Adaboost con aplicación a detección de caras mediante el algoritmo de Viola-Jones

Trabajo final - Introducción al reconocimiento de patrones

Néstor Paz

Febrero de 2009

ÍNDICE

1 INTRODUCCIÓN

El presente trabajo tiene por objetivo el estudio del algoritmo Adaboost y su aplicación en el detector de caras de Viola-Jones.

El documento se puede dividir en tres partes, la primera aborda el paper "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting" de Freund y Sachapire.

La idea dentro de esta primera parte es entender el algoritmo Adaboost dada su aplicación en el detector de Viola-Jones. Para poder entenderlo es necesario hacer referencia al algoritmo Hedge(β). Uno de los aportes más significativos del paper, además de los algoritmos en si, es el análisis que los autores hacen de los mismos, donde establecen un marco teórico, encontrando límites que acotan el error máximo y mostrando la velocidad de convergencia de cada uno. Por lo tanto en esta primera se presentan también las demostraciones principales, de manera que quede claro el marco teórico y los resultados a los que se llega, más que meramente describir los algoritmos. Solo se llegó hasta el planteo de Adaboost para el caso de dos clases, que es lo necesario para entender la utilización que Viola-Jones hacen del mismo. En el paper se describen además dos extensiones para multiclase y aplicaciones a problemas de regresión que son realmente complejas y no son abordadas en este documento.

La segunda parte recorre el paper " Robust real-time Object Detection" de Viola-jones". Este paper esta muy bien presentado por lo que es difícil escapar a mantener la estructura original del mismo. Básicamente se presentan los principios sobre los que se basa el detector. En particular se destacan las características utilizadas, la selección de las mismas, el método de entrenamiento y la arquitectura del detector. Es de destacar que el enfoque de los autores está puesto más en la velocidad alcanzada por su detector, la cual va a permitir un nuevo mundo de aplicaciones (de tiempo real), que en los índices de detección y error, donde también alcanza una performance excelente.

La tercera parte esta dedicada a pruebas con un detector similar al de Viola-Jones, a partir del código Matlab y C++ de Sreekar Krisnha, donde se implementa la interface para el OpenCV's face detector creado por Rainer Lienhart.

Se compilaron 5 juegos de parámetros distintos y se aplicó cada uno de estos algoritmos compilados a 20 fotos seleccionadas. Se midió el índice de detección, el índice de falsos positivos y el tiempo empleado para cada una de las configuraciones. Se presentan los resultados, el análisis de algunos falsos positivos y las conclusiones sobre esta implementación.

2 PARTE 1: FUNDAMENTOS DE ADABOOST

2.1 Generalidades sobre boosting

El boosting esta inspirado en el algoritmo de aprendizaje en línea Hedge(β) propuesto por Freund y Sachapire.

La idea de boosting es generar un clasificador muy preciso a partir de clasificadores débiles. Boosting va a ir agregando clasificadores simples uno a uno y mediante una selección de las muestras de entrenamiento de cada uno de ellos y la construcción de una combinación de ellos afectada por pesos en los clasificadores así entrenados conseguirá un clasificador tan preciso como queramos.

2.2 Boosting y Adaboost

Adaboost por **ada**ptative **boost**ing, es un nuevo algoritmo de boosting presentado por Freund y Sachapire en el paper "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting"

2.2.1 Introducción

Freund y Sachapire, presentan a partir de un problema ameno el algoritmo Hedge(β), estudian su desempeño y obtienen interesantes límites para el error.

Luego presentan el adaboost y encuentran cierta forma de dualidad entre ambos algoritmos que permiten encontrar límites para el adaboost a partir de lo que se dedujo para el Hedge(β).

El problema presentado por Freund y Sachapire es el clásico problema de ganar dinero apostando en las carreras de caballos. Para conseguir esto, un apostador cansado de perder y viendo que sus amigos ganaban, decide recurrir a sus amigos apostadores, pidiéndoles que apuesten en su nombre. Decide jugar en cada ocasión una suma constante de dinero. Al principio le asigna a cada uno de sus amigos una suma igual de dinero. Si el pudiera saber de antemano a cual de todos ellos le va a ir mejor sin duda le daría a este para que le maneje todo su dinero. Dado que no es clarividente decide ir reasignando los montos en cada jugada de manera de obtener el mejor resultado posible, de manera que en toda la temporada su resultado se parezca lo mas posible a su amigo con mejor suerte.

En el paper, Freund y Sachapire describen un algoritmo simple para dar solución a este problema de asignación dinámica y muestran como éste puede ser aplicado a una gran variedad de problemas. La más destacable de estas es quizás el surgimiento del Adaboost. Este algoritmo de boosting convertirá a un conjunto de clasificadores débiles cuya performance sea apenas por encima de la elección al azar, en un clasificador fuerte arbitrariamente preciso.

2.2.2 Formalización del problema

Llamemos **A** al algoritmo del apostador que podrá elegir entre **N** estrategias posibles, numeraremos estas estrategias 1, ..,N. En cada paso t= 1, 2,. . ., T el apostador decide la distribución P^t sobre las estrategias. Entonces $P_i^t \geq 0$ es el monto asignado a la estrategia i y $\sum_{i=1}^{N} P_i^t = 1$.

A su vez, cada estrategia i sufre alguna pérdida l_i^t que está determinada por digamos el entorno adverso.

Por lo tanto la pérdida sufrida por A es $\sum_{i=1}^{N} P_i^t l_i^t = P^t l^t$.

A esta función de pérdida que la podemos interpretar como la pérdida promedio de las estrategias respecto a la elección de A, se le llama *pérdida de la mezcla*.

En todo el paper se supone que la perdida l_i^t esta acotada entre 0 y 1.

La meta de A, es minimizar su pérdida acumulada relativo a la pérdida de la mejor estrategia posible, esto es minimizar la pérdida neta: $L_A - \min_i L_i$

donde $L_A = \sum_{t=1}^{T} P^t \cdot l_i^t$ y $L_i = \sum_{t=1}^{T} l_i^t$.

L_A es entonces la pérdida acumulada por el algoritmo A en las primeras T jugadas.

2.2.3 Formalización de Boosting

Supongamos que el apostador esta cansado de consultar a los expertos para decidir las apuesta en cada carrera y decide, crear un algoritmo de computación que prediga el ganador de cada carrera, accediendo a la información usual con la que se cuenta antes de cada una (carreras ganadas por cada caballo, mejores tiempos de cada uno, como van las apuestas, etc.).

La idea es que el apostador coleccioné a través del conocimiento de los expertos una buena cantidad de reglas del dedo o reglas gruesas, como por ejemplo: apostar al caballo que ganó recientemente más carreras.

Aquí el apostador se enfrenta a dos problemas, el primero como presentarle a los expertos un conjunto de carreras que le permita extraer las más provechosas reglas del dedo y segundo como combinarlas en una sola regla que le permita sacar el mayor provecho posible.

Como ya vimos, boosting refiere a producir predicciones muy exactas a partir de combinar reglas débiles.

Formalmente, el booster es provisto con un conjunto de entrenamiento etiquetado $(x_1, y_1),...,(x_n, y_n)$ donde yi es la etiqueta asociada a xi. En el ejemplo de las carreras de caballos en la carrera i, xi podrían ser los datos observables antes de la carrera mientras que yi podría ser el número del ganador.

En cada ronda t= 1, 2,. . ., T el booster idea una distribución D_t sobre el conjunto de ejemplos y requiere (de no importa donde) una hipótesis débil h_t con error menor que ε_t respecto de la distribución D_t, esto es que ε_t es igual a la probabilidad de que $h_t(x) \neq y_i$ según la distribución D_t.

Vemos que la distribución D_t especifica la importancia de cada ejemplo para la actual ronda. Luego de T rondas, el booster debe combinar esas hipótesis débiles en un sola regla de predicción.

A diferencia de anteriores algoritmos, Hedge(β) no necesita conocimiento previo de cuán exactas sean las hipótesis débiles. En vez de eso, el se adapta a esas precisiones y genera una hipótesis por mayoría ponderada donde el peso de cada hipótesis se corresponde con su exactitud.

Como se probará mas adelante para problemas binarios, el error de la hipótesis final (respecto al conjunto de ejemplos dado) esta acotado por exp(-2$\sum_{t=1}^{T} \gamma_t^2$) donde $\varepsilon_t = 0.5 - \gamma_t$ es el error en la ronda t. Entonces, si podemos encontrar esas hipótesis, apenas mejor que elegir al azar en forma consistente, este límite muestra que el error de la hipótesis final cae exponencialmente rápido.

Queda claro que la precisión de la hipótesis final mejora, cuando la predicción de cualquiera de las hipótesis débiles mejora. Esto es contrario a los algoritmos previos donde la precisión de la hipótesis final solo dependía de la precisión de la hipótesis más débil.

A su vez, si todas las hipótesis débiles tienen la misma precisión, la performance de este nuevo algoritmo queda muy cerca de la del mejor algoritmo de boosting conocido.

2.3 El Algoritmo Hedge(β)

El pseudo código para Hedge(β) se muestra en la figura 1.

Algorithm Hedge(β)
Parameters: $\beta \in [0, 1]$
initial weight vector $\mathbf{w}^1 \in [0, 1]^N$ with $\sum_{i=1}^{N} w_i^1 = 1$
number of trials T
Do for $t = 1, 2, ..., T$

1. Choose allocation

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$$

2. Receive loss vector $\ell^t \in [0, 1]^N$ from environment.
3. Suffer loss $\mathbf{p}^t \cdot \ell^t$.
4. Set the new weights vector to be

$$w_i^{t+1} = w_i^t \beta^{\ell_i^t}$$

Figura 1.- Pseudo código de Hedge(β) para asignación on-line.

El algoritmo mantiene un vector de pesos cuyo valor para el tiempo t notamos de la siguiente manera $W^t = \langle w_1^t, ..., w_N^t \rangle$, siempre los pesos son no negativos. Además para t=1 el vector debe sumar uno, o sea $\sum_{i=1}^{N} w_i^1 = 1$.

