.1016/j.patcog.2004.11.003

Ciresan, D. C., Giusti, A., Gambardella, L. M., & Schmidhuber, J. (2012). Deep neural networks segment neuronal membranes in electron microscopy images. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), Nips (p. 2852- 2860). Recuperado de: http://dblp.uni-trier.de/db/conf/nips/nips2012.html#CiresanGGS12

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (2011). Flexible, high performance convolutional neural networks for image classification. In T. Walsh (Ed.), Ijcai (p. 1237-1242). IJCAI/AAAI. Recuperado de: http://dblp.uni-trier.de/db/conf/ijcai/ijcai2011.html#CiresanMMGS11

Ciresan, D. C., Meier, U., Masci, J., & Schmidhuber, J. (2011). A committee of neural networks for traffic sign classification. In Ijcnn (p. 1918-1921). IEEE. Recuperado de: http://dblp.uni-trier.de/db/conf/ijcnn/ijcnn2011.html#CiresanMMS11

Cootes, T., & Taylor, C. (2004). Statistical models of appearance for computer vision. Science and Biomedical Engineering.

Draper, B., Baek, K., Bartlett, M., & Beveridge, J. (2003). Recognizing Faces with PCA and ICA. Computer Vision and Image Understanding, Special Issue on Face Recogni- tion, 1–26. Recuperado de: http://www.sciencedirect.com/science/article/pii/S1077314203000778

Georghiades, A., Belhumeur, P., & Kriegman, D. (2001, June). From few to many: illu-mination cone models for face recognition under variable lighting and pose. IEEE Transactions on Pattern Analysis and Machine Intelligence, 23 (6), 643–660. doi: 10.1109/34.927464

Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks.. Recuperado de: http://arxiv.org/abs/1302.4389

Guo, G., Li, S. Z., & Chan, K.(2000). Face recognition by support vector machines. In Proceedings fourth ieee international conference on automatic face and gesture re- cognition (cat. no. pr00580) (pp. 196–201). IEEE Comput. Soc. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=840634, doi: 10.1109/AFGR.2000.840634

Hinton, G., & Srivastava, N. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv: . . . , 1–18. Recuperado de: http://arxiv.org/abs/1207.0580

Jonsson, K., Matas, J., Kittler, J., & Li, Y. (2000). Learning support vectors for face verification and recognition. In Proceedings fourth ieee international conference on automatic face and gesture recognition (cat. no. pr00580) (pp. 208–213). IEEE Comput. Soc. Recu- perado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=840636, doi: 10.1109/AFGR.2000.840636

Ju, Q. (2010). A high performance automatic face recognition system using 3D shape information. Unpublished doctoral dissertation, University of York. Recuperado de: http://etheses.whiterose.ac.uk/1200/

Lawrence, S., Giles, C., Tsoi, A., & Back, A. (1998). Face recognition: A hybrid neural network approach, pp. 1–22. Recuperado de: http://drum.lib.umd.edu/handle/1903/803

Lawrence, S., Giles, C. L., Tsoi, a. C., & Back, a. D. (1997, January). Face recognition: a convolutional neural-network approach. IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council, 8 (1), 98–113. Recuperado de: http://www.ncbi.nlm.nih.gov/pubmed/18255614, doi: 10.1109/72.554195

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE , 86 (11), 2278–2324. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=726791, doi: 10.1109/5.726791

LeCun, Y., Bottou, L., Orr, G., & Mu¨ller, K. (1998). Efficient backprop. Neural networks.

LeCun, Y., Jackel, L., & Bottou, L. (1995). Comparison of learning algorithms for handwritten digit recognition. RT Conference Proceedings. Recuperado de: http://mleg.cse.sc.edu/edu/csce822/uploads/Main.ReadingList/KNNrecognition.pdf

Li, S. Z., Zhu, L., Zhang, Z., Blake, A., Zhang, H., & Shum, H. (2002). Statistical learning of multi-view face detection. In In proceedings of the 7th european conference on computer vision (pp. 67–81).

Moghaddam, B., Pfister, H., & Machiraju, R. (2004). Finding optimal views for 3D face shape modeling. Sixth IEEE International Conference on Automatic Face and Gesture Recognition, 2004. Proceedings., 31–36. doi: 10.1109/AFGR.2004.1301505

Moon, H., & Phillips, P. (2001). Computational and performance aspects of PCA-based face-recognition algorithms. Perception-London.

