Trabajo Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica

Mención en Instrumentación Electrónica y Control

Aceleración del algoritmo de Viola-Jones mediante rejillas de procesamiento masivamente paralelo en el plano focal

Autor: Eloy Parra Barrero

Tutores: Ángel Rodríguez Vázquez, Jorge Fernández Berni

Dep. de Electrónica y Electromagnetismo

Área de Electrónica

Escuela Técnica Superior de Ingeniería

Sevilla, 2015

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica
Mención en Instrumentación Electrónica y Control

Aceleración del algoritmo de Viola-Jones mediante rejillas de procesamiento masivamente paralelo en el plano focal

Autor:
Eloy Parra Barrero

Tutores:
Ángel Rodríguez Vázquez
Catedrático de Universidad
Jorge Fernández Berni
Profesor Interino Asimilado a Asociado

Dep. de Electrónica y Electromagnetismo
Área de Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado: Aceleración del algoritmo de Viola-Jones mediante rejillas de procesamiento masivamente paralelo en el plano focal

Autor: Eloy Parra Barrero

Tutores: Ángel Rodríguez Vázquez, Jorge Fernández Berni

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

A mi abuelo Eloy, a quien le hubiera gustado ver este trabajo.

Agradecimientos

A mis tutores, Ángel Rodriguez Vázquez y Jorge Fernández Berni por su orientación y apoyo a lo largo de todo el desarrollo del trabajo y en especial en los últimos días de tensión en los que ni siquiera el océano Atlántico ha podido interponerse en la ayuda prestada.

Eloy Parra Barrero

Sevilla, 2015

Resumen

El algoritmo de Viola-Jones es un método de detección de objetos que se usa ampliamente en la detección de caras en imágenes y video. El algoritmo se basa en la comparación entre las intensidades luminosas de regiones rectangulares de las imágenes denominadas Haar-like features que calcula empleando una imagen integral.

En el Instituto de Microelectrónica de Sevilla se ha desarrollado un sensor de imagen con procesamiento en el plano focal que, entre otras operaciones, es capaz de realizar de manera masivamente paralela una pixelación reconfigurable de la imagen.

En este trabajo se estudia la posibilidad de emplear esta funcionalidad del sensor para sustituir al cálculo de la imagen integral en la obtención de las Haar-like features y conseguir así una aceleración del proceso de detección.

Se ha trabajado con una implementación del algoritmo disponible en la librería de visión artificial OpenCV y se han tratado de adaptar las features de una cascada facilitada en la librería de modo que puedan ser calculadas a partir de la información que proporcione el chip empleando unas pocas rejillas de pixelación.

Debido al gran número y variedad de features que componen la cascada, se decidió adaptar únicamente la primera etapa de la misma, que es la más discriminativa, dejando que el procesador ejecute el resto de la cascada sobre las ventanas de búsqueda que pasen la primera etapa.

Con la ayuda de caras genéricas se han adaptado algunas features de la primera etapa original y se han diseñado otras nuevas obteniendo dos diseños alternativos de la primera etapa que podrían realizarse empleando información proporcionada por el chip. Uno de los diseños rechaza más del 97% de las ventanas de búsqueda sin caras manteniendo aproximadamente el 95% de las caras. Utilizando esta primera etapa, la detección se realiza en un 6.6% del tiempo que tarda el algoritmo original.

Como resultado secundario se expone que utilizando una de las primeras etapas diseñadas con el algoritmo original y por tanto desligada del chip, se consigue un ahorro en el tiempo de cómputo del 38% con respecto al algoritmo original manteniendo la sensibilidad por encima del 98%.

Abstract

The Viola-Jones object detection framework is one of the world's most used algorithms for face detection in images or video. It relies on the comparison between the brightness of different rectangular regions called Haar-like featres which can be obtained by using an integral image.

An image sensor with focal-plane processing has been developed in the Institute of Microelectronics of Seville which has the capacity to provide programmable pixelation of multiple rectangular image regions in parallel.

In this project, we study how to use this functionality as a substitute for the computation of the integral image when it comes to obtaining Haar-like features, thus accelerating the detection process.

We have worked with an implementation of the algorithm and a cascade of features provided by the computer vision library OpenCV and we have tried to adapt the set of features so as to be able to calculate them from the information produced by the chip using a few pixelation grids.

The set of features is extremely large and varied so we decided to adapt only the features in the first stage of the cascade which is the most discriminative stage. This allows the processor to run the rest of the cascade only on the detection windows that pass the first stage.

With the help of generic faces, we have adapted some of the features in the original first stage and designed a few others resulting in two alternative designs for the first stage that can be calculated from the information provided by the chip. With one of the designs, more than 97% of the detection windows are discarded in the first stage while preserving approximately 95% of the faces. Using this first stage, the detection process takes only a 6.6% of the time it takes for the original algorithm and cascade.

As a secondary result, we have measured that one of the first stages designed, applied within the original algorithm and therefore independent of the chip, can result in a 38% reduction in computation time while keeping the detection efficiency over 98%.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La visión artificial juega un papel fundamental en el desarrollo de sistemas inteligentes y encuentra aplicación en áreas como el control de procesos, la navegación de vehículos, la supervisión y vigilancia, la interacción usuario-máquina o la búsqueda y organización de información en las cada vez mayores bases de datos.

El rápido aumento en la capacidad de cómputo de los micro-procesadores ha permitido un gran desarrollo de los sistemas de visión artificial en las últimas décadas.

Dentro de esta disciplina existen diversas áreas como la reconstrucción de escenas para obtener modelos 3D, la restauración de imágenes ruidosas empleando filtros o modelos y principalmente el reconocimiento visual, es decir, determinar si un determinado objeto, clase o característica está presente en una imagen.

El reconocimiento visual a su vez se puede clasificar en reconocimiento de objetos, cuando se buscan instancias de una determinada clase, identificación, cuando se trata de reconocer a un objeto en particular, como una cara en concreto o la huella dactilar de una persona, y la detección, en donde se buscan ciertas características como tejido celular anormal o defectos de fabricación en una cadena de montaje.

Actualmente, los algoritmos que consiguen mejor resultado en este tipo de tareas se basan en redes neuronales convolucionales. El desempeño de estos sistemas en los bancos de prueba de ImageNet está cerca de la de los humanos [1]. La figura 1-1 muestra dos ejemplos de interpretación semántica y descripción de imágenes empleando esta técnica.

Figura 1-1. Interpretación y descripción de imágenes mediante redes neuronales convolucionales. Primera imagen: "Un hombre está al lado de un elefante". Segunda imagen: "Un hombre cabalga por una calle al lado de un edificio" [2]

Estos sistemas son muy costosos computacionalmente, lo cual no resulta extraño si tenemos en cuenta que aproximadamente la mitad del cortex cerebral de los primates está involucrado en el procesamiento de información visual [3]. No obstante, existen tareas que pueden realizarse de manera muy eficiente con menos carga computacional, como es el caso de la detección de caras. En particular, el procedimiento desarrollado por Paul Viola y Michael Jones y presentado en 2001 [4] supuso una revolución en este campo, permitiendo por primera vez la detección de caras en tiempo real en dispositivos de prestaciones modestas.

El método de Viola-Jones y variaciones de éste siguen siendo ampliamente utilizadas en la actualidad.

Éste método lleva a cabo básicamente una comparación entre valores medios de intensidades luminosas en regiones rectangulares de las imagenes denominadas Haar-like features. Este modo de operación convierte al algoritmo de Viola-Jones en candidato para una posible aceleración desde el mismo inicio de la cadena de procesamiento en sistemas de visión, esto es, el plano focal.

1.1 Procesamiento en el plano focal

La visión artificial se suele dividir en los tres estadios consecutivos que se muestran en la figura 1-2: visión temprana, tareas a medio nivel y procesamiento a alto nivel [5]. La visión temprana opera en cada píxel resultado de la lectura del sensor y realiza las mismas operaciones con cada uno de ellos. Las tareas a medio nivel no trabajan con los datos en bruto sino con una menor cantidad de información de mayor nivel de abstracción. Finalmente, el procesamiento a alto nivel se encarga de dar una interpretación semántica de la escena y opera con un volumen muy reducido de datos y con un flujo de instrucciones muy irregular.

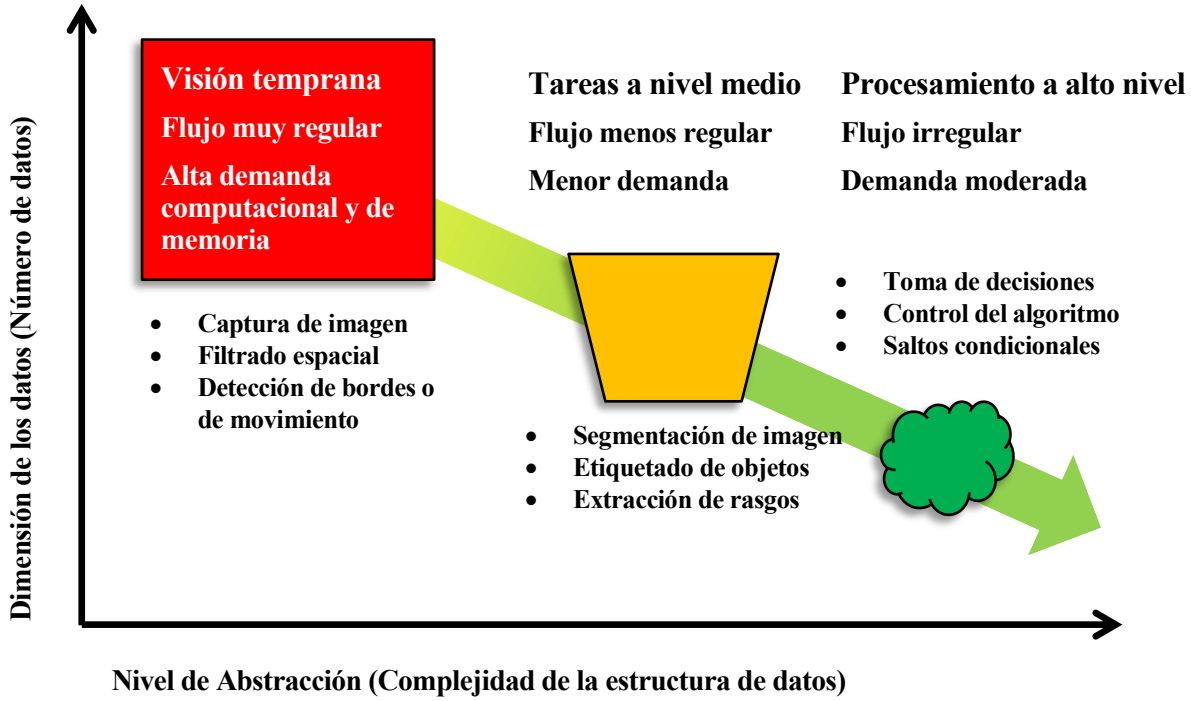

Figura 1-2. Comparación entre la dimensión de los datos y el nivel de abstracción en la visión artificial [6]. La visión temprana representa la etapa crítica en términos de demanda computacional y de memoria.

Debido a la naturaleza relativamente simple y repetitiva de las operaciones que se realizan en la visión temprana, así como del gran volumen de información que se maneja, el procesamiento masivamente paralelo en el plano focal se presenta como la arquitectura más indicada para realizar este tipo de tareas. Además, las arquitecturas de procesamiento en el plano focal pueden incorporar electrónica analógica y beneficiarse de sus mayores prestaciones en cuanto a velocidad, área y consumo de potencia aprovechando las menores exigencias de precisión de la visión temprana.

El procesamiento en el plano focal está inspirado en la operación de los sistemas de visión natural donde el sensor, es decir, la retina, no solo adquiere la información sino que también la pre-procesa obteniendo como resultado un flujo de información por el nervio óptico comprimido por un factor de ~100 [7].

Aparte de mejorar la eficiencia de los sistemas de procesamiento de imágenes, el procesamiento en el plano focal puede ayudar a mejorar la privacidad de éstos.

Cada vez estamos más monitorizados por la tecnología y es difícil garantizar que la información se utilice únicamente por las personas legitimadas para ello y para los fines acordados. Por ello, cuanta menos información salga del plano focal, más seguro será el sistema. En el área del reconocimiento de caras esto resulta especialmente importante.

Con estos objetivos en mente, se ha desarrollado en el Instituto de Microelectrónica de Sevilla un sensor con procesamiento en el plano focal [6] que permite realizar, entre otras operaciones, una pixelación reconfigurable.

Los patrones de reconfiguración se cargan en serie en dos registros de desplazamiento que determinan qué filas y columnas interactúan entre sí. Existe asimismo la posibilidad de cargar en paralelo hasta 6 patrones diferentes para 6 escalas de pixelación sucesivas.

Las señales de reconfiguración se corresponden con las señales $EN_{S_{i,i+1}}$, $EN_{S_{j,j+1}}$, $\overline{EN_{S_{i,i+1}}}$, $\overline{EN_{S_{j,j+1}}}$ a nivel de píxel, donde las coordenadas (i, j) indican la posición del píxel en el array. Estas señales controlan la activación de interruptores MOS para la redistribución de carga entre capacidades nMOS. La redistribución de carga es el proceso que da soporte a todas las funcionalidades del dispositivo y permitiría acelerar el cálculo de las Haar-like features así como obtener una versión promediada de la imagen integral que es la representación intermedia empleada en el algoritmo de Viola-Jones para calcular las Haar-like features.

La figura 1-3 muestra la arquitectura del sensor junto con un esquemático y layout de un píxel multi-función.

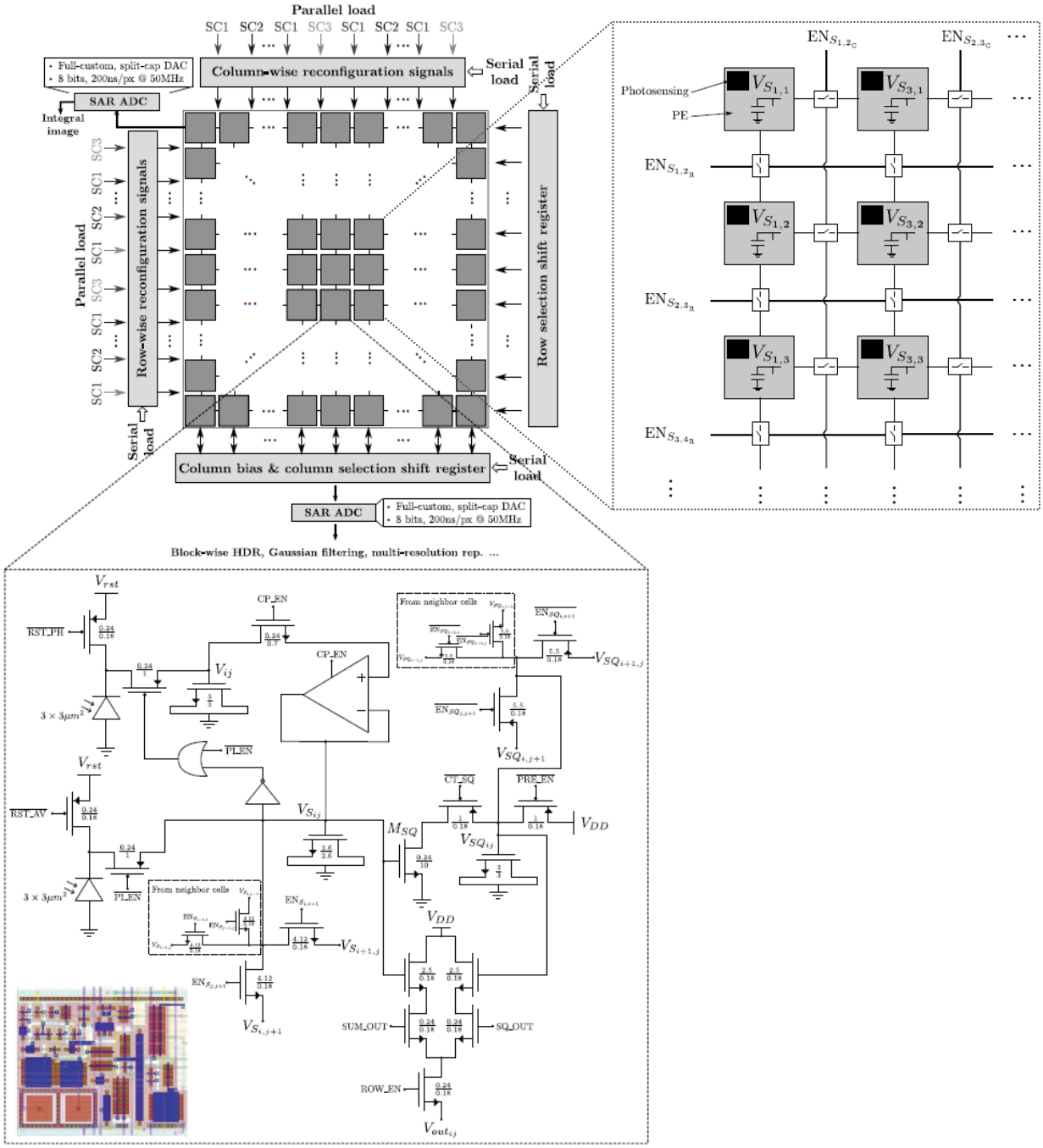

Figura 1-3. Arquitectura del sensor con procesamiento en el plano focal junto con un esquemático y layout de un píxel multi-función de señal mixta [6].