Si supiéramos a priori cual es la mejor estrategia, le daríamos a esa mas peso, como es de esperar si no tenemos conocimiento a priori, le daremos a cada estrategia el mismo peso o sea $w_i^1 = 1/N$, notemos que los pesos de las siguientes rondas no tienen porque sumar uno.

El algoritmo asigna entre las estrategias utilizando el vector actual de pesos luego de normalizarlo, o sea, en el tiempo t el algoritmo elije el vector de distribución $P^t = \frac{W^t}{\sum_{i=1}^{N} w_i^t}$.

Luego que el vector de pérdidas l_t ha sido recibido, el vector de pesos W^t es actualizado utilizando la siguiente regla:

$$w_i^{t+1} = w_i^t \cdot \beta^{l_i^t}$$

En forma mas general, se puede mostrar que el análisis es aplicable con una pequeña modificación a la siguiente regla de actualización:

$$w_i^{t+1} = w_i^t \cdot U_\beta\left(l_i^t\right)$$

Donde $U_\beta : [0,1] \to [0,1]$ es cualquier función parametrizada por $\beta \in [0,1]$ que satisfaga la condición:

$\beta^r \leq U_{\beta(r)} \leq 1-(1-\beta)r$ Para todo $r \in [0,1]$

2.3.1 Análisis de Hedge(β)

La idea es encontrar los límites superior e inferior para la expresión $\sum_{i=1}^{N} w_i^{T+1}$, que juntos implican un límite superior para la pérdida total del algoritmo.

Lema1

Límite superior, se pude demostrar que para cualquier secuencia del vector de pérdidas $l^1,...,l^T$,

$$\ln\left(\sum_{i=1}^{N} w_i^{T+1}\right) \leq -(1-\beta)L_{Hedge(\beta)}$$

Para la demostración se utiliza que por concavidad $\alpha^r \leq 1-(1-\alpha)r$, para $\alpha \geq 0$ y $r \in [0,1]$.

$$\sum_{i=1}^{N} w_i^{t+1} = \sum_{i=1}^{N} w_i^t \beta^{l_i^t} \leq \sum_{i=1}^{N} w_i^t (1-(1-\beta)l_i^t) = \left(\sum_{i=1}^{N} w_i^t\right)(1-(1-\beta)P^t \cdot l^t)$$

la última igualdad es debida a la definición del vector P^t. Luego aplicamos repetidamente para t=1, …, T y teniendo en cuenta que $\sum_{i=1}^{N} w_i^1 = 1$ (porque así se había impuesto) nos queda que:

$$\sum_{i=1}^{N} w_i^{T+1} \le \prod_{t=1}^{T} (1-(1-\beta)P^t \cdot l^t) \le \exp(-(1-\beta)\sum_{t=1}^{T} P^t \cdot l^t)$$

Esto es dado que $1+x \le e^x$ para todo x, y recordando que $L = \sum_{t=1}^{T} P^t \cdot l_i^t$ entonces:

$$L_{Hedge(\beta)} \le \frac{-\ln(\sum_{i=1}^{N} w_i^{T+1})}{1-\beta}$$ Ecuación 5

Con lo que queda demostrado el lema 1.

Teorema2

Se puede demostrar que para cualquier secuencia del vector de pérdidas $l^1,...,l^T$, y para cualquier $i \in \{1,...,N\}$ se tiene que

$$L_{Hedge(\beta)} \le \frac{-\ln(w_i^1) - L_i \ln\beta}{1-\beta}$$ Ecuación 7

y más genéricamente que para cualquier conjunto no vacío $S \subseteq \{1,..,N\}$

$$L_{Hedge(\beta)} \le \frac{-\ln(\sum_{i\in S} w_i^1) - (\ln\beta)\max_{i\in S} L_i}{1-\beta}$$ Ecuación 8

Vemos que las 2 expresiones son iguales si S= { i }.

La segunda se demuestra en forma sencilla dado que:

$$\sum_{i=1}^{N} w_i^{T+1} \ge \sum_{i\in S} w_i^{T+1} = \sum_{i\in S} w_i^1 \beta^{L_i} \ge \beta^{\max_{i\in S} L_i} \sum_{i\in S} w_i^1$$

Ahora, juntando con la ecuación 5 nos queda pronta la demostración.

El límite encontrado por la ecuación 7 establece que la performance de Hedge(β) no es mucho peor que la de la mejor estrategia i para la secuencia dada. La pérdida depende entonces de los pesos iniciales y de la elección de β.

Si no tenemos información a priori, y tomamos todos los pesos iniciales iguales o sea $w_i^1 = 1/N$ entonces el límite se trasforma en:

$$\boxed{L_{Hedge(\beta)} \leq \frac{\min_i L_i \ln(1/\beta) + \ln N}{1-\beta}} \qquad \text{Ecuación 9}$$

Al depender solo de ln N, vemos que el límite es bueno inclusive si N es un número grande de estrategias.

El límite de la ecuación 8, que es una generalización del límite encontrado en la ecuación 7, es especialmente aplicable cuando el numero de estrategias N es infinito, entonces la sumatorias se pueden trasformar en una integral y el máximo en supremo. Este límite se trasforma en:

$$L_{Hedge(\beta)} \leq c \min_i L_i + a \ln N$$

Donde $c = \ln(1/\beta)/(1-\beta)$ y $a = 1/(1-\beta)$.

Utilizando los resultados de Vovk (referencia 1), se puede demostrar que los valores de las constantes *a* y *c* alcanzados por el algoritmo son óptimos.

Elección de β

Si bien lo estudiado es para un β cualquiera, siempre vamos a querer elegir β de manera de maximizar el conocimiento previo que tengamos acerca del problema específico.

Lema 4

Supongamos que $0 \leq L \leq \tilde{L}$ y que $0 \leq R \leq \tilde{R}$, tomemos $\beta = g(\tilde{L}/\tilde{R})$ donde $g(z) = 1/(1+\sqrt{2/z})$, entonces $\frac{-L\ln\beta + R}{1-\beta} \leq L + \sqrt{2\tilde{L}\tilde{R}} + R$

Este lema puede ser aplicado a cualquiera de los límites encontrados dado que son de la misma forma. Aplicado al límite encontrado en la ecuación 9, nos queda:

$$L_{Hedge(\beta)} \le \min_i L_i + \sqrt{2\tilde{L}\ln N} + \ln N \text{ , para } \beta = g(\tilde{L}/\ln N)$$

En general se conoce el número de rondas T, y lo podemos utilizar como límite superior de la pérdida acumulada para cada estrategia L, o sea $\tilde{L} = T$.

Dividiendo ambos lados entre T, obtenemos un límite explícito de la tasa del promedio de pérdida por ronda del algoritmo, que se aproxima a la pérdida promedio de la mejor estrategia.

$$\frac{L_{Hedge(\beta)}}{T} \le \frac{\min_i L_i}{T} + \frac{\sqrt{2\tilde{L}\ln N}}{T} + \frac{\ln N}{T}$$

Como $\tilde{L} \le T$, en el peor de los casos la tasa de convergencia es del orden de $\sqrt{(\ln N)/T}$, pero si $\tilde{L}$ es del orden de cerdo, entonces la convergencia será mucho mas rápida, del orden de (ln N)/T.

Para el caso general no se puede mejorar el termino de $\sqrt{2\tilde{L}\ln N}$ por mas de un factor constante, lo que da un límite inferior al problema.

2.4 Marco de trabajo teórico para boosting.

Sea **X** el dominio, llamaremos concepto a una función booleana $c : X \to \{0,1\}$

C una clase, será una colección de conceptos.

Se tiene acceso a ejemplos etiquetados de la forma $(x, c(x))$, donde x es tomado con una cierta distribución D fija y no conocida en el dominio X.

La idea es que el algoritmo de aprendizaje logre, luego de un cierto tiempo generar una hipótesis $h : X \to [0,1]$. El valor h(x) puede ser interpretado como una predicción randómica de la etiqueta de x, que valdrá 1 con probabilidad h(x) y 0 con probabilidad 1-h(x).

El error de h es el valor esperado de $E_{x-D}(|h(x) - c(x)|)$, donde x es elegido según la distribución D.

Un weak PAC –learning algorithm, al cual para simplificar la notación llamaremos clasificador débil es aquel que dados $\varepsilon, \delta > 0$ y un ejemplo al azar nos da con probabilidad 1-δ una hipótesis con error $\varepsilon \le 1/2 - \gamma$ con $\gamma > 0$.

Lo que hace boosting es tomar esos clasificadores débiles, llamarlos múltiples veces cada vez con diferente distribución sobre X, y finalmente generar una única hipótesis con la combinación de las hipótesis así generadas. La idea intuitiva es que se varía la distribución de manera que se incremente la probabilidad de los ejemplos más difíciles, y así obligar al clasificador débil a generar nuevas hipótesis que tengan menor error en esa parte.

2.5 Adaboost para 2 clases

El clasificador débil recibirá ejemplos del tipo (x_i, y_i) elegido siguiendo una distribución **P**, fija pero desconocida. A su vez, en esta versión para dos clases, el conjunto **Y** solo tiene dos posibles etiquetas $Y = \{0,1\}$.

Asumimos que una secuencia de N ejemplos de entrenamiento, muestras etiquetadas, $(x_1, y_1) \ldots (x_N, y_N)$ son elegidas de **X**x**Y** siguiendo la distribución **P**. Se utilizará boosting para encontrar la hipótesis h_f que es consistente con la mayoría de los ejemplos, $h_f(x_i) = y_i$ para la mayoría de $1 \le i \le N$

El algoritmo Adaboost se muestra en la figura 2, la meta del algoritmo es encontrar la hipótesis final con el menor error relativo, dada la distribución D sobre los ejemplos de entrenamiento.
A diferencia de la distribución **P** que es sobre **X**x**Y** y es dada por la "naturaleza", la distribución **D** es solo sobre las instancias de los ejemplos de entrenamiento, y es controlada por el clasificador.
A falta de conocimiento previo, la distribución D será generalmente la distribución uniforme, con D(i)=1/N. El algoritmo mantiene un conjunto de pesos w^t sobre el conjunto de entrenamiento. En cada iteración t la distribución P^t es calculada para normalizar esos pesos. Con esta distribución, se alimenta al clasificador débil, que genera una hipótesis h_t que se espera tenga poco error sobre esa distribución. Con esta nueva hipótesis h_t, el algoritmo genera el siguiente vector de pesos w^{t+1} y el proceso se repite. Luego de T iteraciones, se obtiene la hipótesis final h_f. Esta salida h_f combina las salidas de los T clasificadores débiles utilizando voto pesado por mayoría.