Murphy-Chutorian, E., & Trivedi, M. M. (2009, April). Head pose estimation in computer vision: a survey. IEEE transactions on pattern analysis and machine intelligence, 31 (4), 607–26. doi: 10.1109/TPAMI.2008.106

Nefian, A. (1996). Statistical approaches to face recognition.

Nefian, A. (1999). A hidden Markov model-based approach for face detection and recognition. Unpublished doctoral dissertation, Georgia Institute of Technology.

Nefian, A., & Hayes, M. (1998). Face detection and recognition using hidden Markov models. In Proceedings 1998 international conference on image processing. icip98 (cat. no.98cb36269) (Vol. 1, pp. 141–145). IEEE Comput. Soc. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=723445, doi: 10.1109/ICIP.1998.723445

O'Neill, M. (2006). Neural Network for Recognition of Handwritten Digits.

Pesquisa, P. D., Leonel, L., & Junior, D. O. (2005). Relatório Final Captura e Alinhamento de Imagens : Um Banco de Faces Brasileiro. , 1–10.

Phillips, P., Wechsler, H., Huang, J., & Rauss, P. J. (1998, April). The FERET database and evaluation procedure for face-recognition algorithms. Image and Vision Computing, 16 (5), 295–306. doi: 10.1016/S0262-8856(97)00070-X

Reddy, T. H. (2012). Multi-View Facial Recognition Using Eigenfaces By PCA And Artificial Neural Network. , 2 (1), 24–27.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986, October). Learning representations by back-propagating errors. Nature, 323 (6088), 533–536. doi: 10.1038/323533a0

Simard, P., Steinkraus, D., & Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. In Seventh international conferen- ce on document analysis and recognition, 2003. proceedings. (Vol. 1, pp. 958–963). IEEE Comput. Soc. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1227801, doi: 10.1109/ICDAR.2003.1227801

Singh, R., Vatsa, M., Ross, A., & Noore, A. (2007, October). A mosaicing scheme for pose-invariant face recognition. IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics: a publication of the IEEE Systems, Man, and Cybernetics Society , 37 (5), 1212–25.

Srivastava, N. (2013). Improving neural networks with dropout. Unpublished doctoral disser- tation, University of Toronto. Recuperado de: http://www.cs.toronto.edu/~nitish/msc thesis.pdf

Stiefelhagen, R. (2004). Estimating Head Pose with Neural Networks - Results on the Pointing04 ICPR Workshop Evaluation Data. Pointing'04 ICPR Workshop, 8–11. Recupe- rado de: https://cvhci.anthropomatik.kit.edu/~stiefel/papers/pointing2004final.pdf

Turk, M., & Pentland, A. (1991). Eigenfaces for recognition. Journal of cognitive neuroscience.

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001, 1 , I–511–I–518. doi: 10.1109/CVPR.2001.990517

Viola, P., & Jones, M. (2004). Robust real-time face detection. International journal of computer vision, 57 (2), 137–154.

Wang, W., Shan, S., Gao, W., Cao, B., & Yin, B. (2002). An improved active shape model for face alignment. In Proceedings. fourth ieee international conference on multimodal interfaces (pp. 523–528). IEEE Comput. Soc. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1167050, doi: 10.1109/ICMI.2002.1167050

Werbos, P. (1988). Backpropagation: past and future. In Ieee international conference on neural networks (Vol. 1, pp. 343–353 vol.1). IEEE. doi: 10.1109/ICNN.1988.23866

Weyrauch, B., Heisele, B., Huang, J., & Blanz, V. (2004). Component-Based Face Recognition with 3D Morphable Models. In 2004 conference on computer vision and pattern recognition workshop (pp. 85–85). IEEE. Recuperado de: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1384878, doi: 10.1109/CVPR.2004.315

Yang, M. (2001). Face recognition using kernel methods. Proc. of NIPS'01 .

Zhang, C., & Zhang, Z. (2010). A survey of recent advances in face detection. (June). Recuperado de: http://202.114.89.42/resource/pdf/6582.pdf, doi: 10.1.1.167.5270

Zhang, X., & Gao, Y. (2009, November). Face recognition across pose: A review. Pattern Recognition, 42 (11), 2876–2896. doi: 10.1016/j.patcog.2009.04.017

Zhao, W., Chellappa, R., Phillips, P. J., & Rosenfeld, A. (2003, December). Face recognition, ACM Computing Surveys, 35 (4), 399–458. doi: 10.1145/954339.954342