En la figura 1-4 se muestra una imagen capturada por el chip y el resultado de aplicar un determinado patrón de interconexión de los píxeles.

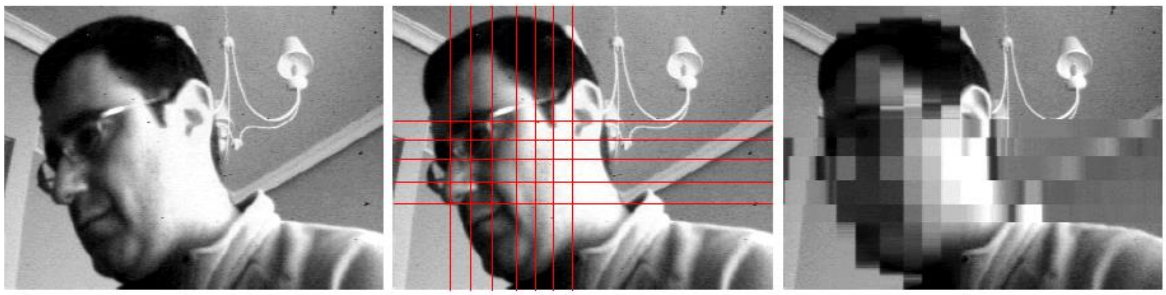

Figura 1-4. (a) Imagen original capturada por el chip. (b) Patrones para la interconexión de los píxeles. (c) resultado de la pixelación. [6]

1.2 Objetivo del trabajo

Como se explicará más adelante, el algoritmo de Viola-Jones emplea una imagen integral para obtener las Haar-like features. En este trabajo se pretende explorar la posibilidad de sustituir el uso de la imagen integral por los valores que podría proporcionar el sensor descrito anteriormente mediante redistribución de carga y por tanto reducir el coste computacional del algoritmo y mejorar la privacidad del sistema haciendo que menos información salga del plano focal.

En particular, se estudiará la posibilidad de utilizar unos pocos patrones de redistribución de carga (que denominaremos "rejillas") para obtener las features necesarias para el algoritmo.

Puesto que realizar un entrenamiento para obtener toda una serie de features nuevas es un proceso de excesiva complejidad para el alcance de este trabajo, se partirá de una cascada de features ya entrenada y se intentará adaptarlas para que puedan ser calculadas a partir de la información proporcionada por el chip.

1.3 Metodología

En este trabajo primero se describirán las características más destacadas del algoritmo de Viola-Jones y después se trabajará con una implementación del mismo disponible en la librería OpenCV. A continuación se describen brevemente las herramientas y criterios de evaluación empleados en la parte práctica del trabajo.

1.2.1 Librería de visión artificial OpenCV

Open Source Computer Vision (OpenCV) es una librería de visión artificial y machine learning de licencia BSD, lo que permite que sea empleada gratuitamente en aplicaciones tanto académicas como comerciales. Se estima que OpenCV cuenta con una comunidad activa de más de 47000 usuarios y el número de descargas supera los 9 millones [8].

OpenCV está escrito en C/C++ optimizado para operar en tiempo real y cuenta con más de 2500 algoritmos optimizados para detección y seguimiento de objetos, extracción de modelos 3D y visión estereoscópica, boosting, redes neuronales y máquinas de vectores de soporte entre otros.

En este trabajo utilizaremos las funciones que implementan el algoritmo de Viola-Jones.

1.2.2 Entorno de trabajo

Se ha empleado principalmente Eclipse Luna 4.4.1. para C/C++ como entorno de desarrollo y MinGW como compilador. Ocasionalmente se ha hecho uso de MatLab y Excel para la obtención de gráficas.

1.2.3 Banco de pruebas

Para comprobar el rendimiento del algoritmo se ha utilizado un banco de caras frontales de Caltech [9]. El banco cuenta con 450 caras de 27 personas diferentes desde una perspectiva frontal y la resolución de las imágenes es de 896x592.

1.2.4 Criterios de evaluación de rendimiento

El rendimiento del detector de caras se medirá empleando las siguientes figuras de mérito [10]:

- True positive: número de detecciones acertadas.
- False positive: número de detecciones que no contienen el objeto buscado.
- False negative: número de instancias del objeto buscado que no son detectadas.
- Precision: ratio entre true positive y el total de detecciones realizadas.

$$PR = \frac{tp}{tp + fp} \tag{1-1}$$

- Recall o sensibilidad: ratio entre true positive y el total de posibles detecciones existentes en la imagen.

$$RC = \frac{tp}{tp + fn} \tag{1-2}$$

2 EXPLORACIÓN DEL ALGORITMO DE VIOLA-JONES

El algoritmo de Viola-Jones es un método de detección de objetos que destaca por su bajo coste computacional, lo que permite que sea empleado en tiempo real. Su desarrollo fue motivado por el problema de la detección de caras, donde sigue siendo ampliamente utilizado, pero puede aplicarse a otras clases de objetos que, como las caras, estén caracterizados por patrones típicos de iluminación [10][11].

El algoritmo se basa en una serie de clasificadores débiles denominados Haar-like features que se pueden calcular eficientemente a partir de una imagen integral. Estos clasificadores, que por sí mismos tienen una probabilidad de acertar solo ligeramente superior a la del azar, se agrupan en una cascada empleando un algoritmo de aprendizaje basado en AdaBoost para conseguir un alto rendimiento en la detección así como una alta capacidad discriminativa en las primeras etapas.

2.1. Haar-like features

Las Haar-like features son los elementos básicos con los que se realiza la detección. Reciben este nombre por similitud a los wavelet de Haar [12]. Estas features son rasgos o características muy simples que se buscan en las imágenes y que consisten en la diferencia de intensidades luminosas entre regiones rectangulares adyacentes. Las features por tanto quedan definidas por unos rectángulos y su posición relativa a la ventana de búsqueda y adquieren un valor numérico resultado de la comparación que evalúan.

En el trabajo presentado por Viola-Jones existen tres tipos de features representadas en la figura 2-1:

- Features de dos rectángulos cuyo valor es la diferencia entre las sumas de los píxeles contenidos en ambos rectángulos. Las regiones tienen la misma área y forma y son adyacentes.
- Features de tres rectángulos que calculan la diferencia entre los rectángulos exteriores y el interior multiplicado por un peso para compensar la diferencia de áreas.
- Features de cuatro rectángulos que computan la diferencia entre pares diagonales de rectángulos.

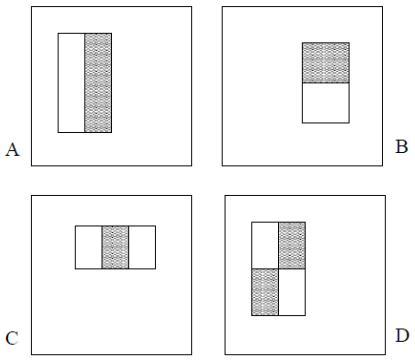

Figura 2-1. Ejemplos de features de dos, tres y cuatro rectángulos y su posición relativa a la ventana de búsqueda [4]. La suma de los píxeles en las áreas grises se resta a la de las áreas blancas.

En el trabajo de Viola-Jones, las features se definen sobre una ventana de búsqueda básica de 24x24 píxeles, lo que da lugar a más de 180000 features posibles.

2.2. Imagen integral

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

$$ii(x,y) = \sum_{x' \le x,\ y' \le y} i(x',y') \tag{2-1}$$

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

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

$$s(x,y) = s(x,y-1) + i(x,y) \tag{2-2}$$

$$ii(x,y) = ii(x-1,y) + s(x,y) \tag{2-3}$$

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

Usando la imagen integral, cualquier suma rectangular se puede calcular con cuatro referencias a memoria como se muestra en la figura 2-2. Las features de dos rectángulos se pueden computar con 6 referencias a memoria puesto que comparten vértices. En el caso de features de tres rectángulos se pasa a 8, y a 9 para features de cuatro rectángulos.

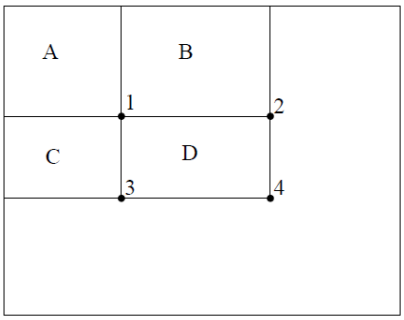

Figura 2-2. Ejemplo del cálculo de la suma de píxeles en un rectángulo [13]. El valor de la suma de los píxeles en el rectángulo D es igual al valor de la imagen integral en 4 (A+B+C+D) menos el valor de la imagen integral en 2 y en 3 (A+B, A+C) y más el valor de la imagen integral en 1 (A).

2.3. Proceso de aprendizaje

Es necesario realizar un proceso de entrenamiento supervisado para crear la cascada de clasificadores. Este proceso se realiza mediante un algoritmo basado en AdaBoost, un meta-algoritmo adaptativo de machine learning cuyo nombre es una abreviatura de adaptative boosting.

El boosting consiste en tomar una serie de clasificadores débiles y combinarlos para construir un clasificador fuerte con la precisión deseada. AdaBoost fue introducido por Freund y Schapire en 1995 resolviendo muchas de las dificultades prácticas asociadas al proceso de boosting [14].

En el procedimiento de Viola-Jones, AdaBoost se utiliza tanto para seleccionar un pequeño set de features de las 180000 posibles como para entrenar el clasificador.

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

$$h_j(x) = \begin{cases} 1 & si\ p_j f_j(x) < p_j \theta_j \\ 0 & e.o.c \end{cases} \quad (2\text{-}4)$$

A continuación se describe el algoritmo de AdaBoost empleado y se incluye una representación visual del proceso (figura 2-3). En cada ronda se selecciona un clasificador débil y por tanto una feature.

- Se parte de un conjunto de imágenes $(x_1, y_1),\ldots,(x_n, y_n)$ donde $y_i = 0, 1$ para ejemplos negativos y positivos respectivamente.
- Se inicializan los pesos $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ para $y_i = 0, 1$ respectivamente donde m es el número de negativos y l el número de positivos.
- Para cada ronda, $t = 1,\ldots,T$:
  1. Normalizar los pesos:

$$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}} \quad (2\text{-}5)$$

  2. Para cada feature, j, entrenar un clasificador h_j que solo use una feature. El error se evalúa teniendo en cuenta los pesos w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$
  3. Se escoge el clasificador, h_t, con menor error ϵ_t.
  4. Se actualizan los pesos:

$$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i} \quad (2\text{-}6)$$

  Donde $e_i = 0$ si el ejemplo x_i se clasifica correctamente y 1 en caso contrario y $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$
- El clasificador fuerte final es:

$$h(x) = \begin{cases} 1 & si \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & e.o.c \end{cases} \quad (2\text{-}7)$$

donde $\alpha_t = \frac{1}{\beta_t}$.

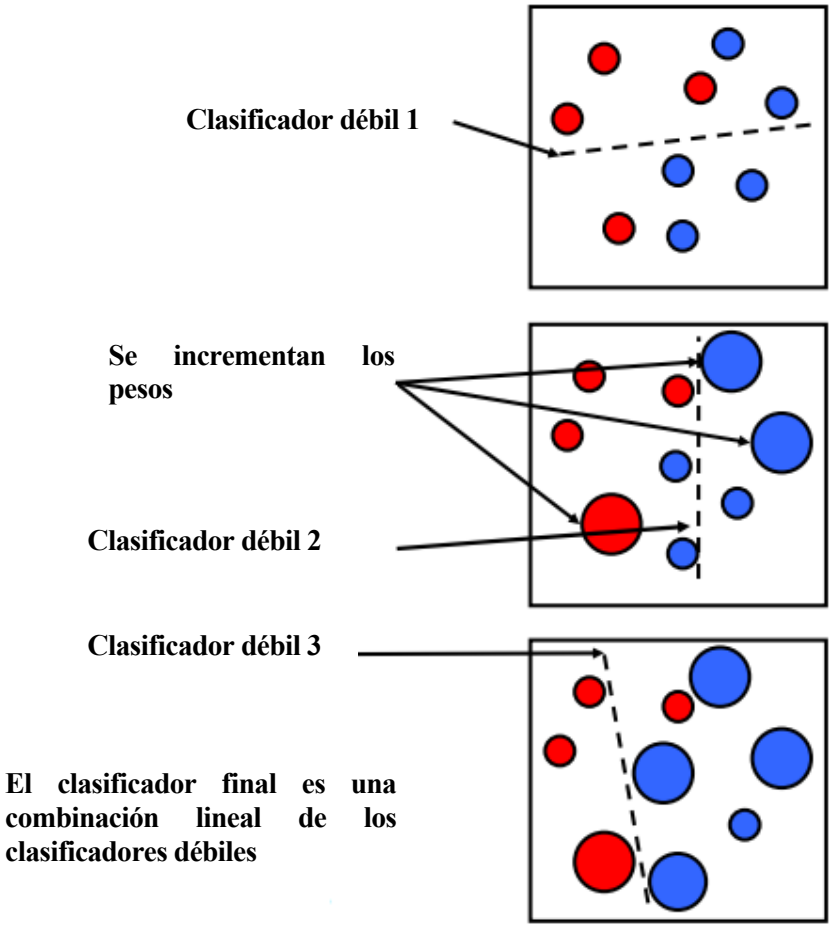

Figura 2-3. Representación visual del proceso de AdaBoost [15]. En cada ronda se aumenta el peso de los ejemplos mal clasificados anteriormente y se busca un nuevo clasificador que minimice el error.

2.4. Cascada atencional

En vez de construir un único clasificador mediante el proceso descrito en el apartado anterior, se pueden construir clasificadores más pequeños y eficientes que rechacen muchas ventanas negativas (es decir, aquellas que no incluyan ninguna instancia del objeto buscado) manteniendo casi todas las positivas (es decir, las que contienen una instancia del objeto buscado). Estos clasificadores más simples se utilizan para rechazar la mayoría de las ventanas de búsqueda y solo en aquellas en las que hay mayores probabilidades de encontrar caras se llama a clasificadores más complejos que disminuyan el número de falsos positivos. Este proceso se representa en la figura 2-4.

Se obtiene así una cascada de clasificadores, cada uno de los cuales es entrenado con AdaBoost y después sus valores umbrales se ajustan para minimizar los falsos negativos.

La cascada entrenada por Viola-Jones tiene 38 etapas y más de 6000 features pero de media se evalúan únicamente 10 features por ventana de búsqueda [4].

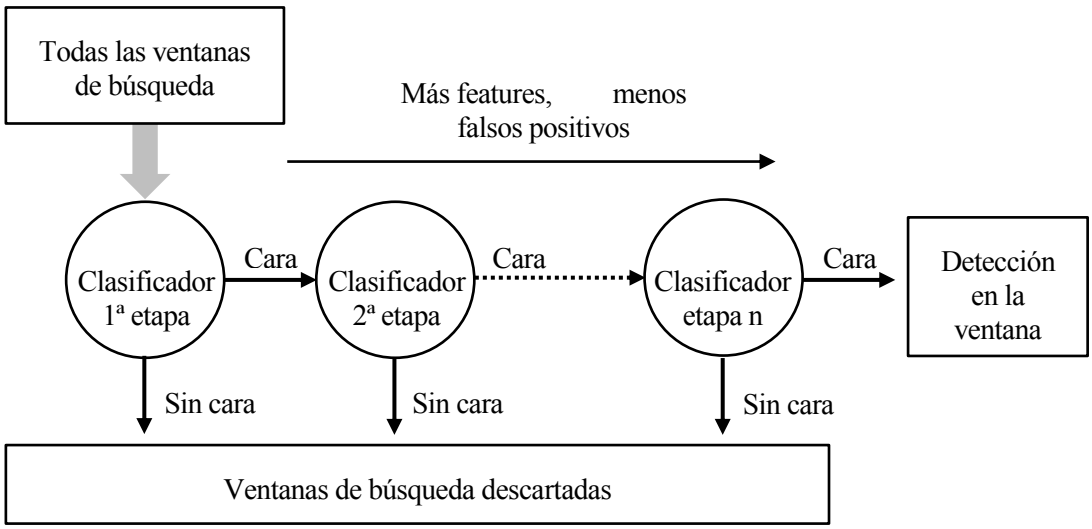

Figura 2-4. Descripción esquemática de una cascada atencional [10]. Una serie de clasificadores se aplican a cada ventana de búsqueda. Los clasificadores iniciales rechazan un gran número de ventanas rápidamente.