Algorithm AdaBoost

Input: sequence of N labeled examples $\langle (x_1, y_1), ..., (x_N, y_N) \rangle$
distribution D over the N examples
weak learning algorithm **WeakLearn**
integer T specifying number of iterations

Initialize the weight vector: $w_i^1 = D(i)$ for $i = 1, ..., N$.

Do for $t = 1, 2, ..., T$

1. Set

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^{N} w_i^t}$$

2. Call **WeakLearn**, providing it with the distribution $\mathbf{p}^t$; get back a hypothesis $h_t: X \to [0, 1]$.
3. Calculate the error of h_t: $\varepsilon_t = \sum_{i=1}^{N} p_i^t \, |h_t(x_i) - y_i|$.
4. Set $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$.
5. Set the new weights vector to be

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

Output the hypothesis

$$h_f(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} (\log 1/\beta_t) \, h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \log 1/\beta_t \\ 0 & \text{otherwise.} \end{cases}$$

Figura 2. - algoritmo Adaboost.

El parámetro β_t es elegido en función del error ε_t y es utilizado para actualizar el vector de pesos. Es interesante destacar que, el adaboost, utiliza la misma regla de combinación que el mejor algoritmo conocido en la práctica y que antes carecía de justificación teórica.

2.5.1 Comparación con Hedge(β)

Comparando los dos algoritmos vemos que hay una gran similitud entre ellos, se evidencia una suerte de dualidad entre el algoritmo de asignación en línea y el problema del boosting. O sea se puede mapear el problema del boosting en el problema de la asignación en línea.

Podría pensarse intuitivamente las estrategias como las hipótesis débiles, y las rondas como los ejemplos; sin embargo los autores proponen el mapeo inverso, o sea las estrategias con los ejemplos y las rondas se asocian a los clasificadores débiles. Otra dualidad aparece en la definición del error, dado que para el Hedge(β), la perdida l_i^t será pequeña si la i-esima estrategia se presenta como muy acertada en la ronda t, mientras que para Adaboost la perdida $l_i^t = 1 - |h_t(x_i) - y_i|$ que aparece como parte de la actualización de pesos es pequeña si la hipótesis sugerida es una mala predicción.

La explicación es que mientras para Hedge(β) el peso de una estrategia se incrementa en la medida que esta es exitosa, para Adaboost el peso asociado a un ejemplo se aumenta en la medida de que este sea mas difícil de clasificar.

La diferencia técnica más grande entre ambos algoritmos es que en Adaboost el parámetro β ya no es constante sino que varia en cada iteración dependiendo del error ε_t.

2.5.2 Análisis de Adaboost para dos clases

Es factible analizar el Adaboost utilizando un β fijo y algoritmo Hedge(β) con lo cual se puede obtener un primer limite para el error (del orden de $e^{-T\gamma^2/2}$), sin embargo, con el uso mas inteligente del β, se llega con Adaboost a resultados muy superiores referidos al límite del error.

Empezamos por demostrar el teorema 6:

Suponemos que el clasificador débil, cuando es llamado por Adaboost genera hipótesis con error $\varepsilon_1 \ldots\ldots \varepsilon_T$, entonces el error de la hipótesis final generada por Adaboost $\varepsilon = \Pr_{i--D}\left[h_f(x_i) \neq y_i\right]$, esta acotado por:

$$\varepsilon \leq 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t (1-\varepsilon_t)}$$

La demostración se hace mediante la adaptación del lema 1 y teorema 2 de la primera parte:

$$\sum_{i=1}^{N} w_i^{t+1} = \sum_{i=1}^{N} w_i^t \beta^{1-|h_t(x_i - y_i)|} \leq \sum_{i=1}^{N} w_i^t (1-(1-\beta_t)(1-|h_t(x_i) - y_i| = (\sum_{i=1}^{N} w_i^t)(1-(1-\varepsilon_t)(1-\beta_t))$$

Combinando este proceso para todo t de 1 a T. Obtenemos:

$$\sum_{i=1}^{N} w_i^{T+1} \leq \prod_{t=1}^{T} (1-(1-\varepsilon_t)(1-\beta_t))$$ Ecuación 16

Luego, la hipótesis final h_f como fue definida en el algoritmo comete error en la instancia i si y solo si:

$$\prod_{t=1}^{T} \beta_t^{-|h_t(x_i-y_i)|} \geq (\prod_{t=1}^{T} \beta_t)^{-1/2}$$ Ecuación 17

El peso final para cualquier instancia i es:

$$w_i^{T+1} = D(i) \prod_{t=1}^{T} \beta_t^{1-|h_t(x_i)-y_i|}$$ Ecuación 18

Luego, combinando la ecuación 17 y 18 y acotando por la suma de los ejemplos donde la hipótesis final es incorrecta:

$$\sum_{i=1}^{N} w_i^{T+1} \geq \sum_{i:h_f(x_i)\neq y_i} w_i^{T+1} \geq \left(\sum_{i:h_f(x_i)\neq y_i} D(i) \right) \left(\prod_{t=1}^{T} \beta_t \right)^{1/2} = \varepsilon . \left(\prod_{t=1}^{T} \beta_t \right)^{1/2}$$

donde ε es el error en la hipótesis final.
Combinando esta ecuación con la 16, tenemos que:

$$\varepsilon \leq \prod_{t=1}^{T} \frac{1-(1-\varepsilon_t)(1-\beta_t)}{\sqrt{\beta_t}}$$

Como todos lo términos son positivos podemos minimizar, minimizando cada término por separado, lo que calculamos el valor de β_t. Al derivar e igualar a cero nos

queda: $\beta_t = \varepsilon_t /(1-\varepsilon_t)$. Sustituyendo este valor de β_t en la ecuación 20 queda completa la demostración.

Este error también puede ser reescrito como:

$$\varepsilon \le \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} = \exp\left(-\sum_{t=1}^{T} KL(1/2 \| 1/2-\gamma_t) \right) \le \exp\left(-2\sum_{t=1}^{T} \gamma_t^2 \right)$$

donde KL(aIIb)=a ln (a/b) +(1-a) ln ((1-a)/(1-b)) es la divergencia de Kullback-Leibler.

En el caso de que el error de todas las hipótesis se igual a $1/2-\gamma$, entonces la expresión se simplifica a:

$$\varepsilon \le \left(1-4\gamma^2\right)^{T/2} = \exp(-T.KL(1/2 \| 1/2-\gamma_t)) \le \exp(-2T\gamma^2)$$

Finalmente resaltemos que: el teorema 6 implica que en Adaboost el error final depende del error de cada una de las hipótesis débiles, mientras en previos algoritmos dependía solamente del error máximo de la hipótesis más débil, ignorando las ventajas de aquellas que tenían menor error.

3 PARTE 2: EL DETECTOR DE CARAS DE VIOLA-JONES

3.1 Introducción

En el paper '**Robust real-time Object Detection'**, Paul Viola y Michel Jones presentan un marco de trabajo para la realización de un detector de objetos visuales que es capaz de realizar el procesamiento de imágenes en forma extremadamente rápida, logrando no obstante una alta tasa de detección.

El trabajo de los mismos, estaba motivado en gran parte por la tarea de realizar un detector de caras eficiente y rápido. Queda claro que el objetivo del detector es posibilitar la detección de caras sobre video en tiempo real, sin que para ello sea necesario la utilización de computadoras especialmente potentes.

Si bien el resto del artículo esta referido a la detección de caras, se entiende que el mismo razonamiento es valido para la detección de cualquier objeto.

En este trabajo los autores terminan diseñando un sistema de detección de caras que alcanza niveles, tanto de detección como de falsos positivos comparables con los mejores hasta ese momento (referencias 2, 3, 4, 5 y 6), pero con una velocidad de procesamiento muy superior.

Tomando como base imágenes de 384 por 288 píxeles, el detector es capaz de procesar 15 cuadros por segundo (o sea 15 imágenes de 384 por 288 píxeles por segundo), esto operando en un Intel Pentium III de 700 Mhz convencional.

Este detector trabaja con imágenes en escalas de grises alcanzando buenos resultados, esto lo diferencia de otros que utilizan más información como el color de los píxeles, etc. Esa información podría ser agregada a este detector para mejorar más aun su performance.

Para los autores hay tres contribuciones principales que caracterizan a este detector, se puede ver que todas ellas están enfocadas hacia la eficiencia computacional, de manera de aumentar la velocidad y la eficiencia.

La primera es la utilización (para representar la imagen) de la imagen integral; esto permite evaluar de forma muy rápida las características utilizadas. No se trabaja entonces con la intensidad de la imagen en si, sino que con una representación de la misma. La gran ventaja es que se pueden calcular las características utilizadas en cualquier lugar de la imagen y en cualquier escala en el mismo lapso de tiempo (como veremos mas adelante) y en forma sumamente eficiente desde el punto de vista del gasto computacional.

La segunda contribución es el método de construcción del clasificador, que se lleva a cabo seleccionando un número reducido de significativas características, utilizando para ello una modificación del algoritmo Adaboost.

Adaboost, es utilizado tanto para seleccionar las características, como para entrenar los clasificadores.

La tercera contribución es el método para combinar una sucesión de clasificadores cada vez más complejos en una estructura de cascada. La idea es que los clasificadores más sencillos, de bajo costo computacional descarten una gran cantidad de sub-ventanas, dejando así concentrarse a los clasificadores más complejos en las zonas donde es más promisorio que haya una cara.