2.5. Proceso de detección

Las imágenes usadas para entrenar al algoritmo fueron normalizadas para minimizar los efectos de diferentes condiciones de iluminación, por tanto, también resulta necesario realizar la normalización en el proceso de detección. Para ello, en vez de normalizar la imagen antes de comenzar el análisis, lo cual implicaría cambiar el valor de todos los píxeles, resulta más sencillo corregir los valores de las features conforme se van calculando.

Para normalizar se emplea la varianza:

$$\sigma^2 = m^2 - \frac{1}{N}\sum x^2 \tag{2-8}$$

Donde *m* es la media del valor de los píxeles, que puede calcularse a partir de la imagen integral. La suma de los píxeles al cuadrado se puede obtener a partir de una imagen integral de la imagen al cuadrado.

La cascada de features se evalúa sobre una ventana de búsqueda cuadrada que barre la imagen con incrementos de unos pocos píxeles. La búsqueda se realiza a distintas escalas obtenidas al multiplicar la escala anterior por un factor de escala, normalmente entre 1.1 y 1.3.

Puesto que el detector es poco sensible a pequeñas translaciones y diferencias de escala, se suelen producir múltiples detecciones alrededor de cada cara. De hecho, se puede exigir que las detecciones tengan un determinado número mínimo de detecciones vecinas para disminuir el número de falsos positivos.

Para combinar las detecciones que se refieren al mismo objeto, se fusionan las detecciones cuyas áreas se solapen más de un determinado valor umbral y el recuadro con la detección final se calcula como la media de todos los recuadros que se han fusionado.

2.6. Implementación del algoritmo de detección en OpenCV

OpenCV incluye funciones tanto para entrenar una cascada como para detectar objetos. Aquí se describen únicamente las funciones relativas a la detección.

El algoritmo de detección que se implementa en OpenCV es una versión del algoritmo de Viola-Jones desarrollada por Lienhart que permite emplear features inclinadas a 45º grados [16], aunque la cascada de features que hemos empleado usa únicamente las features originales de Viola-Jones.

Otra particularidad del método empleado en OpenCV es que las features se definen sobre una ventana de búsqueda básica de 20x20 píxeles en vez de 24x24.

El código relativo a las funciones en cuestión se encuentra en la dirección \opencv\sources\modules\objdetect\src dentro del conjunto de ficheros descargados de OpenCV. En concreto, en los ficheros "cascadedetect.cpp" de 1300 líneas de código con funciones para cascadas de tipo Haar, Hog y LBP y "haar.cpp" con más de 2500 líneas de código específicas para cascadas de tipo Haar.

Este código compilado forma parte de la librería "objectdetect".

OpenCV también facilita una serie de cascadas ya entrenadas. En este trabajo hemos empleado la cascada "haarcascade_frontalface_alt" que puede encontrarse en la dirección \opencv\sources\data\haarcascades.

Las funciones de detección se integran en la clase CascadeClassifier y la función de partida es detectMultiScale cuya definición es la siguiente:

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

Los argumentos de ésta función son:

- cascade: cascada de clasificadores Haar. Se carga previamente a partir de un fichero XML o YAML utilizando Load().
- image: matriz de tipo CV_8U sobre la que se busca el objeto u objetos a detectar.
- objects: vector de rectángulos donde cada rectángulo contendrá un objeto detectado.
- scaleFactor: factor de escala que define los incrementos en las escalas de búsqueda.
- minNeigbours: parámetro que especifica cuántos vecinos debe tener cada rectángulo candidato para que sea aceptado.
- flags: selección de algunas opciones para formatos de cascada antiguos.
- minSize: tamaño mínimo de los objetos a detectar.
- maxSize: tamaño máximo de los objetos a detectar.

En la figura 2-5 se muestra la jerarquía de funciones que parte de la función detectMultiscale. En gris están las funciones con las que más se ha trabajado y que se describen brevemente a continuación.

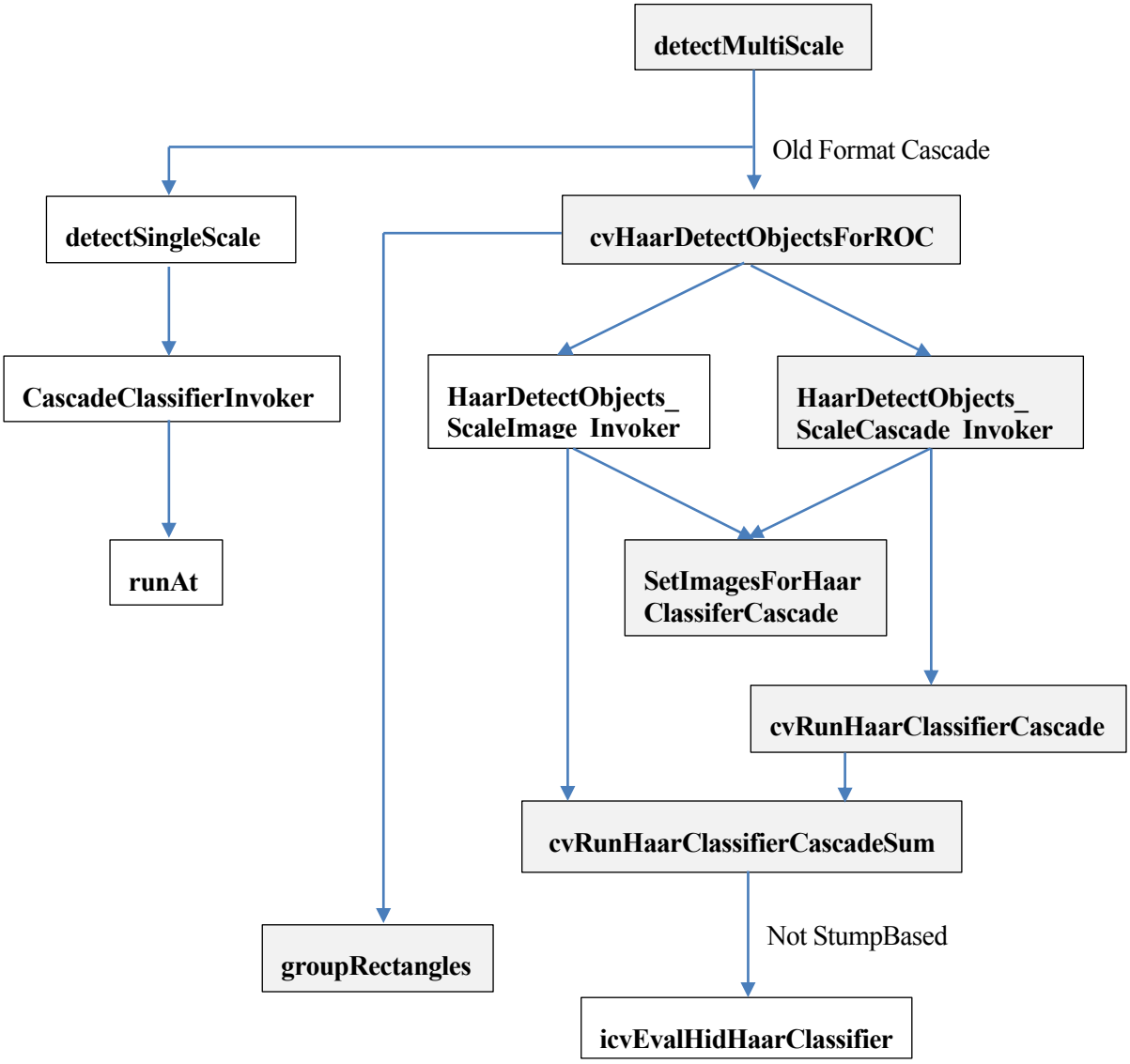

Figura 2-5. Jerarquía de funciones en OpenCV para la implementación del algoritmo de Viola-Jones. En gris están las funciones con las que más se ha trabajado.

En el caso de la cascada empleada, detectMultiScale simplemente llama a la función cvHaarDetectObjectsForROC.

La función cvHaarDetectObjectsForROC se encarga de realizar el escalado y de llamar al resto de funciones necesarias para la detección. En la opción por defecto se escala la cascada de features manteniendo la imagen constante, por tanto, la imagen integral se calcula una sola vez para la imagen original. Con flag= `0|CV_HAAR_SCALE_IMAGE`, se escala la imagen por lo que en cada iteración del escalado hay que calcular una nueva imagen integral. Nosotros hemos trabajado principalmente con la opción por defecto en la que se escala la cascada.

Para cada factor de escala, se llama a HaarDetectObjects_ScaleCascade_Invoker que se encarga de realizar el barrido de la imagen en las direcciones horizontal y vertical. Para cada ventana de búsqueda, se llama a SetImagesForHaarClassiferCascade que inicializa los punteros necesarios para que la cascada se evalúe en dicha ventana de búsqueda y a cvRunHaarClassifierCascadeSum que ejecuta la cascada.

Por último, la función groupRectangles se encarga de combinar las detecciones múltiples en un único rectángulo descartando los rectángulos que tengan menos de un determinado número de vecinos.

Para conseguir una mayor familiaridad con el código y un mejor entendimiento de las features, se ha desarrollado un pequeño programa que muestra por pantalla aquellas features que el algoritmo está evaluando en cada momento. Tanto para este programa como para todos los que se desarrollan en el trabajo, se ha copiado el código original de OpenCV a partir del cual se construyen las librerías en programas para poder modificarlo y complementarlo. En este caso, se ha escrito una función que puede ser llamada desde cvRunHaarClassifierCascadeSum y que carga una imagen mostrando las features conforme van siendo evaluadas.

Originalmente esta función fue escrita para el caso en el que se escala la imagen en vez de la cascada, por lo que el tamaño de las features se incrementa usando el factor de escala para que puedan representarse junto con la imagen a tamaño original. La función tiene la siguiente definición:

```
void muestra_features(CvHidHaarTreeNode* node, CvPoint pt, double factor, Size
winSize)
```

Para cada rectángulo, obtengo la esquina superior izquierda y la inferior derecha. Por ejemplo:

```
esq10= Point((node->feature.rect[0]).rectangulo.x + pt.x, (node-
>feature.rect[0]).rectangulo.y + pt.y);
esq10 = esq10*factor;
esq13= Point((node->feature.rect[0]).rectangulo.x + (node-
>feature.rect[0]).rectangulo.width + pt.x, (node->feature.rect[0]).rectangulo.y +
(node->feature.rect[0]).rectangulo.height + pt.y);
esq13 = esq13*factor;
```

Después, se dibuja el rectángulo usando las esquinas y dependiendo del signo del peso, se rellena de color blanco o negro.

```
rectangle(image, esq10, esq13, Scalar(0, 0, 0), 1, 8, 0 );
rectangle(image, esq10, esq13, (node->feature.rect[0]).weight <0 ? Scalar(0, 0, 0) :
Scalar(255, 255, 255), CV_FILLED, 8, 0 );
```

Empleando esta función se ha creado un video que se adjunta en el CD que acompaña al trabajo y que muestra las features que se van evaluando. En la figura 2-6 se pueden ver dos frames del mismo.

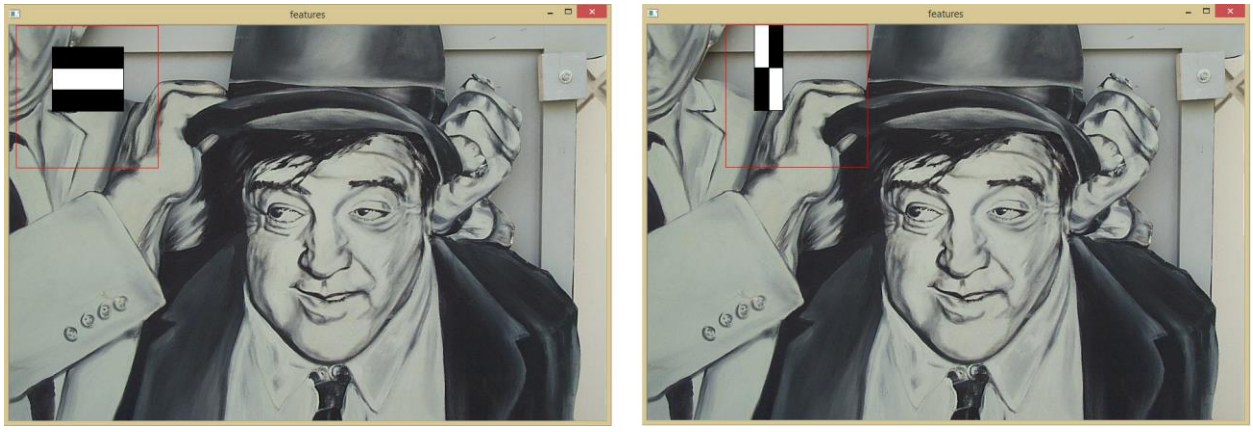

Figura 2-6. Dos features y su posición relativa a la ventana de búsqueda en la imagen "image_0403" del banco de pruebas [9].

3 DISEÑO DE UNA NUEVA PRIMERA ETAPA BASADA EN REJILLAS

Al representar las features se vio que éstas eran muy numerosas y variadas y que por tanto no resultaría posible adaptar todas ellas de manera que pudiesen ser obtenidas a partir de unas pocas rejillas de procesamiento. Por tanto, nos hemos centrado en adaptar la primera etapa que es la más discriminativa de acuerdo al procedimiento de operación del algoritmo anteriormente descrito. Solo las ventanas de búsqueda que pasen la primera etapa serán analizadas por el procesador empleando la cascada completa.

Con ello cumplimos dos objetivos. En primer lugar, comprobamos la viabilidad de emplear features obtenidas a partir del sensor. Aunque sean solo de la primera etapa, el resultado podría ser extrapolable a una cascada completa. En segundo lugar, estudiaremos la posibilidad de acelerar el algoritmo haciendo que se calcule únicamente la imagen integral de las zonas que pasan la primera etapa, que serán sobre las que el procesador tenga que calcular features.

3.1 Análisis de rendimiento sin aplicación de valores umbral

El algoritmo original emplea valores umbral para determinar si una feature se cumple o no. Como ya se ha comentado, estos valores umbrales están calculados para imágenes normalizadas y para realizar la normalización de las imágenes a analizar sería necesario calcular las imágenes cuadráticas. Esto no implementa en el sensor puesto que es costoso en términos de área y provocaría que el sensor tuviese poca sensibilidad o que la resolución del mismo fuese más baja, teniendo en cuenta los diferentes compromisos que se deben aplicar en el diseño físico de un sensor de estas características.

Por tanto, necesitamos comprobar cómo de bien funciona la primera etapa si dejamos de aplicar valores umbrales.

La parte de código que hay que modificar para dejar de usar valores umbral en la primera etapa es aquella en la que se evalúa la cascada y que forma parte de la función cvRunHaarClassifierCascadeSum.

En forma de comentario (en color verde y seguido de "//") se encuentran las líneas de código originales que han sido modificadas. Se ha sustituido el valor umbral de las features por 0, de manera que simplemente se copare el signo del contraste entre las regiones analizadas. También se han añadido unas variables "descartada" y "nodescartada" que llevarán la cuenta de ventanas descartadas o no descartadas en la primera etapa.

```
for( i = start_stage; i < cascade->count; i++ )
{
	stage_sum = 0.0;
	if (i==start_stage) //en el código original no se distingue entre la primera
etapa y el resto de etapas sino que lo que viene a continuación se emplea en todas.
	{
		for( j = 0; j < cascade->stage_classifier[i].count; j++ )
		{
CvHidHaarClassifier* classifier = cascade->stage_classifier[i].classifier + j;
CvHidHaarTreeNode* node = classifier->node;
```

```
//double t = node->threshold*variance_norm_factor;
double sum = calc_sum(node->feature.rect[0],p_offset) * node->feature.rect[0].weight;
sum += calc_sum(node->feature.rect[1],p_offset) * node->feature.rect[1].weight;
//stage_sum += classifier->alpha[sum >= t];
stage_sum += classifier->alpha[sum >= 0];
            }
            if( stage_sum < cascade->stage_classifier[i].threshold )
            {
                  descartada++;
                  return -i;
            }
            else
                  nodescartada++;

      }
```

También puede resultar interesante comprobar cómo de afectado se ve el rendimiento si además del valor umbral que sirve para determinar si una feature se cumple o no, se deja de utilizar también el valor umbral de la etapa (`cascade->stage_classifier[i].threshold` en el código). Esto es útil puesto que si al adaptar la primera etapa se cambia alguna feature, el valor umbral de la etapa resultado del entrenamiento y adaptado a las features originales dejaría de tener sentido.

Por tanto, se va a sustituir el valor umbral de la etapa por la exigencia de que se verifiquen simultáneamente las tres features que componen la etapa. El código para ello sería el siguiente:

```
double sum[3];
for( j = 0; j < 4; j++ )
{
CvHidHaarClassifier* classifier = cascade->stage_classifier[i].classifier + j;
CvHidHaarTreeNode* node = classifier->node;
sum[j] = calc_sum(node->feature.rect[0],p_offset) * node->feature.rect[0].weight;
sum[j] += calc_sum(node->feature.rect[1],p_offset) * node->feature.rect[1].weight;
if (sum[j] > 0)
      result++;
}
if(sum[0] < 0 || sum[1] < 0 || sum[2] <0)
{
      descartada++;
      return -i;
}
else
      nodescartada++;
```

La probabilidad de que una feature se cumpla en una imagen de ruido es 0.5, puesto que se está comprobando si una región es más clara o más oscura que otra región y en principio ambas opciones son equiprobables

Si las features no estuviesen correlacionadas, al exigir el cumplimiento simultáneo de tres features, la probabilidad de pasar la etapa es la opuesta a que las tres features den resultado negativo, es decir, $1 - 0.5^3 = 0.875$. Sin embargo, cabe esperar que haya cierta correlación entre las features y por tanto el porcentaje de descartes sea menor.

A continuación se muestra el programa que se ha escrito para leer y analizar las 450 imágenes del banco de pruebas.

```
int main( )
{
Mat image;
vector<Rect> faces;
ostringstream filename, destino;
int c, d, u, cero=0, una=0, varias=0, caras=450;

//se crea y se carga la cascada
CascadeClassifier face_cascade;
face_cascade.load("C:/opencv/sources/data/haarcascades/haarcascade_frontalface_alt.xml");

//para cada imagen
for(int i=1 ; i <= caras; i++)
{
	//se construye el nombre del fichero a leer y se lee
	c = i/100;
	d = (i%100)/10;
	u = (i%100)%10;

	filename.str("");
	destino.str("");
	filename << "image_0" << c << d << u << ".jpg";
	image = imread(filename.str(), CV_LOAD_IMAGE_COLOR);

	//se hace la llamada a la función de detección de caras
	face_cascade.detectMultiScale( image, faces, 1.1, 3, 0, Size(80, 80) );

	//se dibujan rectángulos en las caras detectadas
	for( int i = 0; i < faces.size(); i++ )
	{
		rectangle(image, faces[i], Scalar(0, 0, 255), 2, 8, 0 );
	}

	//se guardan las imágenes con las caras detectadas. Dependiendo del número de caras detectadas se guardan bajo el nombre “nada”, “face” o “varias” para facilitar el recuento.
	if (faces.size()==0)
	{
		destino << "nada_0" << c << d << u << ".jpg";
		cero++;
	}
	else if (faces.size()==1)
	{
		destino << "face_0" << c << d << u << ".jpg";
		una++;
	}
	else
	{
		destino << "varias_0" << c << d << u << ".jpg";
		varias++;
	}

	imwrite( destino.str(), image);
}

//se escriben los resultados en un fichero de texto
ofstream myfile;
myfile.open ("results.txt");
myfile << "Fotos sin cara detectada: " << cero << endl;
```

```
myfile << "Fotos con una cara: " << una << endl;
myfile << "Fotos con varias caras detectadas: " << varias << endl;
myfile.close();

return 0;
}
```

Con esto ya podemos comprobar cómo de afectada se ve la detección al eliminar el uso de valores umbral. Se están buscando caras con un tamaño mínimo de 80x80 para evitar detectar algunas caras secundarias más pequeñas que hay en algunas imágenes y facilitar el recuento.

	Algoritmo original	Sin usar valor umbral de features en la primera etapa	Sin usar valores umbral de features ni de etapa en la primera etapa
True positive	445	445	437
False positive	7	7	6
False negative	5	5	13
Precision	98.45	98.45	98.65
Recall	98.89	98.89	97.11
Porcentaje de ventanas descartadas en la primera etapa	28.9	19.8	66.7

Tabla 3-1. Comparación entre el rendimiento de (a) el algoritmo original, (b) una modificación del mismo en la que se deja de utilizar valor umbral para las features de la primera etapa y (c) modificación en la que, además, tampoco se utiliza valor umbral de etapa en la primera etapa.

Podemos ver que al dejar de usar valor umbral para la verificación de las features, el rendimiento en cuanto a caras detectadas no varía pero el porcentaje de ventanas de búsqueda descartadas en la primera etapa disminuye aproximádamente del 29% al 20%, lo cual no resulta interesante si buscamos una primera etapa muy discriminativa para poder ahorrar en el cálculo de la imagen integral.

Al dejar de usar valor umbral de etapa y exigir el cumplimiento simultaneo de todas las features que la componen, el número de falsos negativos no aumenta excesivamente pero se consigue aumentar el porcentaje de descartes de la primera etapa del 29% al 67%, lo cual sí resulta beneficioso.

3.2 Creación de una cara genérica

Antes de tratar de adaptar las features de la primera etapa de modo que puedan ser calculadas a partir de unas pocas rejillas, resultaría beneficioso disponer de una cara genérica que permitiese valorar de manera heurística la bondad de las modificaciones.

Para este fin se ha escrito un programa que calcula la media aritmética de las caras del banco de caras. Para no tener que localizar las caras manualmente, se emplea la versión original del algoritmo de detección de caras para que las localice. De esta manera también nos aseguramos de que las caras queden justo en la posición de la ventana de búsqueda para la cual la cascada de features que estamos empleando está entrenada.

A continuación se muestran las modificaciones realizadas sobre el programa anterior para calcular la cara genérica. En primer lugar se definen nuevas matrices y se inicializa la matriz que va a contener la suma de todas las caras. Esta matriz debe poder albergar valores de 32 bits en vez de 8 para que no desborde conforme se vayan sumando las caras.

```
Mat image, croppedFaceImage, croppedFace, face, universal;

universal = Mat::zeros(dimension, dimension, CV_32FC1);
```

En el bucle for, se sustituye el siguiente código:

```
face_cascade.detectMultiScale( image, faces, 1.1, 3, 0|CV_HAAR_SCALE_IMAGE, Size(100,
100) );


      if( faces.size() == 1 ) //cuando detecta únicamente una cara, resulta ser
siempre un true positive. De esta manera aseguramos que utilice únicamente caras para
hacer la media.
      {
            caras++;
            //se crea una imagen a partir de la región con cara detectada, se cambia
            el tamaño, se convierte a imagen en blanco y negro y se ecualiza el
            histograma para compensar las distintas condiciones de iluminación.
            resize(image(faces[0]), croppedFace, dsize, 0, 0, INTER_LINEAR );
            cvtColor( croppedFace, croppedFace, CV_BGR2GRAY );
            equalizeHist(croppedFace,croppedFace);


            //cambiar el formato y sumar a la imagen en la que se van acumulando todas
            las anteriores.
            croppedFace.convertTo(croppedFaceImage, CV_32FC1);
            universal = universal + croppedFaceImage;

    }
```

Tras sumar todas las caras se convierte la imagen a formato de 8 bits a la vez que se divide por el número de caras para hacer la media.

```
universal.convertTo(face, CV_8U, 1.0/caras);
```

Con ello se obtienen los siguientes resultados en resolución 20x20 y 50x50:

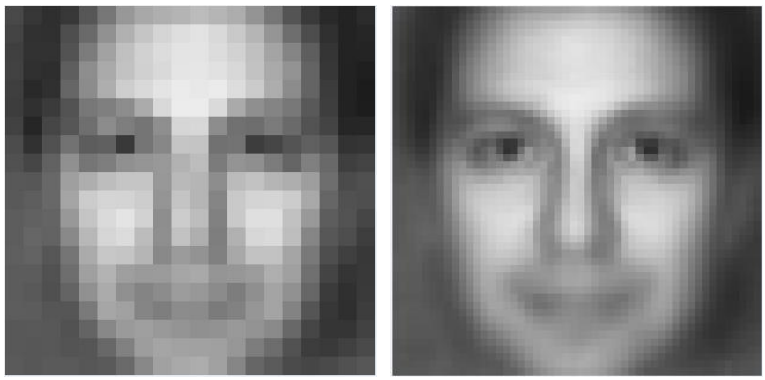

Figura 3-1. Cara genérica. La media de 440 caras en 20x20 y 50x50.

3.3 Primer diseño

Con esto ya podemos tratar de adaptar las features de la primera etapa para que puedan ser calculadas en el chip a partir de un pequeño set de rejillas como las que se muestran a modo de ejemplo en la figura 3-2.

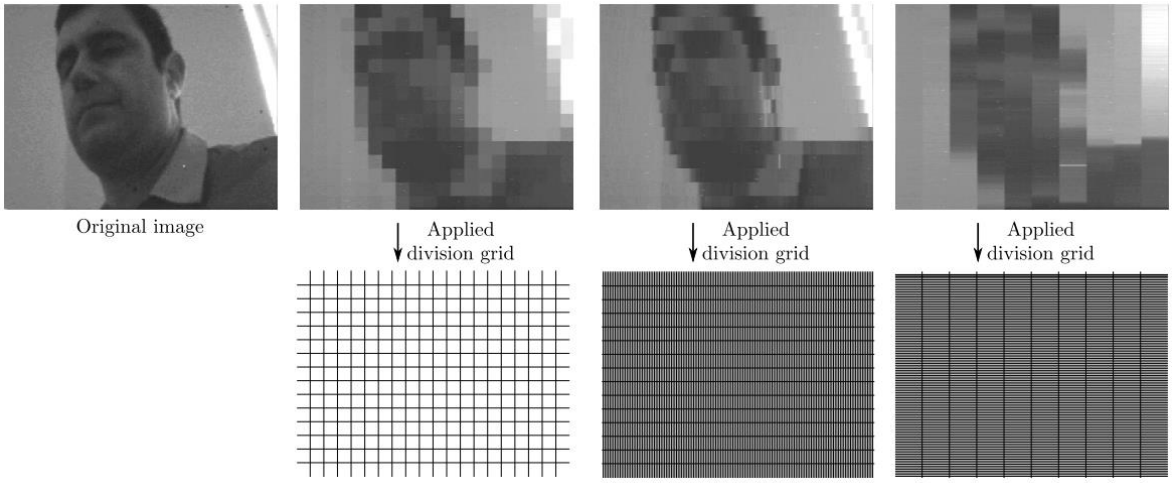

Figura 3-2. Imagen capturada por el sensor con procesamiento en el plano focal y resultado de aplicar tres rejillas diferentes para la redistribución de carga [6].

En la figura 3-3 se representan las tres features que componen la primera etapa de la cascada "haarcascade_frontalface_alt" proporcionada por OpenCV superpuestas a la imagen de la cara genérica.

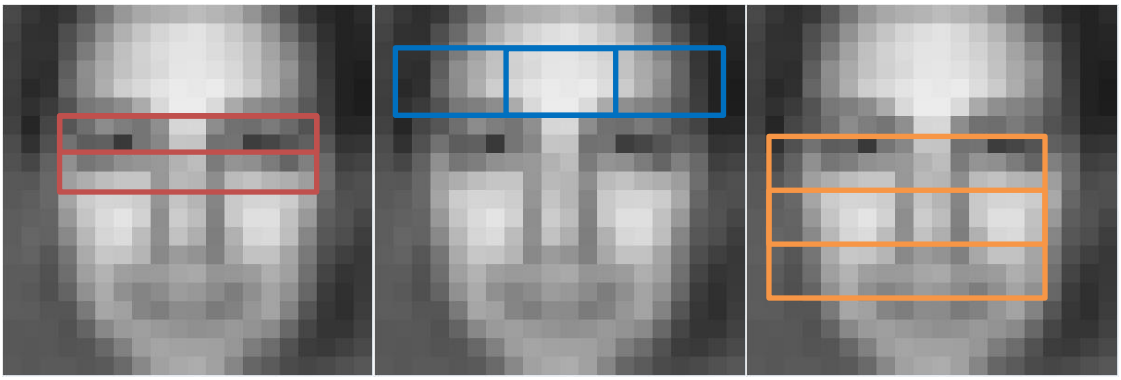

Figura 3-3. Primeras tres features de la cascada"haarcascade_frontalface_alt" sobre una cara genérica.

La primera feature se basa en que la banda de los ojos es más oscura que una banda inferior. La segunda feature hace uso de la diferencia iluminación entre el centro y los laterales de la frente. La última feature se basa en que la banda horizontal de las mejillas es más clara que otras que incluyen parte de los ojos y de la boca.

Las tres features utilizan rectángulos de dimensiones diferentes y por tanto se necesitarían varias rejillas para poder obtenerlas. Además, aunque hayan sido resultado de un proceso de entrenamiento, a simple vista parecen mejorables, especialmente la última que está desplazada hacia la izquierda lo cual no tiene sentido si se están buscando objetos con simetría según el eje central.

Analizando la imagen de la cara genérica se observa que con la rejilla mostrada en la figura 3-4 se consiguen separar bastante bien los distintos rasgos de la cara. Se han diseñado cuatro nuevas features que pueden calcularse a partir de recuadros de dicha rejilla.

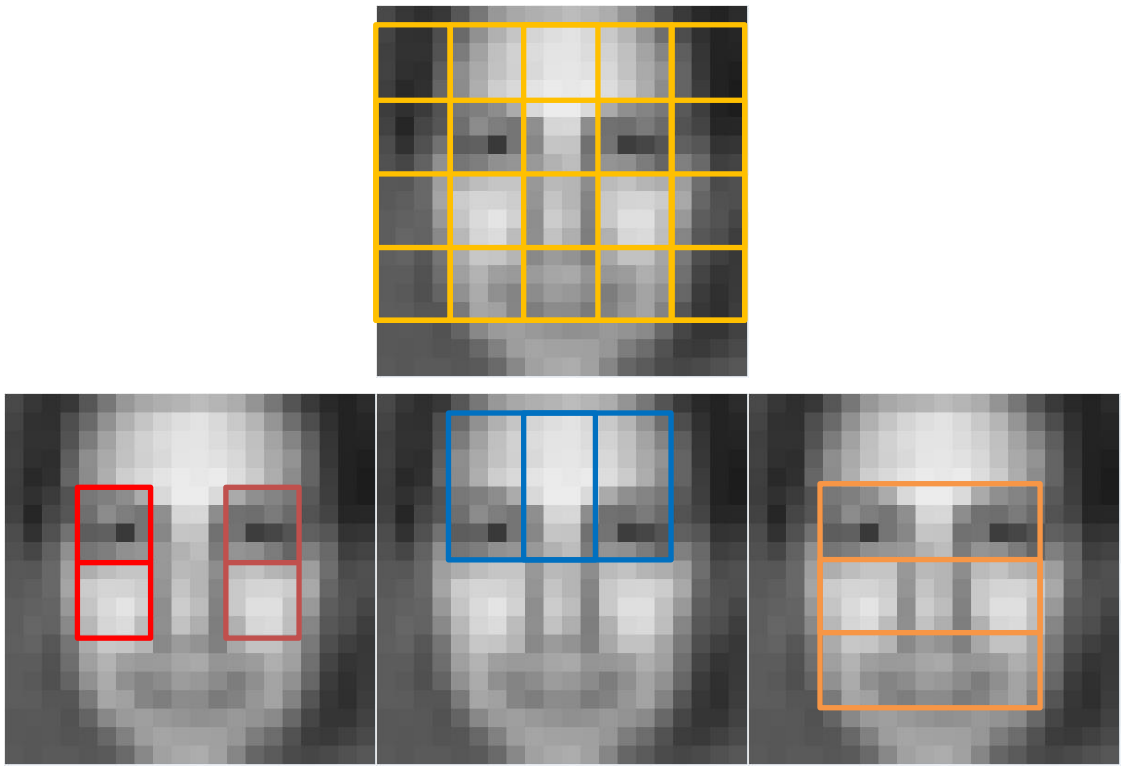

Figura 3-4. Rejilla y cuatro nuevas features en parte basadas en las features de la cascada.

Si no hubiese en absoluto correlación entre las features y se exige el cumplimiento simultáneo de las cuatro para pasar a la siguiente etapa de la cascada, la probabilidad de descartar la ventana de búsqueda en imágenes compuestas exclusivamente de ruido sería $1-0.5^4=0.9375$, lo cual resultaría muy beneficioso para nuestros objetivos. Puesto que las features se solapan, la probabilidad de descarte será algo menor.

3.4 Resultados preliminares

Para hacer una primera comprobación del rendimiento de la primera etapa diseñada, se ha desarrollado un programa que analiza imágenes con ella y realiza el barrido de las imágenes tal y como se podría realizar empleando el chip, es decir, la ventana de búsqueda se desplaza en ambas direcciones de macro-píxel en macro-píxel y los aumentos de escala se realizan fusionando 4 macro-píxeles de la escala anterior, lo que sería equivalente a emplear un factor de escala de 2.