Este procedimiento incrementa drásticamente la velocidad del detector, esto es fácil de intuir ya que prácticamente todas las sub-ventanas que se le presentan al detector son negativas (o sea que no hay en esa sub-ventana una cara). Un sencillo experimento realizado por los autores, que comentaremos luego, da una idea de hasta que punto esto es así.

La versión final del detector de Viola-Jones presentada, esta compuesta por 32 clasificadores puestos en cascada. En el proceso una imagen que es reconocida como una cara necesita alrededor de 80.000 operaciones, sin embargo gracias a esta arquitectura de cascada el promedio de operaciones para una sub-ventana es muy bajo. Probado en un difícil conjunto de datos de prueba que contenía 507 caras y 75 millones de sub-ventanas, la detección se llevó a cabo utilizando un promedio de 270 instrucciones de microprocesador por cada sub-ventana. Este resultado muestra que este detector es a eficiencia similar, 15 veces más rápido que el considerado mejor hasta ese momento (Referencia 3).

Esta claro que un detector de estas características tiene un rango importante de aplicaciones, pues su alta velocidad posibilita, por un lado, detección en tiempo real (aplicable en codificación de video, tele-conferencias, seguimiento de objetos en movimiento, etc.) y por otro aplicación en diversos dispositivos de baja potencia de cálculo, handhelds y sistemas embebidos (por ejemplo cámaras de fotos).

3.2 Bases del detector

3.2.1 Las características

Este detector de objetos clasifica las imágenes basado en la evaluación de características sencillas. Las principales razones para utilizar características son que por un lado éstas pueden actuar como un codificador del dominio del conocimiento y por otro que los sistemas basados en características son mucho más rápidos que los que operan directamente con los píxeles.

Las características utilizadas por Viola-Jones son del estilo de las características Harr basis (Wavelets) y han sido utilizadas por otros autores (referencia 7). Estas características son las 'características rectángulo', existen varias formas de dichas características, pero todas ellas se basan en la suma y diferencia de los valores de los píxeles de áreas dentro de rectángulos.

Viola-Jones utilizan tres características, la característica *Dos-rectángulos* con orientación Horizontal y vertical, la *tres-rectángulos* y la *cuatro-rectángulos.* Ver figura 3.

El valor de la característica *Dos-rectángulos* es la diferencia entre la suma del valor de los píxeles dentro de cada región, o sea se suma el valor de los píxeles dentro de una región (mitad del rectángulo donde se avalúa la característica) y se le resta la suma del valor de los píxeles de la otra región (la otra mitad del rectángulo).
El valor de la característica *Tres-rectángulos* es diferencia entre la suma del valor de los píxeles dentro de los rectángulos exteriores y el rectángulo interior.
El valor de la característica *Cuatro-rectángulos* es la diferencia entre la suma del valor de los píxeles dentro de los rectángulos en una diagonal del valor y la suma de los píxeles dentro de los rectángulos de la otra diagonal. Ver figura 3.

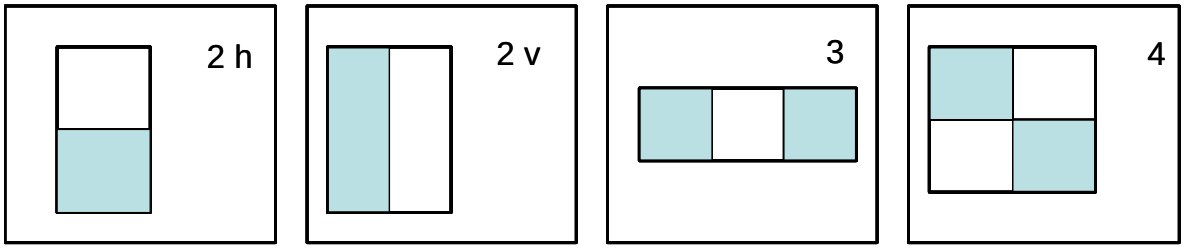


Figura 3.- Características dos rectángulos horizontal, dos rectángulos vertical, tres rectángulos y cuatro rectángulos.

La resolución base con la que trabaja el detector es de 24 por 24 píxeles, el set total de características para este tamaño de imagen es extremadamente alto **45396**, mucho mayor que la cantidad de píxeles contenidos en la imagen de 24 por 24 píxeles, 576 (la misma cantidad de píxeles) serían las necesarias para describir completamente la imagen. En este sentido es que se utilizará Adaboost para seleccionar las más importantes.

En una posterior mejora de este detector, Lienhart añade dos tipos nuevos, los tipos 3 y 4 (ver figura 4) y utiliza los tipos 1, 2, 3 y 4 descartando el tipo 5. A su vez introduce nuevas orientaciones 45º y 135

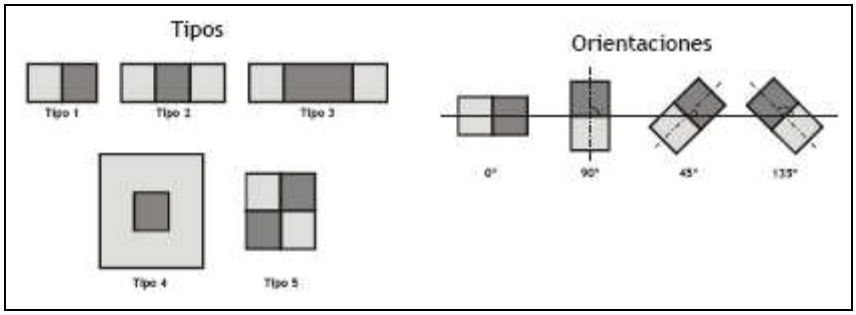


Figura 4. -Diferentes tipos y orientaciones de características tipo Haar, propuestas por Viola-Jones y Lienhart et al. Las zonas claras se computan con signo positivo y las oscuras con signo negativo.

3.2.2 La imagen integral

Las características rectángulo, pueden ser calculadas en forma extremadamente rápida utilizando una representación intermedia de la imagen que los autores llaman *'Imagen integral'*, en vez de trabajar con la imagen misma.

La imagen integral en un punto (*x, y*) de la imagen base, es la suma de todos los píxeles contenidos en el rectángulo formado por el origen *(0,0)* y punto(*x, y*). O sea:

$$ii(x,y) = \sum_{xi \le x, yi \le y} i(x_i, y_i)$$

donde ii(x,y) es la imagen integral en el punto (x,y) e i(x,y) es el valor de los píxeles de la imagen original, ver figura 5.

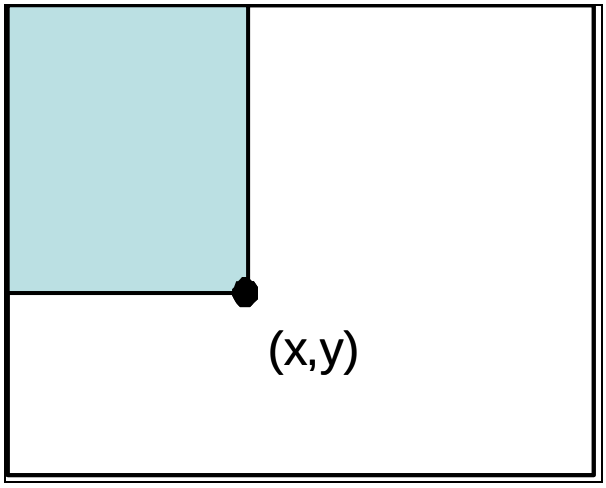


Figura 5.- Imagen integral del punto (x,y), corresponde a la suma de los píxeles del área coloreada.

Utilizando la siguiente recursividad, se puede obtener la imagen integral en una sola pasada sobre la imagen original:

S(x,y)= S(x, y-1) + i (x,y) con S(x, -1)=0 y ii(-1,y)=0

ii(x,y)= ii (x, y-1) + S(x,y) Donde S (x,y) es la suma hasta y de la fila x.

Utilizando la imagen integral cualquier suma dentro de un rectángulo puede ser computada en 4 referencias a la tabla, ver figura 6.

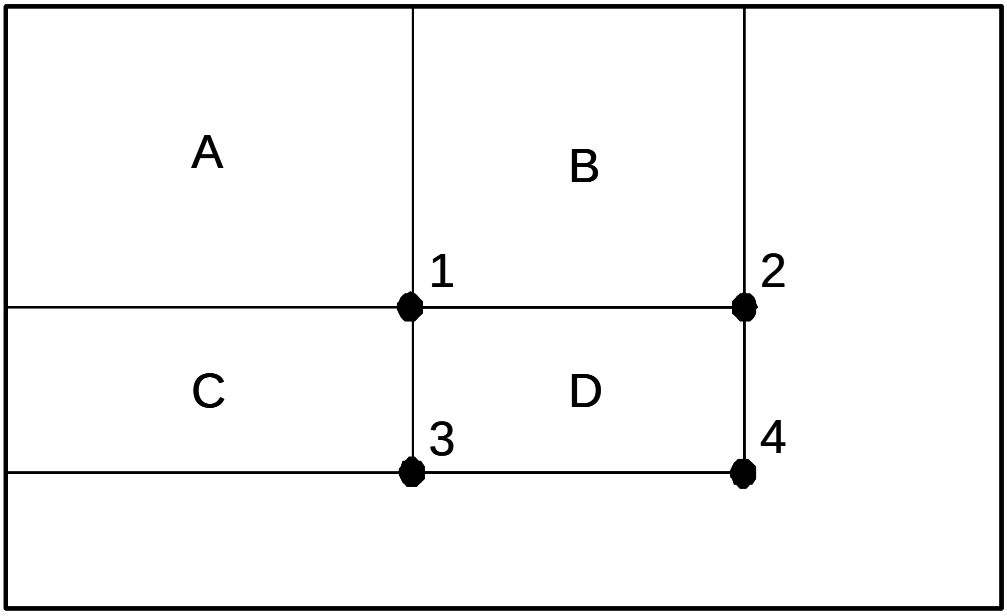


Figura 6.- Accediendo al resultado de la imagen integral de los puntos 1, 2, 3 y 4, podemos calcular en forma sencilla y rápida la suma de los píxeles dentro de cualquiera de los rectángulos.