El programa, que se basa en el código original de OpenCV y emplea alguna de sus funciones, se muestra a continuación.

```
//se define una estructura para contener las features consistente en dos rectángulos.
cada uno de los rectángulos incluye punteros a sus cuatro esquinas, peso, un
rectángulo que contendrá las medidas del rectángulo en la escala básica y otro que se
utilizará para contener las medidas del rectángulo en las diferentes escalas.
typedef struct HaarFeatures
{
    struct
    {
        sumtype *p0, *p1, *p2, *p3;
        float weight;
        Rect rectangle, rectangle0;
    }
    rect[2];
} HaarFeatures;
```

```
void findROIs(const CvArr* _img, vector <Rect> &ROIs, double scaleFactor)
{
     Size winsize, winsize0;
     winsize0.width=20;
     winsize0.height=20;

     HaarFeatures features[4];
     double evaluation[4];
     int p_offset;

     CvMat *img = (CvMat*)_img;
     Ptr<CvMat> temp, sum, sqsum;

     temp = cvCreateMat( img->rows, img->cols, CV_8UC1 );
     sum = cvCreateMat( img->rows + 1, img->cols + 1, CV_32SC1 );
     sqsum = cvCreateMat( img->rows + 1, img->cols + 1, CV_64FC1 );

     if( CV_MAT_CN(img->type) > 1 )
         {
             cvCvtColor( img, temp, CV_BGR2GRAY );
             img = temp;
         }

     //aquí se definen de manera explícita las features.
     //bandas verticales ojos y frente
     features[0].rect[0].rectangle0.x=4; features[0].rect[0].rectangle0.y=1;
     features[0].rect[0].rectangle0.width=12;
     features[0].rect[0].rectangle0.height=8; features[0].rect[0].weight=-1;
     features[0].rect[1].rectangle0.x=8; features[0].rect[1].rectangle0.y=1;
     features[0].rect[1].rectangle0.width=4;
     features[0].rect[1].rectangle0.height=8; features[0].rect[1].weight=3;
     //ojo izquierdo vertical
     features[1].rect[0].rectangle0.x=4; features[1].rect[0].rectangle0.y=5;
     features[1].rect[0].rectangle0.width=4;
     features[1].rect[0].rectangle0.height=8; features[1].rect[0].weight=-1;
     features[1].rect[1].rectangle0.x=4; features[1].rect[1].rectangle0.y=9;
     features[1].rect[1].rectangle0.width=4;
     features[1].rect[1].rectangle0.height=4; features[1].rect[1].weight=2;
     //ojo derecho vertical
     features[2].rect[0].rectangle0.x=12; features[2].rect[0].rectangle0.y=5;
     features[2].rect[0].rectangle0.width=4;
     features[2].rect[0].rectangle0.height=8; features[2].rect[0].weight=-1;
     features[2].rect[1].rectangle0.x=12; features[2].rect[1].rectangle0.y=9;
     features[2].rect[1].rectangle0.width=4;
     features[2].rect[1].rectangle0.height=4; features[2].rect[1].weight=2;
     //tres bandas horizontales ojos y boca vs mejillas
     features[3].rect[0].rectangle0.x=4; features[3].rect[0].rectangle0.y=5;
     features[3].rect[0].rectangle0.width=12;
     features[3].rect[0].rectangle0.height=12; features[3].rect[0].weight=-1;
     features[3].rect[1].rectangle0.x=4; features[3].rect[1].rectangle0.y=9;
     features[3].rect[1].rectangle0.width=12;
     features[3].rect[1].rectangle0.height=4; features[3].rect[1].weight=3;

     //se calcula la imagen integral
     cvIntegral(img, sum, sqsum);

     //para cada escala empezando a buscar en 80x80.
     for(double factor=4; ; factor *=2)
     {
      winsize.width = winsize0.width*factor;
```

```
        winsize.height = winsize0.height*factor;

        if( winsize.width > img->width || winsize.height > img->height )
              break;

        //se actualiza el tamaño de cada uno de los dos rectángulos de cada una de las
        cuatro features.
        for(int i=0; i<4; i++)
              for (int j=0; j<2; j++)
              {
features[i].rect[j].rectangle.x=features[i].rect[j].rectangle0.x*factor;
features[i].rect[j].rectangle.y=features[i].rect[j].rectangle0.y*factor;
features[i].rect[j].rectangle.width=features[i].rect[j].rectangle0.width*factor;
features[i].rect[j].rectangle.height=features[i].rect[j].rectangle0.height*factor;

features[i].rect[j].p0 = sum_elem_ptr(*sum, features[i].rect[j].rectangle.y,
features[i].rect[j].rectangle.x);
features[i].rect[j].p1 = sum_elem_ptr(*sum, features[i].rect[j].rectangle.y,
features[i].rect[j].rectangle.x+features[i].rect[j].rectangle.width);
features[i].rect[j].p2 = sum_elem_ptr(*sum,
features[i].rect[j].rectangle.y+features[i].rect[j].rectangle.height,
features[i].rect[j].rectangle.x);
features[i].rect[j].p3 = sum_elem_ptr(*sum,
features[i].rect[j].rectangle.y+features[i].rect[j].rectangle.height,
features[i].rect[j].rectangle.x+features[i].rect[j].rectangle.width);
              }


        //se realiza el barrido
        for(int y=0; y<=img->rows-winsize.height; y+=winsize.height/5)
              for(int x=0; x<=img->cols-winsize.width; x+=winsize.width/5)
              {
                    p_offset = y * img->rows + x;

                    for(int i=0; i<4; i++) //se evalúa cada feature
                    {
                          evaluation[i] = calc_sum(features[i].rect[0],p_offset) *
features[i].rect[0].weight;
                          evaluation[i] += calc_sum(features[i].rect[1],p_offset) *
features[i].rect[1].weight;
                    }

                        if(evaluation[0] < 0 || evaluation[1] < 0 || evaluation[2] <0
|| evaluation[3] <0)
                          descartada++;
                          else
                          {
                                ROIs.push_back(Rect(x, y, winsize.width,
winsize.height));Rect(x, y, winsize.width, winsize.height);
                                nodescartada++;
                          }
              }
     }
}
```

Si se aplica este programa a las imágenes del banco de pruebas, miles de ventanas de búsqueda pasan la primera etapa y habría que comprobar manualmente si las caras están entre ellas. Para evitar esto, y como primera aproximación, se han analizado las 440 caras recortadas empleadas para construir la cara genérica. Las

imágenes se analizan con una ventana de búsqueda del mismo tamaño que la imagen por lo que el número total de ventanas de búsqueda es igual al número de imágenes analizadas y el recuento es inmediato.

Para calcular el porcentaje de ventanas descartadas por la primera etapa sí se han analizado las 450 imágenes del banco de pruebas.

Con ello se obtienen los siguientes resultados:

True positive	438
False negative	2
Recall	0.9955
Porcentaje de ventanas descartadas con la primera etapa en el banco de pruebas	85.70%

Tabla 3-2. Resultados de evaluar 440 caras recortadas con la primera etapa diseñada.

Los resultados preliminares son satisfactorios; tanto el porcentaje de ventanas descartadas en la primera etapa como el recall son muy altos. En el capítulo 6 se realiza un análisis más exhaustivo de esta primera etapa en el que se integra con el resto del algoritmo de detección.

4 ESTIMACIÓN DEL POSIBLE AHORRO EN EL CÓMPUTO DE LA IMAGEN INTEGRAL

Acabamos de diseñar una primera etapa basada en features que se pueden obtener a partir de una rejilla básica y que rechaza un alto porcentaje de ventanas de búsqueda. En este capítulo se estudia la posibilidad de ahorrar en el cálculo de la imagen integral aprovechando ésta capacidad discriminativa de la primera etapa.

En primer lugar, se calcula el tiempo que tarda actualmente el programa en calcular la imagen integral. En segundo lugar, se realiza una estimación teórica del porcentaje de la imagen que pasará la primera etapa tras el barrido y por tanto el porcentaje de la imagen del cual habrá que calcular la imagen integral.

4.1 Medición del tiempo de cómputo de la imagen integral

Hemos medido el tiempo que se tarda en calcular la imagen integral y la proporción que éste supone con respecto al tiempo de ejecución del algoritmo completo para ver hasta qué punto resultaría beneficioso tratar de ahorrar en esta dirección.

Para medir el tiempo que tarda la detección de caras completa se utilizan las funciones getTickCount y getTickFrequency como se muestra a continuación:

```
double tick, time=0;
for(int i=1 ; i <= caras; i++)
{
	…

	tick = (double)getTickCount();
	face_cascade.detectMultiScale(image, faces, 1.1, 9, 0, Size(40, 40));
	time +=((double)getTickCount() – tick)
	…
}
time = time/getTickFrequency();
```

Y de manera análoga para la función que calcula la imagen integral.

Los programas se han ejecutado en un procesador Intel Core i7-3630QM con frecuencia de CPU de 2.40GHz.

En la siguiente tabla se muestran los resultados de la temporización del algoritmo completo al analizar las imágenes del banco de pruebas y el porcentaje del tiempo destinado al cálculo de la imagen integral para distintos tamaños mínimos de ventana de búsqueda. Para calcular estos tiempos se ha empleado la librería original de OpenCV “objectdetect”.

Tamaño mínimo de búsqueda	Tiempo total* (s)	Porcentaje de tiempo destinado a la imagen integral (%)
20	316.48	0.56
40	122.32	1.46
80	28.50	6.27
160	9.76	18.31

Tabla 4-1. Tiempo total en analizar las 450 imágenes del banco de pruebas y porcentaje del tiempo destinado al cálculo de la imagen integral para distintos tamaños de ventana de búsqueda mínima.

Estos resultados se representan en forma gráfica:

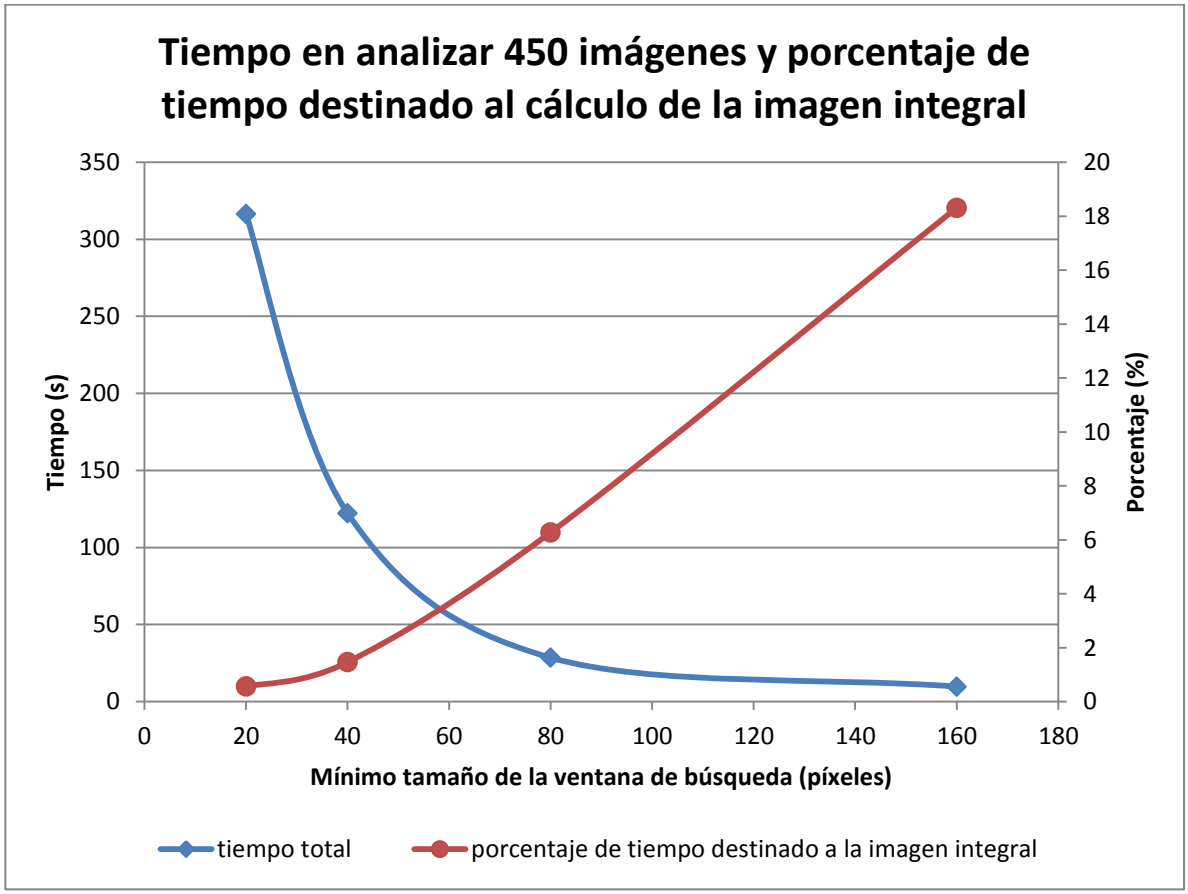

Figura 4-1. Representación gráfica del tiempo total en analizar las 450 imágenes del banco de pruebas y del porcentaje del tiempo destinado al cálculo de la imagen integral para distintos tamaños de ventana de búsqueda mínima.

Se observa que el tiempo que se tarda en calcular las 450 imágenes integrales (1.7 segundos) supone un porcentaje relativamente pequeño del tiempo total de ejecución del algoritmo, especialmente cuando la búsqueda empieza en escalas bajas.

No obstante, se ha realizado un estudio para estimar de manera aproximada la porción de imagen pasa la primera etapa y sobre la que habría que calcular la imagen integral.

4.2 Estimación de la porción de imagen integral necesaria

En la primera etapa se descarta el 85.7% de ventanas de búsqueda pero al hacer el barrido se empiezan a producir solapamientos de manera que cada píxel forma parte de varias ventanas de búsqueda. Si cualquiera de las ventanas de búsqueda de las que forma parte un píxel pasa la primera etapa, ese píxel pasará la primera etapa. Por tanto, a pesar de que se esté descartando un alto porcentaje de ventanas de búsqueda, el porcentaje de la imagen que pase la primera etapa será menor. Aquí tratamos de estimar ese porcentaje.

A modo de ejemplo se va a emplear una imagen de 40x52 píxeles. En la escala más baja, los macro-píxeles son de 4x4 píxeles, por tanto la imagen queda reducida a una matriz de10x13 macro-píxeles.

Al hacer el barrido, la ventana de búsqueda se desplaza de macro-píxel en macro-pixel.

En la figura 4-2, el número en cada macro-píxel indica el número de ventanas de búsqueda de las que forma parte tras el primer barrido. En celeste se representa la primera ventana de búsqueda.

1	2	3	4	5	5	5	5	5	4	3	2	1
2	4	6	8	10	10	10	10	10	8	6	4	2
3	6	9	12	15	15	15	15	15	12	9	6	3
4	8	12	16	20	20	20	20	20	16	12	8	4
5	10	15	20	25	25	25	25	25	20	15	10	5
5	10	15	20	25	25	25	25	25	20	15	10	5
4	8	12	16	20	20	20	20	20	16	12	8	4
3	6	9	12	15	15	15	15	15	12	9	6	3
2	4	6	8	10	10	10	10	10	8	6	4	2
1	2	3	4	5	5	5	5	5	4	3	2	1

Figura 4-2. Número de ventanas de búsqueda de las que forma parte cada macro-píxel tras el barrido en la escala más baja. En celeste se representa la primera ventana de búsqueda.

Como se puede observar, el valor de la casilla (i, j) es la multiplicación de los valores de las casillas (i, 0) y (0, j). Por tanto la tabla se puede construir a partir de la primera fila y la primera columna.

En MatLab la primera fila se puede construir de la siguiente manera:

```
ancho=52; b=4; col=1;
nx=floor(ancho/b);
for x=2:nx
   if x<=nx/2
       col=[col, min(min(5,(nx-4)),x)];
   else
       col=[col, min(min(5,(nx-4)),nx-x+1)];
   end
end
```

La primera columna puede construirse de manera análoga.

Después, para construir la matriz, se recorren todos sus elementos y se suma la multiplicación de los elementos de la primera fila y columna que correspondan, teniendo en cuenta que, como se observa en la figura 4-3, para las siguientes escalas habrá que ir de f en f macro-píxeles de la escala básica, donde f es la escala.

```
for i=1:imat
    for j=1:jmat
        if (ceil(i/f)<= length(row) && ceil(j/f) <= length(col))
            image(i,j)=image(i,j)+row(ceil(i/f))*col(ceil(j/f));
        end
    end
end
```

1	2	2	2	2	1	0
1	2	2	2	2	1	0
1	2	2	2	2	1	0
1	2	2	2	2	1	0
1	2	2	2	2	1	0

Figura 4-3. Número de ventanas de búsqueda de las que forma parte cada macro-píxel tras el barrido en la segunda escala. En celeste se representa la primera ventana de búsqueda.

Tras realizar el barrido en todas las escalas posibles (dos en este caso) el resultado es el siguiente:

2	3	5	6	7	7	7	7	7	6	5	3	1
3	5	8	10	12	12	12	12	12	10	8	5	2
4	7	11	14	17	17	17	17	17	14	11	7	3
5	9	14	18	22	22	22	22	22	18	14	9	4
6	11	17	22	27	27	27	27	27	22	17	11	5
6	11	17	22	27	27	27	27	27	22	17	11	5
5	9	14	18	22	22	22	22	22	18	14	9	4
4	7	11	14	17	17	17	17	17	14	11	7	3
3	5	8	10	12	12	12	12	12	10	8	5	2
2	3	5	6	7	7	7	7	7	6	5	3	1

Figura 4-4. Número de ventanas de búsqueda de las que forma parte cada macro-píxel de la escala más baja tras el barrido y escalado completo.

La probabilidad de que un macro-píxel de la escala básica no sea descartado (pase la primera etapa) es la probabilidad opuesta a que sea descartado en todas las búsquedas:

$$P_{pasa} = 1 - P_{descartado}^{n} \tag{4-1}$$

donde *n* es el número de ventanas de búsqueda a las que pertenece.

La porción de recuadro que pasa la primera etapa sigue una distribución de Bernoulli:

$$X \sim Be(p)$$

$$f(x;p) = \begin{cases} p_{pasa} & si\ x = 1 \\ 1 - p_{pasa} & si\ x = 0 \\ 0 & e.o.c. \end{cases} \tag{4-2}$$

Se cumple que,

$$E[X] = p_{pasa} \tag{4-3}$$

Es decir, que la porción de un recuadro que pasa la primera etapa en media es igual a la probabilidad de que el recuadro no sea descartado.

Por tanto el área total de la imagen que pasa la primera etapa es igual a:

$$Área = \sum_{n=1}^{n_{max}} (p_{pasa} \cdot num_{recuadros} \cdot área_{recuadro}) \tag{4-3}$$

En MatLab,

```
for n=1:max(max(image))
    count=nnz(image==n);
    area=area+(1-p^n)*count*b0*b0;
end
factor=area/(ancho*alto);
```

Con esto se ha realizado una gráfica (figura 4-5) que muestra la porción de imagen no descartada tras el barrido en función de la probabilidad de que una ventana de búsqueda sea descartada en la primera etapa. La gráfica se ha realizado para imágenes del tamaño de las del banco de prueba y para distintos tamaños de ventana de búsqueda mínima desde 20x20 píxeles hasta 160x160.

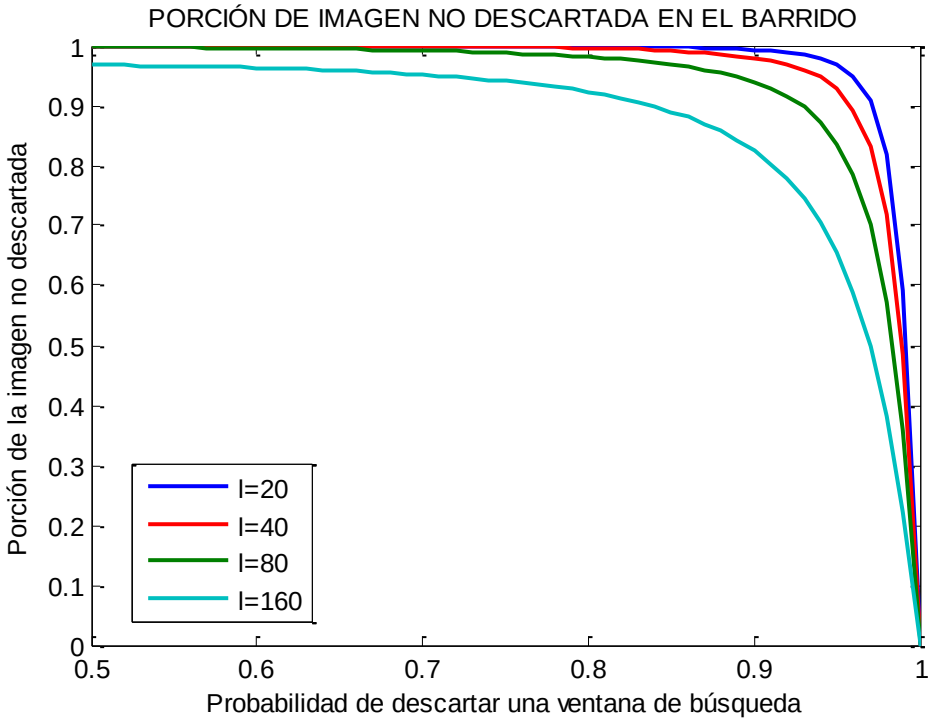

Figura 4-5. Porción de la imagen no descartada tras el barrido en función de la probabilidad de descarte de la primera etapa y calculado para distintos tamaños de la ventana de búsqueda inicial desde 20x20 hasta 160x160.

La primera etapa diseñada en el capítulo anterior tiene una probabilidad de descartar cada ventana de búsqueda del 86% que a priori parecía un valor muy positivo. Sin embargo, según los resultados mostrados en la gráfica, con ese porcentaje de descarte prácticamente el 100% de la imagen pasa la primera etapa cuando se realiza la búsqueda empezando en 20x20 o 40x40 y aproximadamente el 90% cuando se realiza empezando en 160x160.

Estos resultados indican que si se pretende ahorrar en el cálculo de la imagen integral, hace falta una primera etapa extremadamente discriminativa. Estos resultados deben observarse con cierta precaución puesto que se está suponiendo que el que una ventana de búsqueda pase la primera etapa no está correlacionado con que las ventanas vecinas también la pasen o no, y esto no es del todo así; no obstante, sirven como primera aproximación.

5 REDISEÑO DE LA PRIMERA ETAPA

En vista de los resultados anteriores y también con el propósito de seguir experimentando con features obtenidas a partir del chip, se decidió rediseñar la primera etapa para hacerla aún más discriminativa.

Para ello, se han creado y evaluado features nuevas pero antes se han vuelto a construir caras genéricas que reflejan características que no reflejaba la cara genérica anterior.

5.1 Nuevas caras genéricas

Teniendo en cuenta las características del barrido que se pretende realizar, se puede generar una cara universal que de cuenta de la variabilidad que va a haber en la posición de las caras con respecto a la ventana de búsqueda. Puesto que la rejilla, en su escala más baja, tiene recuadros de 4x4 pixeles, el peor desfase posible entre las features que se evalúan y una cara en esa escala es de 2 píxeles puesto que si es más, la cara se evaluará mejor con la siguiente ventana de búsqueda.

Para reflejar esto en la cara universal sería necesario hacer una media de las caras desplazadas aleatoriamente hasta ±2 píxeles en las direcciones vertical y horizontal. Se puede conseguir el mismo efecto haciendo la media de la cara genérica desplazada hasta ±2 píxeles, o lo que es igual, aplicar a la cara genérica un filtro de caja normalizado que emplee el siguiente kernel:

$$\frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\end{bmatrix} \tag{5-1}$$

Con las funciones de OpenCV esto se hace de la siguiente manera:

```
result.create(face.size(), face.type());
blur(face,result,Size(5,5));
```

La figura 5-1 muestra el resultado. Las regiones más distintivas son las de los ojos, la frente y las mejillas.

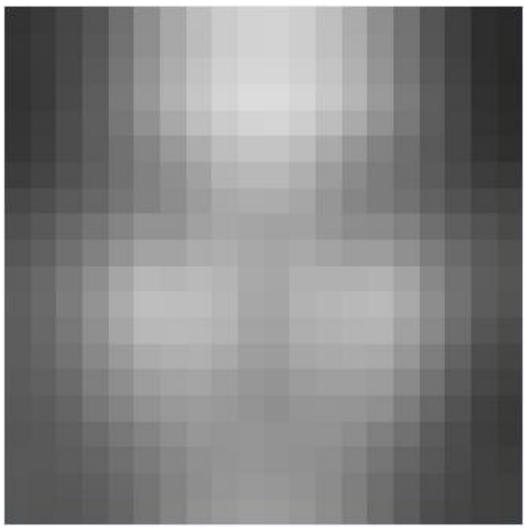

Figura 5-1. Cara genérica en la que se tiene en cuenta la desviación de hasta ±2 píxeles impuesta por la forma de realizar el barrido.

También puede ser interesante tener una imagen que dé cuenta de la variación entre los valores de los píxeles en la cara genérica comparados con los de cada una de las 440 caras que la conforman. De esta manera se podrá saber qué regiones son menos variables y por tanto mejores para la detección.

Para ello se ha escrito el pequeño programa mostrado a continuación.

```
int main()
{
	ostringstream filename;
	int c, d, u;

	Mat universal, face, dif, result, final;
	universal= imread("universal.jpg",CV_LOAD_IMAGE_GRAYSCALE);
	dif.create(universal.size(), universal.type());
	result=Mat::zeros(universal.size(), CV_32FC1);

	for(int i=1 ; i <= 450; i++)
	{
		c = i/100;
		d = (i%100)/10;
		u = (i%100)%10;

		filename.str("");
		filename << "cropped_" << c << d << u << ".jpg";

		face = imread(filename.str(),CV_LOAD_IMAGE_GRAYSCALE);

		dif=abs(universal-face);
		dif.convertTo(dif, CV_32FC1);
		result+=dif;
	}

	result.convertTo(final, CV_8U, 1.0/120);
	imwrite("final.jpg", final);
	return 0;
}
```

Como resultado se obtiene la figura 5-2 en la que un píxel más claro indica que hay mayor variación en ese píxel entre la cara genérica y cada una de las 440 caras que la componen. Se puede observar que la zona de los

ojos y de la boca presenta un mayor nivel de variación entre las distintas imágenes. La variabilidad de las zonas de los ojos se compensa con que son las zonas más oscuras de la cara, incluso en la imagen filtrada. Sin embargo, parece que la zona de la boca es poco indicada para la detección puesto que es bastante variable y además queda poco diferenciada.

Figura 5-2. Diferencia entre cada una de las 440 caras y la cara genérica. Un píxel más claro indica que las caras difieren más en ese punto.

5.2 Búsqueda de nuevas features y segundo diseño

Con la ayuda de las tres caras genéricas desarrolladas se han diseñado 11 features nuevas y se han evaluado sobre el banco de 440 caras recortadas, anotando en cuántas ocasiones no se cumplían cada una de ellas. La siguiente tabla recoge los resultados de la evaluación.

Feature	Número de veces que falla sobre 440
1. Tres bandas horizontales	0
2. Ojo izquierdo vertical	1
3. Ojo derecho vertical	1
4. Bandas horizontales ojos vs. mejillas y nariz	1
5. Banda horizontal ojos	3
6. Banda horzontal frente	9
7. Ojo derecho vs. centro frente	10
8. Ojo izquierdo vs. centro frente	13
9. Bandas horizontales mejilla y nariz vs. boca	16
10. Banda horizontal nariz	31
11. Bandas verticales nariz y boca vs. mejillas	36
12. Nariz vs. entrecejo	42
13. Bandas horizontales frente vs. ojos	43
14. Banda horizontal boca	101

Tabla 5-1. Evaluación de distintas features en 440 caras recortadas.

La nueva primera etapa debe tener más features que la anterior, pero si se exige el cumplimiento simultáneo de

un número alto de features, el número de falsos negativos aumentará. Por tanto se ha decidido exigir el cumplimiento simultáneo de las features marcadas en azul en la tabla, que tienen un índice de fallos bajo, y el cumplimiento de tres de las cuatro features marcadas en verde.

Las features que componen la primera etapa se muestran a continuación.

Cuatro features cuyo cumplimiento se exige:

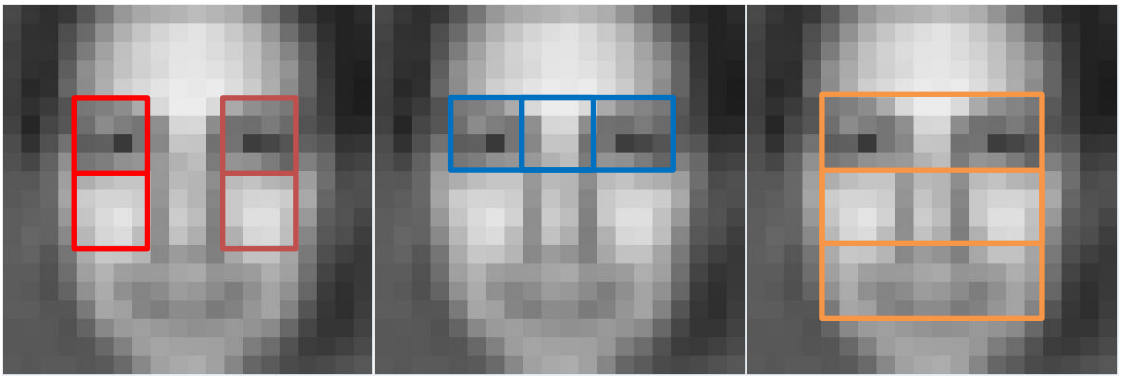

Figura 5-3. Cuatro features cuyo cumplimiento se exige. Las dos primeras y la última son de la primera etapa anterior.

Cuatro features de entre las que se deben cumplir tres:

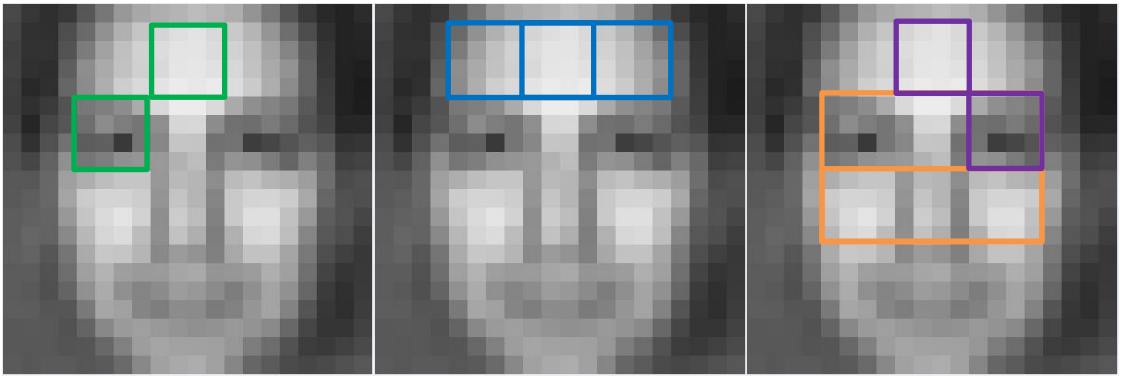

Figura 5-4. Cuatro features de entre las que se tienen que verificar tres.

Las features que comparan el ojo con el centro de la frente no son Haar-like features puesto que no se calculan a partir de rectángulos contiguos pero funcionan igual.

Suponiendo que las features no están correlacionadas, la probabilidad de que se verifiquen tres features de cuatro se puede calcular empleando las fórmulas de la distribución binomial y resulta ser 0.3125.

Por tanto, la probabilidad de descartar una ventana sería de 1 - (1 - 0.9375) * 0.3125 = 0.98.

5.3 Resultados preliminares

Se ha evaluado este segundo diseño empleando el mismo programa y en las mismas condiciones que las empleadas para el primer diseño. En la tabla se muestran los nuevos resultados comparados con los anteriores.

	Primer diseño de la primera etapa con features basadas en rejillas	**Segundo diseño de la primera etapa con features basadas en rejillas**
True positive	438	**416**
False negative	2	**24**
Recall	0.9955	**0.9455**
Porcentaje de ventanas descartadas con la primera etapa en el banco de pruebas	85.70%	**97.16%**

Tabla 5-2. Resultados de evaluar 440 caras recortadas con las dos primeras etapas diseñadas.

El número de falsos positivos ha aumentado aunque se mantiene dentro de valores razonablemente buenos. El porcentaje de ventanas descartadas por la primera etapa ha aumentado hasta casi el valor predicho.

6 EVALUACIÓN DE LAS PRIMERAS ETAPAS

Hasta ahora se han evaluado las primeras etapas empleando caras recortadas en vez de realizando un barrido sobre imágenes completas. En este capítulo se evalúan las primeras etapas realizando primero un barrido fino como hace el algoritmo original y después haciendo el barrido como se podría hacer con el chip.

6.1 Barrido fino

En primer lugar se analiza el resultado de sustituir las features originales de la primera etapa por las diseñadas en este trabajo pero manteniendo el tipo de barrido fino del algoritmo original. Es decir, este barrido no tiene en cuenta el tipo de información que el chip podría proporcionar para acelerar el procesamiento.

Por un lado, se pretende comprobar que los resultados preliminares en cuanto a caras detectadas por las rejillas en imágenes de caras recortadas se mantienen cuando se hace el barrido fino. Esto nos dará una idea de la cota superior de detecciones que cabe esperar cuando se haga un barrido más grueso y permitirá evaluar la bondad de éste. Por otra parte, se cuantificará el rendimiento y el ahorro computacional que supone sustituir en el algoritmo original la primera etapa por las diseñadas en este trabajo que son más discriminativas.