Por ejemplo, la suma de los píxeles dentro del rectángulo D puede ser calculada fácilmente como 4+1-2-3. Donde 4 es la imagen integral en el punto 4, etc.

Entonces es claro, que la resta del área de dos rectángulos cualesquiera puede ser calculada en 8 referencias a la tabla, pero en las características de dos rectángulos al ser estos adyacentes se puede calcular con solo seis referencias, solo ocho para el caso de tres rectángulos y nueve para la de cuatro rectángulos.

Si bien por un lado las características rectángulo pueden resultar a priori un poco primitivas o toscas, en comparación con otras alternativas (Steerable filtres, integrales difusas, etc.), por un lado son capaces de dar una representación muy rica de la imagen si son eficientemente entrenadas y por otro su extremadamente alta eficiencia computacional compensan con creces su falta de flexibilidad.

Además, gracias a la imagen integral cualquier característica rectángulo puede ser calculada en cualquier lugar de la imagen y a cualquier escala utilizando para ello unas pocas operaciones, esto hace que este detector sea extremadamente rápido.

3.2.3 Selección de características

Como ya hemos visto para cada sub-ventana de 24 por 24 píxeles tenemos un total de 45396 características rectángulo, un número muy superior a la cantidad de píxeles contenidos en esa imagen. Si bien, como vimos, cada característica se puede calcular en forma muy eficiente,

calcular todas ellas para cada sub-ventana es inmanejable, por lo que se debe hacer una selección de las mismas. Dada la experiencia, esta visto que un pequeño numero de estas correctamente combinadas es suficiente para formar un clasificador eficiente. Lo interesante es encontrar cuales son esas características.

Por lo tanto debe utilizarse algún método para realizar la selección de características. Viola-Jones utiliza una variante de Adaboost, tanto para seleccionar las características como para entrenar el clasificador.

Como vimos en las secciones anteriores Adaboost utiliza una combinación de algoritmos débiles para terminar formando un algoritmo de clasificaron muy fuerte. Lo interesante de utilizar Adaboost es que este procedimiento de aprendizaje es muy fuerte, y como ya hemos visto el error de entrenamiento tiende a cero en forma exponencial con el número de turnos.

Adaboost puede ser interpretado como un ambicioso proceso de selección de características, dado que es un mecanismo agresivo para seleccionar un pequeño conjunto de buenas funciones de clasificación.

El truco esta en asociar las funciones de clasificación (los llamados clasificadores débiles) con las características rectángulo. Adaboost se convierte entonces en un efectivo procedimiento para selección un número reducido de "Buenas Características".

El procedimiento práctico para lograrlo es restringir cada clasificador débil al conjunto de funciones de clasificación donde, cada una dependa de solamente una característica rectángulo. Entonces, cada clasificador débil será diseñado para seleccionar la característica rectángulo que mejor separa los ejemplos negativos de los positivos. A su vez para cada función de clasificación el clasificador débil determina el umbral óptimo de clasificación que minimiza el número de ejemplos mal clasificados.

Entonces si $h_j(x)$ es el clasificador débil que consiste en la característica f_j, el umbral θ_j y p_j para mantener el signo de la inecuación:

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

donde x es una sub-ventana de 24 por 24 píxeles.

En la práctica, ninguna característica sola es capaz de realizar la tarea con bajo error, las primeras resultantes del proceso de selección producen errores entre el 10 y el 30%, mientras que las seleccionadas más adelante y que por lo tanto realizan una tarea mas complicada (debido a la naturaleza del Adaboost) producen errores entre el 40 y 50%.

Se muestra en la tabla1 el proceso de selección y entrenamiento.

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights,
  $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
  so that w_t is a probability distribution.
  2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$.
  3. Choose the classifier, h_t, with the lowest error ϵ_t.
  4. Update the weights:
  $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$
  where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
where $\alpha_t = \log \frac{1}{\beta_t}$

Table 1: The boosting algorithm for learning a query online. T hypotheses are constructed each using a single feature. The final hypothesis is a weighted linear combination of the T hypotheses where the weights are inversely proportional to the training errors.

3.2.4 Resultados de la selección y aprendizaje:

Experiencias realizadas por los autores demostraron que un clasificador construido con 200 características (recordemos que teníamos 45396 posibles) produce resultados razonables, alcanzando para un 95% de detección una tasa de falsos positivos de 1 en 14084. Ver figura 7.

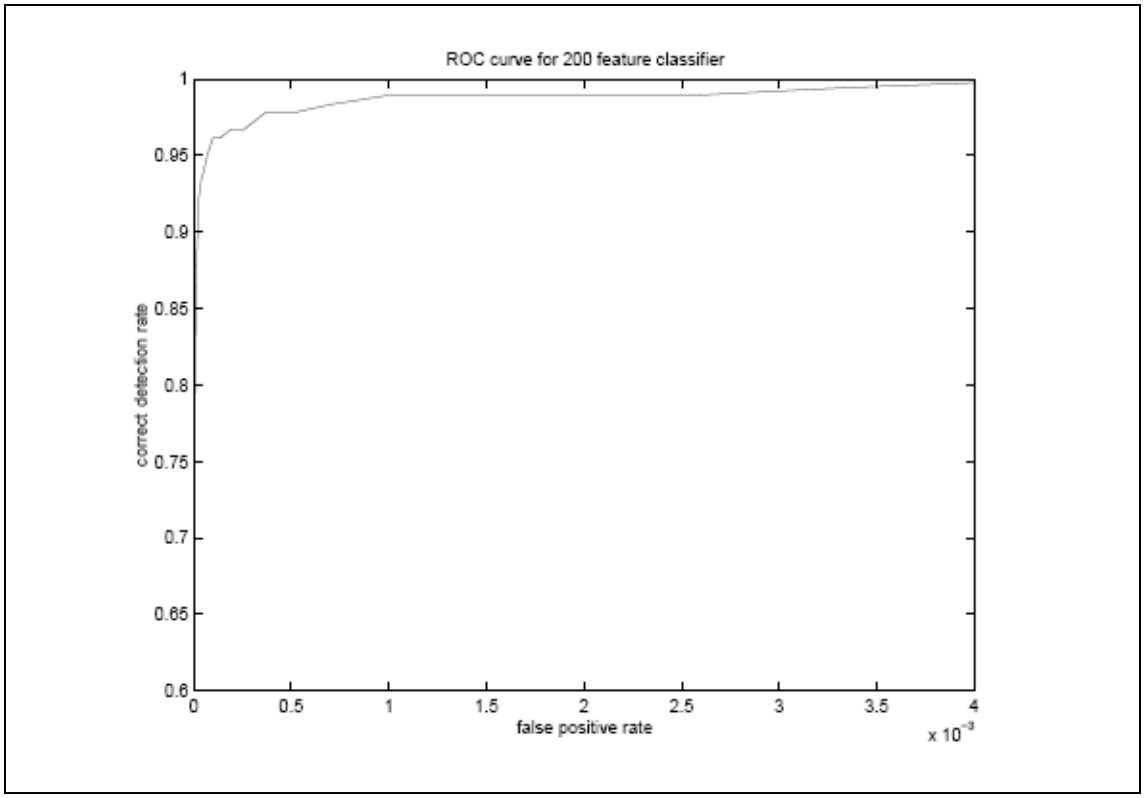


Figura 7.- Curva ROC para un clasificador compuesto por 200 características rectángulo

En nuestro caso las características rectángulo que inicialmente selecciona el algoritmo Adaboost son muy significativas, y además les podemos dar una interpretación intuitiva sencilla. Ver figura 8.

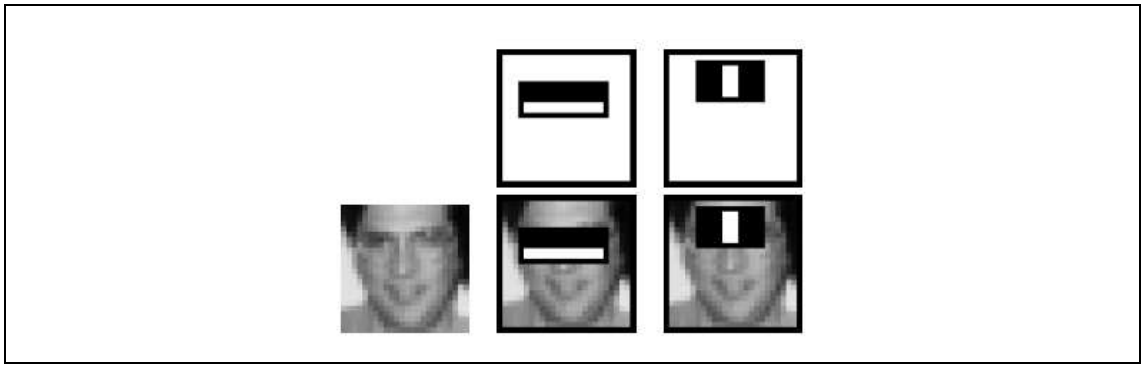


Figura 8.- Primera y segunda característica seleccionada por Adaboost.

La primera característica seleccionada podemos interpretarla fácilmente, dado que la zona correspondiente a los ojos se presentará más oscura que la correspondiente a la parte de la cara inmediatamente abajo, formada por los pómulos y nariz. La segunda quizás más fácil aun, hace notar las diferencia entre la intensidad en la zona correspondiente a cada ojo (zona mas

oscura) con de la zona ente ellos, correspondiente a parte de la nariz y generalmente más iluminada.

3.2.5 Arquitectura en cascada

Detrás de la elección de la estructura en cascada esta la necesidad de lograr un altísimo nivel de detección a la vez que una reducción drástica de la necesidad de cálculo y por ende un bajo costo computacional.
La idea fundamental para lograrlo, es la habilidad de los muy eficientes y sencillos clasificadores entrenados con Adaboost de poder rechazar la mayoría de las sub-ventanas, dejando pasar todos los casos positivos. Esos clasificadores muy sencillos y de bajo costo computacional se ponen en las primeras etapas de manera que en pocas operaciones se pueda rechazar la mayoría de las ventanas, dejando así los clasificadores más complejos concentrarse en pocos casos.
Es de destacar que muy pocas de las miles de ventanas a analizar contiene un objeto, por lo que la gran mayoría de las mismas deberán ser rechazadas.