Para ello, se modifica el fichero XML de la cascada sustituyendo las features originales de la primera etapa por las nuevas. A continuación se muestra el inicio del fichero de la cascada con dos nuevas features. Los valores umbrales y "left_val" y "right_val" pueden dejarse tal y como estaban puesto que no se usan en las primeras etapas diseñadas.

```
<opencv_storage>
<haarcascade_frontalface_alt type_id="opencv-haar-classifier">
  <size>20 20</size>
  <stages>
    <_>
      <!-- stage 0 -->
      <trees>
        <_>
          <!-- tree 0 -->
          <_>
            <!-- root node -->
            <feature>
              <rects>
                <_>4 1 12 8 -1.</_>
                <_>8 1 4 8 3.</_></rects>
              <tilted>0</tilted></feature>
            <threshold>4.0141958743333817e-003</threshold>
            <left_val>0.0337941907346249</left_val>
            <right_val>0.8378106951713562</right_val></_></_>
        <_>
          <!-- tree 1 -->
```

```
<_>
  <!-- root node -->
  <feature>
    <rects>
      <_>4 5 4 8 -1.</_>
      <_>4 9 4 4 2.</_></rects>
    <tilted>0</tilted></feature>
  <threshold>0.0151513395830989</threshold>
  <left_val>0.1514132022857666</left_val>
  <right_val>0.7488812208175659</right_val></_></_>
<_>
```

Los resultados son los siguientes. En los tres primeros casos se ha llamado a la función detectMultiScale con un valor de *minNeighbours* igual a 8 mientras que en el último caso se ha utilizado 4 aumentando el número de falsos positivos para perder menos verdaderos positivos.

	Algoritmo original	Features originales sin valores umbral en la primera etapa	Primer diseño de la primera etapa con features basadas en rejillas	Segundo diseño de la primera etapa con features basadas en rejillas
True positive	445	437	442	427
False positive	7	6	3	10
False negative	5	13	8	23
Precision	98.45	98.65	99.32	97.71
Recall	98.89	97.11	98.22	94.89
Porcentaje descartado en la primera etapa	28.9	66.7	74.9	93.7
Área analizada (píxeles)	$6.01 \cdot 10^{11}$	$4.55 \cdot 10^{11}$	$4.38 \cdot 10^{11}$	$4.11 \cdot 10^{11}$
Tiempo de cómputo (s)	86.3	41.9	32.8	21.0

Tabla 6-1. Comparación del algoritmo original con las features originales y del algoritmo modificado en el que no se aplican valores umbral en la primera etapa con las features originales y con las features de las dos primeras etapas diseñadas.

Los resultados son bastante buenos y coinciden aproximadamente con los resultados preliminares obtenidos en las imágenes de caras recortadas.

Empleando el primer diseño de la segunda etapa se consigue un ahorro en el tiempo de cómputo del 38.0% con respecto al algoritmo original mientras que el recall disminuye ligeramente del 98.9% a 98.2%.

Utilizando el segundo diseño, se consigue un ahorro en el tiempo de cómputo del 75.7% con respecto al algoritmo original mientras que el recall pasa a ser 94.9%.

6.2 Barrido basado en el procesamiento en el plano focal

En principio, al hacer el barrido empleando features obtenidas a partir del chip, la ventana de búsqueda se moverá de macro-píxel en macro-píxel y aumentará de escala con un factor de 2.

En este apartado se pretende cuantificar cómo de afectada se ve la primera etapa al realizar el barrido este

barrido más grueso.

Medir esto no es tan fácil como en el caso anterior puesto que si enlazamos la primera etapa con el resto de la cascada como se ha hecho para el caso del barrido fino, no podremos saber si los fallos en la detección se deben a realizar el barrido grueso con la primera etapa o están relacionados con el resto de la cascada. Si, por el contrario, analizamos la imagen únicamente con la primera etapa, se obtendrían miles de ventanas de búsqueda que habría que revisar manualmente.

La solución que se ha adoptado consiste en cambiar el tamaño y la posición de las caras con respecto a una ventana de búsqueda fija en lugar de mover la ventana de búsqueda sobre una imagen fija. Es decir, se han creado imágenes de tamaño fijo en las que la búsqueda se realiza al mismo tamaño de la imagen y en las que se incluyen caras a distintas escalas y en distintas posiciones. De esta manera resulta inmediato contar el número de aciertos y fallos en la detección.

Para que esto sea representativo de la búsqueda completa, las caras tienen que tener los tamaños que se detectarían a esa escala. Como la búsqueda se realizaría a las escalas 20x20, 40x40, 80x80…, las caras que se detectarían con la ventana de búsqueda de 40x40 serán las que tengan tamaños entre 30x30 y 60x60. Por tanto, las caras para el banco de pruebas tendrán un tamaño aleatorio acotado por esos dos valores.

En cuanto al desfase entre la rejilla y las caras, la peor posibilidad sería medio macropíxel, es decir, 4 píxeles en la escala de 40x40. Por tanto las caras estarán descentradas hasta ±4 píxeles de manera aleatoria.

Esto se correspondería a realizar el barrido grueso que se ha contemplado hasta ahora. Sin embargo, podemos analizar otras posibilidades para mejorar el rendimiento de la detección. Por ejemplo, se puede realizar una segunda búsqueda a escalas intermedias (30x30, 60x60, 120x120...) en un siguiente frame.

Incluyendo este tipo de escalado intermedio, las caras que se corresponderían a la escala de 40x40 tendrían tamaños entre 35 y 50.

Otra posible mejora puede consistir en emplear rejillas más finas. Utilizando una rejilla el doble de fina, los macro-píxeles en la escala más básica serían de 2x2 y para obtener los macro-píxeles originales con los que se calculan las features el ordenador tendría que sumar 4 macro-píxeles nuevos. La figura 6-1 representa, para las dos primeras escalas, los macro-píxeles originales (en color) compuestos por 4 macro-píxeles nuevos y cómo esto permite realizar un barrido más suave en las direcciones vertical y horizontal.

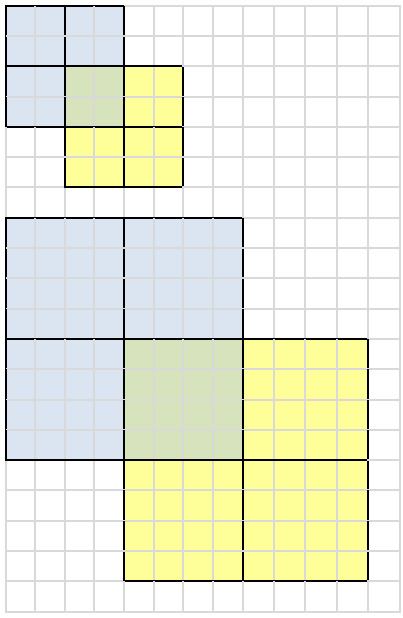

Figura 6-1. Macro-píxeles originales en las dos primeras escalas empleando rejillas el doble de finas. Cada macro-píxel original se compone de 4 macro-píxeles nuevos, lo que permite realizar un barrido más fino.

Para estimar los resultados que se podrían obtener empleando esta rejilla más fina, las caras para el banco de pruebas estarían desplazadas hasta ±1 píxeles en vez de ±2 en la escala básica o ±2 en vez de ±4 en 40x40.

A continuación se muestra el programa que construye estas imágenes.

```
int main( )
{
     Mat image, resized, final;
     ostringstream filename, destino_cropped;
     int c, d, u, imagines=450, sizemin, sizemax, sizemid, size;
     int shiftcols, shiftrows;
     double scale;

     CascadeClassifier face_cascade;
     face_cascade.load(
"C:/opencv/sources/data/haarcascades/haarcascade_frontalface_alt.xml" );
     std::vector<Rect> faces;

     sizemin=30;
     sizemax=60;
     sizemid=40;

     srand(time(0));

     for(int i=1 ; i <= 450; i++)
     {
      c = i/100;
      d = (i%100)/10;
      u = (i%100)%10;

      filename.str("");
      destino_cropped.str("");
      filename << "image_0" << c << d << u << ".jpg";

      size = (rand() % (sizemax-sizemin) + sizemin);
      shiftcols = rand() % (sizemid/5+1) -sizemid/10;
      shiftrows = rand() % (sizemid/5+1) -sizemid/10;

      image = imread(filename.str(), CV_LOAD_IMAGE_COLOR);
      face_cascade.detectMultiScale( image, faces, 1.1, 3, 0|CV_HAAR_SCALE_IMAGE,
Size(100, 100) );

      if( faces.size() == 1 )
      {
            Rect ROI, ROI2;

            scale=(double)faces[0].width/size;

            cvtColor( image, image, CV_BGR2GRAY );
            resize(image, resized, Size(0,0), 1/scale, 1/scale, INTER_LINEAR );

            ROI.x = faces[0].x/scale;
            ROI.y = faces[0].y/scale;
            ROI.width=faces[0].width/scale;
            ROI.height=faces[0].height/scale;
```

```
            ROI2.x=ROI.x+ROI.width/2+shiftcols-sizemid/2;
            ROI2.y=ROI.y+ROI.height/2+shiftrows-sizemid/2;
            ROI2.width=sizemid;
            ROI2.height=sizemid;

            Mat final;
            final.create(Size(sizemid,sizemid), resized.type());
            final=resized(ROI2);
            destino_cropped << "descentrada_" << c << d << u << ".jpg";
            imwrite( destino_cropped.str(), final);
        }
       }
    return 0;
}
```

Con el programa se obtiene este tipo de imágenes:

Figura 6-2. Ejemplo de caras a distintas escalas y descentradas.

A continuación se muestran las estimaciones para distintos tipos de escalado y de barrido. El escalado y barrido fino serían los originales del algoritmo, lo cual se traduce en que las caras recortadas tienen el mismo tamaño que la ventana de búsqueda y están centradas. El escalado grueso corresponde a realizar la búsqueda duplicando la escala mientras que en el escalado intermedio se realizaría otra búsqueda a escalas intermedias. En el barrido grueso se emplea una rejilla básica de 4x4 píxeles y en el barrido intermedio se utiliza una rejilla básica de 2x2 píxeles lo que permite localizar caras en posiciones intermedias con respecto a las que permite la rejilla de 4x4.

		Caras recortadas	
Tipo de escalado	Tipo de barrido	True positive sobre 440	Recall (%)
Fino	Fino	416	94.6
Fino	Intermedio	336	76.4
Fino	Grueso	256	58.2
Intermedio	Fino	400	91.0
Intermedio	**Intermedio**	**337**	**76.6**
Intermedio	Grueso	241	54.8
Grueso	Fino	385	87.5
Grueso	Intermedio	326	74.1
Grueso	Grueso	233	53.0

Tabla 6-2. Evaluación de la segunda primera etapa diseñada.

Realizando un escalado intermedio (por lo que la búsqueda se realizaría en 2 frames) y usando la rejilla de partida de 2x2 píxeles se conseguiría un recall del 76.6%.

Se puede observar también que la primera etapa es más sensible a los desplazamientos en las caras que al escalado.

7 UNIÓN DE LA PRIMERA ETAPA CON EL RESTO DEL ALGORITMO

Finalmente, para concluir el trabajo se ha integrado el segundo diseño de la primera etapa con el resto de la cascada.

Primero se probó a integrar con el resto del algoritmo el programa mostrado en el capítulo 3 que realiza el barrido con la primera etapa. Esto fue posible pero los tiempos de procesado eran excesivamente altos.

Por tanto se optó por añadir las features al fichero de la cascada como se hizo para el barrido fino en el capítulo anterior y modificar el programa original para conseguir que haga el barrido equivalente al que se haría con el tipo de información proporcionada por el chip.

Para ello, en la función cvHaarDetectObjectsForROC se han cambiado el incremento del factor de escala y la variable *ystep* que será el paso con el que se hace el barrido tanto en el eje y como en el eje x.

Para que el factor de escala vaya aumentando según la progresión: 4, 6, 8, 12, 16, 24, 32… que se corresponde a realizar un proceso de escalado a escalas intermedias en otro frame, hay que multiplicar por 4/3 si el factor de escala anterior es múltiplo de 3 o por 3/2 en caso contrario.

Para que el barrido se realice de macro-píxel en macro-píxel, la variable *ystep* debe tomar el valor 4*factor, mientras que si se utiliza una rejilla de la mitad de tamaño, la variable *ystep* valdría 2*factor.

```
//for( ; n_factors-- > 0; factor *= scaleFactor )
for(factor=4; factor*cascade->orig_window_size.width < img->cols - 10;
(int)factor%3==0?factor=factor*4/3:factor=factor*3/2)
        {
        //const double ystep = std::max( 2., factor );
        const double ystep = 2*factor;
```

Para el barrido, se cambia la función HaarDectectOBjects_ScaleCascade_Invoker. Esta función es más difícil de modificar puesto que forma parte de una clase que está definida en otro lugar, por lo que se han reutilizado las variables originales.

Originalmente la función realiza el barrido en el eje y según valor *ystep* y el barrido en el eje x de píxel en píxel o de dos píxeles en dos píxeles según se haya encontrado una cara en la ventana de búsqueda anterior o no. Se ha reaprovechado la variable *ystep* para el salto en ambas direcciones y *xrange.start* para lo que debería haberse llamado *yrange.end*.

```
class HaarDetectObjects_ScaleCascade_Invoker : public ParallelLoopBody
{
public:
    HaarDetectObjects_ScaleCascade_Invoker( const CvHaarClassifierCascade* _cascade,
Size _winsize, const Range& _xrange, double _ystep, size_t _sumstep, const int** _p,
const int** _pq, std::vector<Rect>& _vec, Mutex* _mtx)
    {
```

```
        cascade = _cascade;
        winsize = _winsize;
        xrange = _xrange;
        ystep = _ystep;
        sumstep = _sumstep;
        p = _p; pq = _pq;
        vec = &_vec;
        mtx = _mtx;
    }
    void operator()( const Range& range ) const
    {
        for( int y = 0; y < xrange.start; y+=ystep )
        {
            for( int x = 0; x < xrange.end; x += ystep )
            {
             area+=winsize.width*winsize.height;
             int result = cvRunHaarClassifierCascade( cascade, cvPoint(x, y), 0);
             if( result > 0 )
             {
                    mtx->lock();
                    vec->push_back(Rect(x, y, winsize.width, winsize.height));
                    vec->push_back(Rect(x, y, winsize.width, winsize.height));
                    mtx->unlock();
             }
            }
        }
    }
    const CvHaarClassifierCascade* cascade;
    double ystep;
    size_t sumstep;
    Size winsize;
    Range xrange;
    const int** p;
    const int** pq;
    std::vector<Rect>* vec;
    Mutex* mtx;
};

}
```

Con esto ya se consigue realizar un barrido equivalente al que se podría realizar a partir de las representaciones de la imagen proporcionadas por el chip. Sin embargo, es necesario cambiar también la manera en que se agrupan los rectángulos correspondientes a detecciones múltiples.

Por un lado, como se puede observar en la figura 7-1, los rectángulos están más separados entre sí. Por otro, hay bastantes caras en las que solo se consigue una detección.

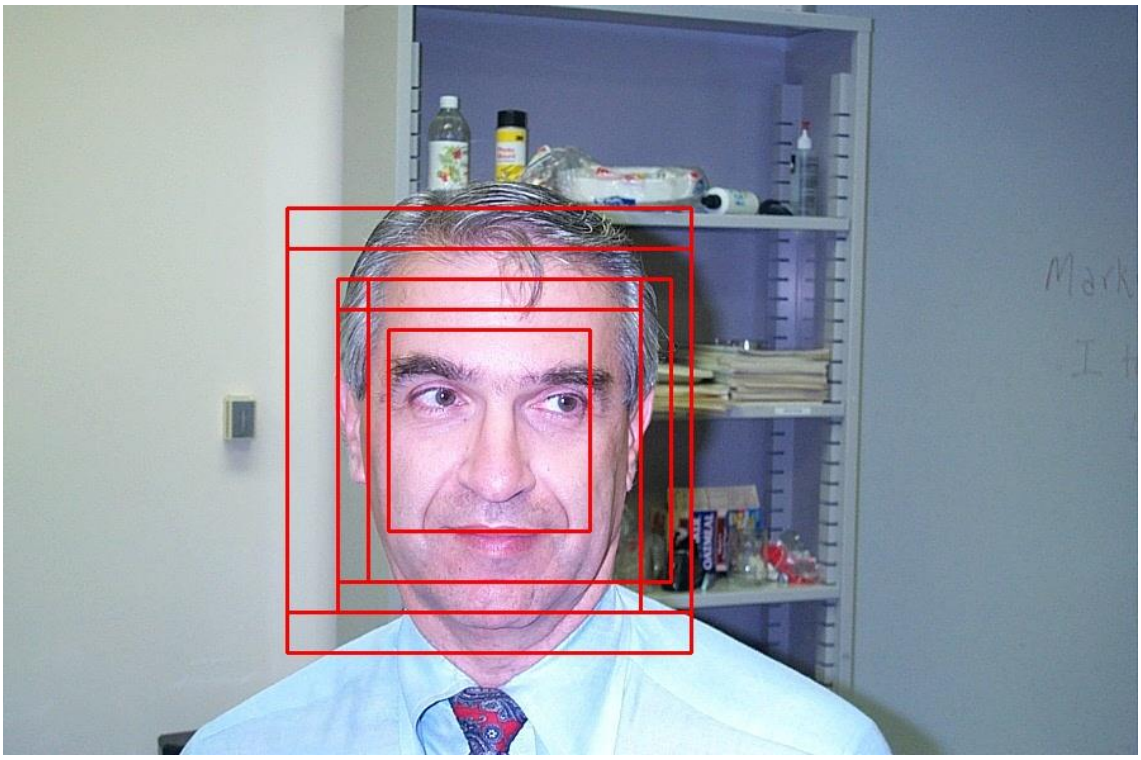

Figura 7-1. Detección de caras en la imagen "image_0096". Se observan varias detecciones para la misma cara.