Cada una de las etapas de la cascada es entrenada utilizando Adaboost. El primer clasificador contiene las dos características que vimos en la figura 8. El umbral inicial del clasificador es calculado para tener el mínimo error de entrenamiento, luego ese umbral es bajado de manera de obtener niveles de detección mas altos, por supuesto esto implica niveles también mas altos de falsos positivos. Se procede entonces a ajustar el umbral de este primer clasificador para obtener 100% de detecciones con un índice de falsos positivos del 40%. El umbral inicial es $\frac{1}{2}\sum_{t=1}^{T}\alpha_t$.

Esta performance, si bien puede parecer no muy buena, es capaz de reducir a menos de la mitad las ventanas a explorar en unas pocas operaciones, asegurando que todas las caras son detectadas.
Esto es fácil de ver dado que una característica rectángulo es evaluada con máximo 9 accesos a la tabla, calcular el clasificador débil para cada característica es simplemente calcular un umbral y luego combinar los clasificadores son solo multiplicación y suma.
Entonces con solo 60 operaciones de microprocesador ya tenemos una gran parte de las ventanas descartadas.

La estructura de la arquitectura en cascada se muestra en la figura 9.
Básicamente la estructura del clasificador adopta la forma de un árbol de decisión. Si el primer clasificador da por buena una ventana, esta pasa al segundo clasificador que también ha sido entrenado para producir una alta tasa de detección, sino la ventana es descartada. El segundo

clasificador la evalúa, si la da por buena se la pasa al tercer clasificador, sino la descarta, y así sigue todo el proceso.

Es claro que en una imagen normal la gran mayoría de las ventanas exploradas serán negativas. Por lo que en el proceso es deseable rechazar tantas ventanas como sea posible, lo antes posible, esto acelerara el proceso.

Es claro también, que a medida que avanzamos en los clasificadores, estos cada vez tiene una tarea más difícil ya que estos no ven las imágenes más típicas sino las que ya pasaron por las otras etapas. Es usual que estos clasificadores se entrenen con imágenes que ya pasaron por las etapas anteriores o sea que los negativos serán en realidad falsos positivos de las otras etapas.

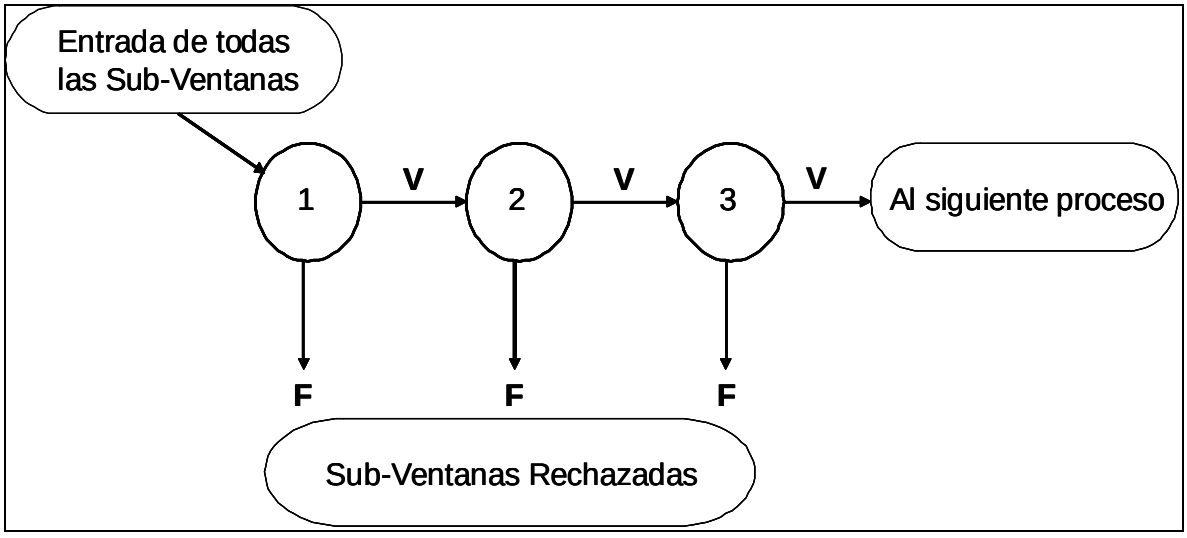


Figura 9.- estructura de clasificadores en cascada

3.3 Entrenamiento de la cascada de clasificadores

Para el diseño de la cascada es necesario fijar las metas de detección y performance. Lo usual sería lograr niveles de detección entre el 85 y 95%, con tasa de falso positivos muy bajas, del orden de 10^{-5} o 10^{-6}.

La idea es alcanzar esos registros pero con significativamente menor esfuerzo computacional.

Dado un clasificador en cascada entrenado, la tasa **F** de falsos positivos viene dada por la siguiente expresión:

$$F = \prod_{i=1}^{K} f_i$$

Donde ***fi*** es la tasa de falsos positivos de la etapa i y ***K*** es el número de etapas del clasificador.
Por otro lado la tasa de detección ***D*** viene dada por la expresión:

$$D = \prod_{i=1}^{K} d_i$$

Por ejemplo para una tasa de detección de 90% y un detector de 10 etapas, cada etapa debe alcanzar una tasa de detección de 0.99, ya que $0.99^{10} \approx 0.9$.
Si bien esto parece en principio un poco complicado de lograr, vemos que la tarea se simplifica por la poco exigente tasa de falsos positivos **F**, ya que $6 \times 10^{-6} \approx 0.3^{10}$, por lo que podemos permitirnos una tasa de falsos positivos de 30%.

El número esperado de características a evaluar en un proceso de teste es

$$N = n_0 + \sum_{i=1}^{K} (n_i \prod_{j<i} p_j)$$

Donde K es el número de etapas o clasificadores, p_i es la tasa de detección del i-ésimo clasificador y n_i es la cantidad de características de i-ésimo clasificador.

Observación importante:
Adaboost esta diseñado para minimizar el error y no para alcanzar altas tasas de detección a costa de tolerantes tasas de falsos positivos. La manera de lograr esto es jugando con el umbral del perceptron generado por Adaboost, umbrales más bajos darán como resultado tasas de detección mas altas y tasas de falsos positivos mas altas, umbrales mas altos darán tasas de falsos positivos mas bajas pero peores tasas de detención. No queda claro si a pesar de este juego con los umbrales las granitas de Adaboost se siguen manteniendo.

El diseño y entrenamiento del sistema implica algunos compromisos, por un lado cuantas más características tenga un clasificador, obtendrá tasas mas altas de detección así como mejores tasas de falsos positivos, pero cuanto mas características se evalúen mas tiempo computacional

precisamos, por lo que se debe minimizar ***N***, para un ***D*** y ***F*** fijados. El óptimo es compromiso entre el número de etapas, los objetivos a alcanzar y el número esperado de características a evaluar, que a su vez esto depende de cómo se diseñe cada etapa. Lamentablemente encontrar ese óptimo es una tarea descomunalmente complicada.

Una forma practica de implementar esto es la siguiente: el usuario fija ***di*** y ***fi***, se entrena el clasificador utilizando Adaboost, se van agregando características hasta alcanzar la performance requerida, se testea el sistema con un conjunto de validación, si la tasa de falsos positivos no es alcanzada, se agrega otra etapa. Los ejemplos de imágenes negativas (o sea que no contienen una cara) para el entrenamiento se obtiene de recolectar todos los falsos positivos del clasificador anterior.

3.4 Un simple experimento

Para explorar las virtudes de la estructura en cascada se entrenaron dos clasificadores, uno monolítico de 200 características y otro con arquitectura de cascada con 10 etapas de 20 características cada una.

La primera etapa del clasificador en cascada se entrenó utilizando 5000 caras y 10000 sub-ventanas no-caras elegidas aleatoriamente de una base de imágenes no-caras. La segunda etapa se entrenó utilizando las mismas 5000 caras más 5000 falsos positivos del primer clasificador. Las siguientes etapas se entrenaron de la misma manera utilizando siempre como ejemplos de no-cara los falsos positivos recolectados de la etapa anterior.

El clasificador monolítico se entreno con la unión de todos los ejemplos con los que se entrenó el clasificador en cascada. Fue una forma eficiente de elegir los ejemplos de no-cara.

Si bien en términos de performance las diferencias fueron mínimas, en términos de velocidad, el clasificador en cascada fue del orden de 10 veces más rápido. En la figura 10 se muestran las curvas ROC de cada clasificador.

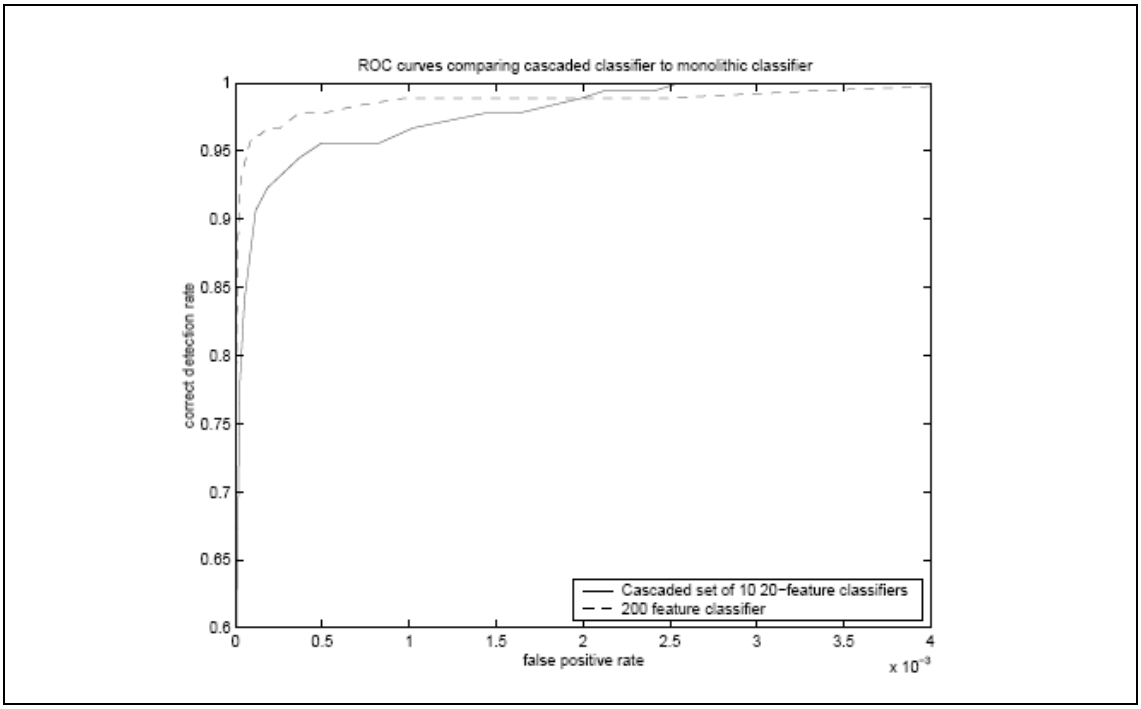


Figura 10.- Curvas ROC de clasificador monolítico de 200 características y clasificador en cascada de 10 etapas de 20 características.

3.5 Resultados

Aquí se presenta la estructura final del detector así como los datos de entrenamiento y los resultados alcanzados.

3.5.1 Datos de entrenamiento

El conjunto de entrenamiento para el detector final fue construido con 4916 imágenes de caras escaladas, centradas y etiquetadas a mano, con una resolución base de 24 x 24 píxeles.

3.5.2 Estructura del detector

La estructura final del clasificador consta de 32 etapas en cascada y un total de 4297 características.

La primera etapa del clasificador está construida con 2 características y rechaza el 60% de las no-caras mientras alcanza prácticamente un 100% de detecciones. La segunda etapa esta construida con 5 características y rechaza el 80% de las no-caras mientras alcanza un 100% de

detecciones. Luego siguen 3 etapas de 20 características cada una, dos etapas de 50 características cada una, 5 etapas de de 100 y 20 de 200 características. Ver figura 11:

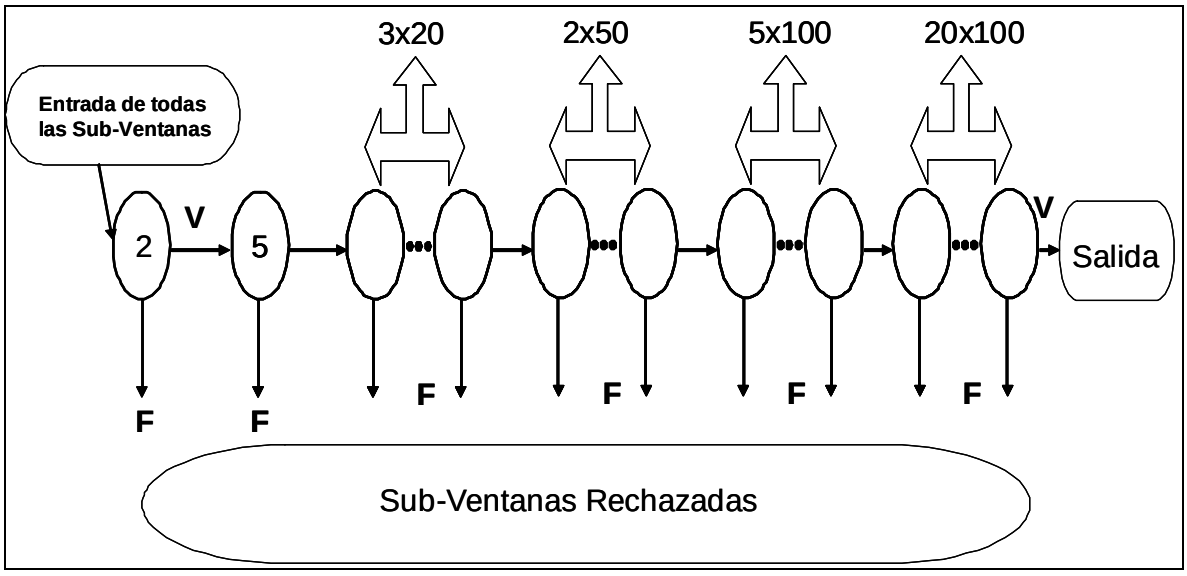


Figura 11.- Estructura final del detector de Viola-Jones

La cantidad de características en cada etapa fue conseguida mediante pruebas, ensayo y error, Luego se fueron agregando etapas hasta conseguir la performance deseada.

Figura 12.- Ejemplo de imágenes utilizadas para el entrenamiento

El entrenamiento de las primeras etapas se hizo con las 4916 caras más 10.000 sub-ventanas no-caras de también 24x24 píxeles, utilizando Adaboost, ver figura 12.

Para las siguientes etapas se utilizaron diferentes muestras de no-caras y los falsos positivos recolectados de las etapas anteriores.
Es muy importante resaltar que para el sistema completo trabajando en una AlphaStation XP900 de 466 Mhz el tiempo de entrenamiento fue del orden de semanas.

3.6 Proceso de la imagen, exploración y detección

Todas las sub-ventanas utilizadas durante el entrenamiento fueron normalizadas para minimizar los efectos de la variación de iluminación (se normaliza la varianza). La normalización durante la exploración de una imagen se pude hacer mediante la post multiplicación de las características que es más eficiente que operar sobre los propios píxeles.

La exploración se hace partiendo de una ventana de 24X24 píxeles, luego se va escalando, lo que realmente se escala es el detector y no la imagen en si, esto es gracias a que las características pueden ser calculadas en cualquier lugar de la imagen y en cualquier escala con el mismo costo computacional.
También se va haciendo una exploración a través de las distintas áreas de la imagen, esto se logra haciendo un corrimiento de Δ píxeles. Este proceso de corrimiento se ve afectado por el escalamiento, o sea si se empezó con un corrimiento de Δ píxeles, cuando escale S el corrimiento en ese factor de escala será [SΔ] donde [] es la operación de redondeo.

La elección de S y de Δ afecta significativamente tanto la performance como la velocidad del detector. Se han obtenido buenos resultados para S= 1.25 y Δ =1 o 2.

3.7 Test del sistema

El sistema fue testado utilizando la base de caras frontales del MIT+CMU, que consta de 130 imágenes con 507 caras etiquetadas. Para Δ=1 y una escala inicial =1 el número de sub-ventanas exploradas fue 75.081.800 para Δ=1.5 y una escala inicial de 1.25 el total de sub-ventanas fue de 18.901.947.

Las curvas ROC para estos test se muestran en la siguiente figura:

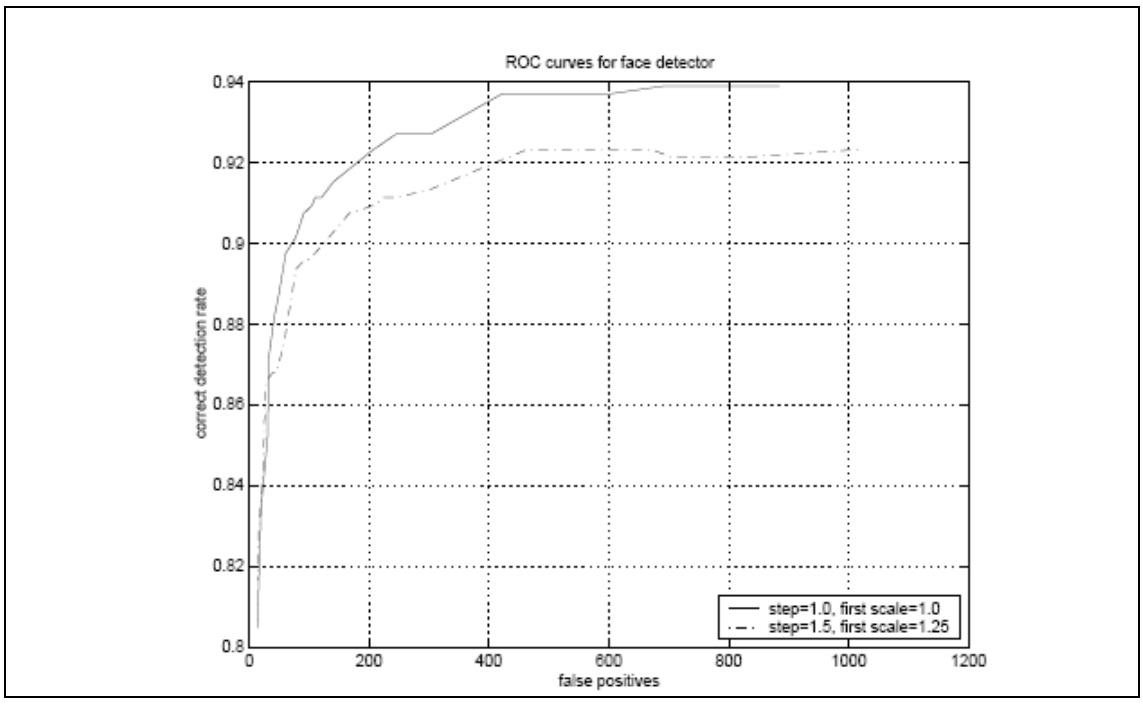


Figura 13.- Test del sistema para distintos valores de escalado y desplazamiento.

4 PARTE 3: PRUEBAS REALIZADAS

A modo de prueba se instaló un programa de reconocimiento de caras frontales similar al de Viola-Jones, a partir del código Matlab de Sreekar Krisnha, donde se implementa el OpenCV's face detector creado por Rainer Lienhart.
Si bien el programa inicial esta hecho en Matlab se debe instalar el Visual Studio .NET 2003 para poder acceder al compilador Microsoft Visual C/C++ versión 7.1, a su vez la versión de Matlab debe ser posterior a la 7.0.0 R14.
Esto es debido a que el programa principal esta hecho en C++ para a su vez poder utilizar las bibliotecas de OpenCV. A su vez el clasificador entrenado aparece como dato en un archivo Xml.