Para que rectángulos que están más separados sigan fusionándose, se aumenta el parámetro GROUP_EPS de 0.2 a 0.6 y para que pueda seguir usándose la función groupRectangles con *minNeighbours* = 1 sin que pierda las detecciones únicas, se duplican todos los rectángulos detectados.

```
int result = cvRunHaarClassifierCascade( cascade, cvPoint(x, y), 0);
if( result > 0 )
{
	mtx->lock();
	vec->push_back(Rect(x, y, winsize.width, winsize.height));
	vec->push_back(Rect(x, y, winsize.width, winsize.height));
	mtx->unlock();
}
```

Buscando caras de tamaño mínimo 80x80 se obtiene los siguientes resultados para distintos tipos de escalado y de barrido que muestran junto con las estimaciones del capítulo anterior.

		Primera etapa+caras recortadas	Algoritmo completo					
Tipo de escalado	Tipo de barrido	Recall (%)	True positive /450	False positive	Recall (%)	Precision (%)	Área (píxeles)	Tiempo (s)
Fino	Fino	94.6	433	14	96.2	96.9	$7.09 \cdot 10^{11}$	31.6
Fino	Intermedio	76.4						
Fino	Grueso	58.2						
Intermedio	Fino	91.0						
Intermedio	**Intermedio**	**76.6**	**426**	**12**	**94.7**	**97.2**	$\mathbf{7.41 \cdot 10^{10}}$	**5.7**
Intermedio	Grueso	54.8	319	4	70.9	98.8	$1.85 \cdot 10^{10}$	4.3
Grueso	Fino	87.5						
Grueso	Intermedio	74.1	420	8	93.3	98.1	$4.04 \cdot 10^{10}$	4.5
Grueso	Grueso	53.0	207	0	46.0	100	$1.04 \cdot 10^{10}$	3.4

Tabla 7-1. Resultado de los distintos tipos de escalado y barrido y comparación con las estimaciones obtenidas a partir de las caras recortadas.

En el escalado y barrido fino, el recall ha aumentado ligeramente con respecto al presentado anteriormente debido a que ahora se están salvando las detecciones únicas que antes se perdían.

En el escalado y barrido intermedio que podría realizarse con el chip, el recall ha mejorado notablemente con respecto a la estimación, alcanzando un valor muy similar al obtenido con el barrido fino. Esto parece deberse al hecho de que las caras pueden detectarse en varias escalas y no únicamente en la escala más cercana como se había supuesto al realizar la estimación.

De nuevo se observa que el tipo de barrido afecta más que el tipo de escalado.

Por último revisamos el tema del posible ahorro en la imagen integral. Según los resultados teóricos, para una primera etapa con un porcentaje de descartes del 97% y para un tamaño mínimo de búsqueda de 80x80, aproximadamente un 70% de la imagen pasaría la primera etapa.

Ahí no se estaba teniendo en cuenta la posibilidad de hacer escalados o barridos intermedios. La figura 7-2 muestra el resultado de analizar una imagen con la primera etapa con escalado y barrido intermedios. Como se puede ver, prácticamente la totalidad de la imagen se utiliza por lo que no resulta interesante tratar de ahorrar en el cálculo de la imagen integral.

Figura 7-2. Imagen "image_0086" analizada con la primera etapa. Los cuadrados rojos son las ventanas que pasan la primera etapa y prácticamente llenan la imagen entera a pesar de que se ha descartado el 97.1% de las ventanas de búsqueda.

8 CONCLUSIONES

8.1 Resumen y conclusiones

En primer lugar se observó que la cascada facilitada por OpenCV está compuesta por un gran número de features que emplean rectángulos de tamaños, formas y posiciones muy variadas. Esto hace que no resulte posible adaptar la cascada de features entera de modo que éstas puedan ser obtenidas a partir del tipo de representaciones de imagen proporcionadas por el chip empleando unas pocas rejillas de procesamiento.

Por tanto, nos centramos en adaptar únicamente la primera etapa, que es la más discriminativa. Con esto pretendíamos estudiar el uso de features obtenidas a partir del chip y tratar de ahorrar en el cálculo de la imagen integral.

Para esto, primero evaluamos el rendimiento del detector al dejar de utilizar valores umbral en la primera etapa puesto que los valores umbrales están calculados para imágenes normalizadas y para la normalización se requiere una imagen integral cuadrática que no va a ser facilitada por el chip.

Los resultados fueron razonablemente buenos. Al dejar de usar valores umbral en la primera etapa la sensibilidad o recall disminuyó ligeramente de 98.9% a 97.1% pero el porcentaje de ventanas descartadas en la primera etapa aumentó del 28.9% al 66.7%.

A continuación se trató de adaptar las tres features que componen la primera etapa original con la ayuda de una cara genérica. Se consiguió diseñar una rejilla a partir de la cual obtener dos features parecidas a las originales, y otras dos completamente nuevas. La sensibilidad de esta primera etapa evaluada sobre 440 caras recortadas fue extremadamente alta, del 99.6%, rechazando el 85.7% de las ventanas de búsqueda.

Alentados por estos resultados se buscó determinar el posible ahorro en el cálculo de la imagen integral que supondría emplear esta primera etapa. Por un lado se vio que el tiempo que tarda el algoritmo original en calcular las imágenes integrales es bastante reducido (0.6% del tiempo total cuando se empiezan a buscar caras en 20x20, o 6% si se comienza la búsqueda en 80x80). Por otro lado, se realizó una estimación teórica de la porción de imagen que pasaría la primera etapa tras el barrido y por tanto la porción de imagen sobre la cual habría que calcular la imagen integral. Se llegó a la conclusión, poco intuitiva, de que con un porcentaje de descarte de ventanas de búsqueda del 85% prácticamente el 100% de la imagen pasaría la primera etapa.

Por tanto se decidió rediseñar la primera etapa para hacerla aún más discriminativa y a la vez seguir evaluando features obtenidas a partir del chip.

De nuevo con la ayuda de caras genéricas, se diseñó una primera etapa consistente en 8 features que presentaba una sensibilidad del 94.6% en las 440 caras recortadas descartando el 97.2% de las ventanas de búsqueda.

El siguiente paso fue evaluar las primeras etapas, no sobre caras recortadas sino haciendo un barrido y escalado de la ventana de búsqueda.

Para el barrido y escalado fino, se sustituyeron las features nuevas en el fichero de la cascada y ejecutó el algoritmo original modificado para que no utilizase valores umbral en la primera etapa. Empleando el primer diseño de la segunda etapa se consigue un ahorro en el tiempo de cómputo del 38.0% mientras que el recall disminuye del 98.9% a 98.2%. Con el segundo diseño, se consigue una sensibilidad del 94.9%, ligeramente superior que la obtenida con las caras recortadas y gracias a su alta capacidad discriminativa, se consigue un ahorro en el tiempo de procesado del 75.7%. Estos resultados están desligados del chip pero pueden resultar interesantes por sí mismos.

Para el barrido y escalado más grueso que se podría realizar con la información proporcionada por el chip, se hizo una estimación del rendimiento en la detección a partir de recortes de caras de distintos tamaños y posiciones. Se exploró la posibilidad de realizar escalados y barridos intermedios y combinando estas dos mejoras se estimó que la primera etapa tendría una sensibilidad del 76.6%.

Por último, se integró el segundo diseño de la primera etapa con el resto de la cascada realizando escalados y

barridos como los que se harían empleando el chip y el resultado fue bastante mejor que el estimado. Realizando un escalado a escalas intermedias, y utilizando una rejilla de partida de 2x2 píxeles para realizar el barrido intermedio, se consiguió una sensibilidad del 94.7% y un ahorro en el tiempo de procesado del algoritmo original del 93.3%.

A pesar de ello, sigue sin ser posible ahorrar en el cálculo de la imagen integral puesto que a pesar de que la primera etapa es muy discriminativa, tras el barrido prácticamente la totalidad de la imagen pasa la etapa.

Por tanto, en este trabajo se ha visto que no resulta interesante tratar de combinar una primera etapa calculada a partir de features obtenidas con el chip si después se depende de la imagen integral para calcular el resto de la cascada en las ventanas que pasen la primera etapa.

No obstante, se ha comprobado que resulta posible conseguir un buen rendimiento en la detección empleando features obtenidas a partir de rejillas con una escala de pixelación relativamente gruesa y que el rendimiento no empeora al realizar el barrido si se realiza un escalado extra a escalas intermedias y la ventana de búsqueda se desplaza con incrementos del tamaño de medio macro-píxel de los empleados para definir las features.

8.2 Futuras investigaciones

Los resultados del trabajo indican que es posible emplear features obtenidas a partir del chip pero que para aprovechar la mejora en la eficiencia que esto supone, sería conveniente crear una cascada completa de features obtenidas a partir del tipo de representaciones de imagen proporcionadas por el chip.

Por tanto sería necesario realizar un proceso de entrenamiento que seleccione features obtenidas a partir de rejillas como las que se han empleado en este trabajo y construya la cascada.

Al realizar el escalado y el barrido grueso la detección es más difícil y además contamos con menos posibles combinaciones de macro-píxeles para construir las features. Por tanto sería conveniente disponer de un set de features más flexibles que no estén limitadas a emplear rectángulos contiguos como las Haar-like features utilizadas por Viola-Jones.

En este trabajo ya se han empleado dos features de este tipo más flexible que comparaban la región del ojo con el centro de la frente. Estas features ya han sido estudiadas y se conocen como features distribuidas en el espacio granular, propuestas originalmente por Huang, Ai, Li y Lao [17] quienes emplean features obtenidas a partir de cuadrados con lado múltiplo de 2 para mejorar la eficiencia computacional (figura 8-1) consiguiendo un buen rendimiento en la detección.

Puesto que este tipo de features son más flexibles, existe un mayor número de posibilidades y son más difíciles de escoger empleando métodos que símplemente analicen todas las posibilidades. En [18] se propone un método para buscar features distribuidas en el espacio granular basado en algoritmos genéticos.

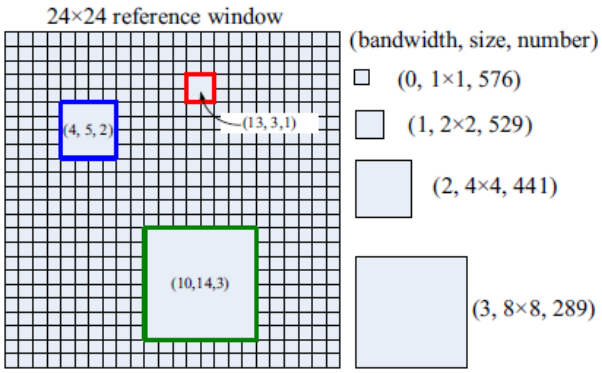

Figura 8-1. Cuadrados constituyentes de las features distribuidas en el espacio granular [16].

Al obtener las features a partir del chip se pierde flexibilidad en cuanto a la localización de las mismas por lo

que podría resultar necesario aumentar la flexibilidad en cuanto a las escalas de búsqueda con respecto a la restricción impuesta en [17]. Para ello se podrían emplear rejillas con escalas intermedias en frames sucesivos como se ha propuesto en este trabajo.

Para poder realizar barridos menos gruesos, los macro-píxeles deben ser más pequeños y el procesador debe sumar varios para calcular el valor de las features. Por tanto, podría resultar interesante utilizar features más pequeñas de modo que no fuese necesario sumar un gran número de macro-píxeles. En este sentido cobra relevancia el trabajo de Abramson y Steux [19], que utiliza únicamente comparaciones de signo entre píxeles denominados puntos de control y que se muestran a continuación:

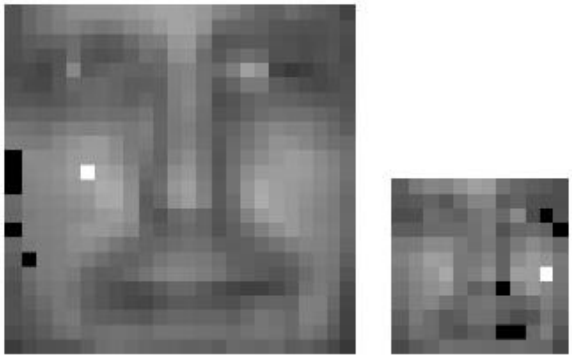

Figura 8-2. Puntos de control en resoluciones 24x24 y 12x12. La feature se verifica si los píxeles marcados en blanco tienen valores mayores que todos los marcados en negro [19].

Las features obtenidas a partir del chip podrían ser de un tipo intermedio entre las features distribuidas en el espacio granular y los puntos de control de Abramson.

REFERENCIAS

[1] Russakovsky, O., *et al* (2015). ImageNet Large Scale Visual Recognition Challenge. Int. J. of Computer Vision. DOI: 10.1007/s11263-015-0816-y.

[2] Karpathy, A., & Fei-Fei, L. (2015). Deep Visual-Semantic Alignments for Generating Image Descriptions. CVPR 2015.

[3] Felleman, D., & Essen, D. (1991). Distributed Hierarchical Processing in the Primate Cerebral Cortex. Cerebral Cortex, 1 (1), 1-47.

[4] Viola, P., & Jones, M. (2001). Robust real-time face detection. Proceedings Eighth IEEE International Conference on Computer Vision. ICCV.

[5] Gonzalez, R., & Woods, R. (2002). Digital Image Processing. Upper Saddle River, N.J.: Pearson Prentice Hall.

[6] Fernández-Berni, J., Carmona-Galán, R., Río, R., Kleihorst, R., Philips, W., & Rodríguez-Vázquez, Á. (2015). Focal-Plane Sensing-Processing: A Power-Efficient Approach for the Implementation of Privacy-Aware Networked Visual Sensors. Sensors, 14 (8), 15203-15226.

[7] Werblin, F., & Roska, B. (2001). Vertical interactions across ten parallel, stacked representations in the mammalian retina. Nature, 410, 583-587.

[8] Bradski, G. (2000). The OpenCV library. Dr. Dobb's Journal of Software Tools. Retrieved from http://opencv.org/

[9] Caltech frontal face dataset. (1999). Retrieved from http://www.vision.caltech.edu/html-files/archive.html

[10] Klette, R. (2014). Concise computer vision: An introduction into theory and algorithms. Springer, 375-413.

[11] Grauman, K., & Leibe, B. (2011). Visual object recognition. San Rafael, Calif.: Morgan & Claypool.

[12] Mallat SG. (1989). A theory for multiresolution signal decomposition: the wavelet representation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11 (7), 674–693.

[13] RAPP User's Manual. (2010). Retrieved from http://www.nongnu.org/rapp/doc/rapp/integral.png

[14] Freund, Y., & Schapire, R. (1999). A short Introduction to Boosting.Journal of Japanese Society for Artificial Intelligence, 14(5), 771-780.

[15] Lowe, D. (2001). The Viola/Jones Face Detector. Retrieved from http://www.cs.ubc.ca/~lowe/425/slides/13-ViolaJones.pdf

[16] Lienhart, R., Kuranov, A., & Pisarevsky, V. (2003). Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection. Pattern Recognition Lecture Notes in Computer Science, 2781, 297-304.

[17] Huang, C., Ai, H., Li, Y., & Lao, S. (2006). Learning Sparse Features in Granular Space for Multi-View Face Detection. 7th International Conference on Automatic Face and Gesture Recognition (FGR06).

[18] Sagha, H., Dehghani, M., & Enayati, E. (2008). Finding Sparse Features for Face Detection Using Genetic Algorithms. IEEE International Conference on Computational Cybernetics.

[19] Abramson, Y., Steux, B., & Ghorayeb, H. (2007). Yet Even Faster (YEF) Real-Time Object Detection. International Journal of Intelligent Systems Technologies and Applications IJISTA, 2 (2/3) 102-112.

[20] OpenCV documentation. Retrieved from http://docs.opencv.org