Observación: al mirar el archivo Xml, que presenta el clasificador entrenado, vemos que en realidad no es el clasificador de Viola-Jones, pues como vimos este consta de 32 etapas y el que encontramos en el Xml consta de 20 etapas. Pero mantiene la misma filosofía que el clasificador de Viola-Jones.

El programa recibe como entradas la foto a la cual se le va efectuar la detección de caras y la dirección donde se encentra el archivo .xml. que contiene el clasificador entrenado.
La salida del programa es una matriz NxM. Donde N es la cantidad de caras detectadas y M=4.
Cada fila corresponde a una cara, y las cuatro columnas representan la ubicación x e y del vértice superior derecho, el ancho y el alto de la foto.

Analizando el archivo C++ encontramos tres parámetros con los que podemos jugar, el tamaño mínimo de la imagen, el factor de escala S y la superposición de rectángulos para considerar multi-detección. Los parámetros por defecto son S=1.1, tamaño mínimo de la imagen 30 X 30 píxeles, multi-detección en 2.

4.1 Parámetros del clasificador y características de las imágenes de prueba

Se testeo el clasificador con 20 fotos digitales formato jpg. Cada imagen fue preprocesada para bajar su resolución a 640 x 480 píxeles, de manera de hacer manejable el tamaño de las mismas. Se tomaron 3 valores del factor de escala, que en principio se presentó como el

parámetro mas sensible, S=1.05, S=1.1 y S=1.25, a su vez se tomó el tamaño mínimo de imagen en 10 x10 y 30 x30 píxeles. Se hizo una modificación en el programa para medir el tiempo insumido en cada una de las pruebas.

Para poder correr estas configuraciones se modifico el código C++ y se compilo generando así cinco .mex. Luego se cargo cada uno de las fotos y se corrió el archivo Matlab llamando en cada ocasión a cada uno de los cinco .mex.

Notación

Se utilizó la siguiente notación: si la foto se llamaba nombre.jpg, la foto procesada con S=1.05 y tamaño mínimo de imagen de 10 x10 píxeles, la foto procesada se llamó nombre1051010.jpg , si se procesaba con S=1.25 y tamaño mínimo de imagen de 30 x30 píxeles , a la foto procesada se le llama npmbre1253030.jpg.

4.2 Presentación de resultados

Los resultados encontrados varían mucho dependiendo del tipo de foto, para fotos donde las caras representaban un tamaño importante dentro del total de la foto, todas las combinaciones respondían razonablemente bien, tendiendo las escalas menores a presentar mas falsos positivos.

Para fotos donde las caras están lejanas empieza a jugar un papel muy importante el tamaño mínimo inicial, cosa que es razonable, aquí la escala no jugó un papel tan importante.

En cuanto a la velocidad este programa esta muy por debajo de lo esperado para el algoritmo Viola-Jones. Para comparar velocidades se probó con fotos de tamaño 320x240 (del orden del utilizado por Viola-Jones) con escala 1.25 y tamaño mínimo de 30x 30 (la mas rápida de las combinaciones probadas), el tiempo empleado es del orden de los 0.7 s que es 10 veces superior al anunciado en el paper (corrido en un PIII Intel de 500Mhz con 256 MBites Ram).

Esto puede deberse a que el programa no esta optimizado y que Matlab no sea la mejor plataforma para ello.

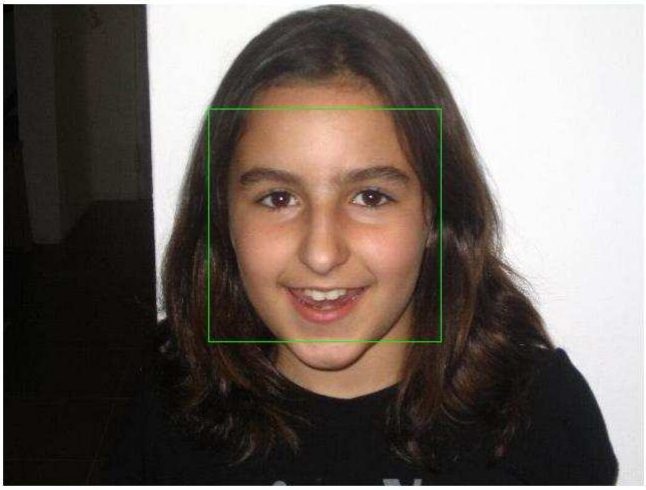
Se analizaron los falsos positivos y se encontró que algunos eran realmente razonables, mientras que no se encontró explicación para otros. Para hacer ese estudio se pasaron las fotos a escalas de grises que es como la toma el algoritmo y luego se jugó con el brillo y el contraste de cada una, intentando encontrar la cara ¨fantasma¨.

Ejemplo1

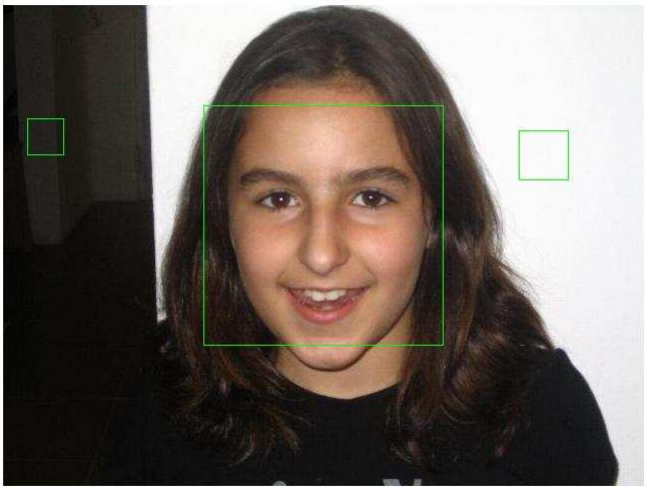
Detección con S=1.1 y T=30X30. Se obtuvo un 100% de detección con 0 falsos positivos, observar que el tamaño de las caras es relativamente grande con respecto al tamaño de la foto, el tamaño de las caras es de aproximadamente 90 x90 píxeles.

Detección con S=1.05 T=30x30, se observa un falso positivo, que puede llegar a ser interpretado como una cara (sobre todo pensando en las dos primeras características, da la sensación que no hay el entrenamiento suficiente, o que hacen falta mas etapas).

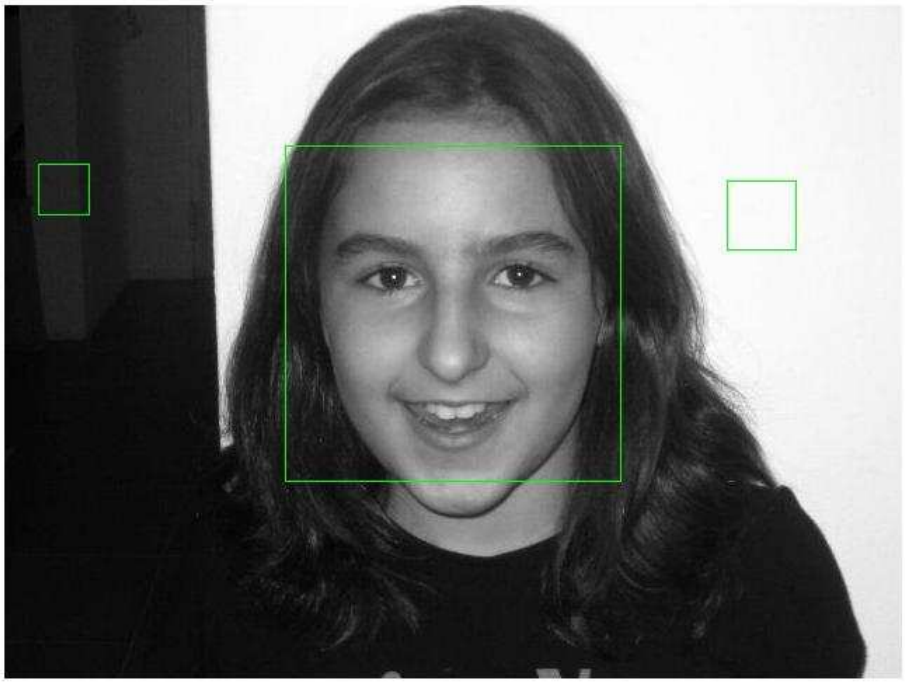
Ejemplo 2



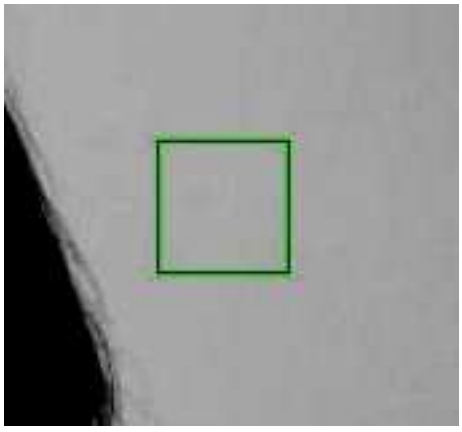
Detección con S=1.1 y T=30X30. Se obtuvo un 100% de detección con 0 falsos positivos, observar que el tamaño de las caras es relativamente grande con respecto al tamaño de la foto, el tamaño de las caras es de aproximadamente 220 x220 píxeles



Detección con S=1.05 T=30x30, se observan dos falsos positivos, que no pueden explicarse. (Da más la sensación de algún tipo de error en la implementación mas que de falta de etapas para afinar y descartar casos más difíciles)



Versión en escala de grises de la foto anterior, misma que ingresa en el clasificador, donde a pesar del cambio en el contraste y brillo no se puede explicar la aparición de los casos de falsa detección.



Ampliación del falso positivo con cambio en contraste y brillo.

Ejemplo 3

Detección con S=1.1 y T=30X30. Se obtuvo un 75% de detección con 0 falsos positivos, observar que el tamaño de las caras es relativamente pequeño respecto al tamaño de la foto, por lo que principalmente debido al tamaño se hace imposible detectarlas, consideramos que se detectaron 3 de 4 posibles.

Para intentar detectar las caras mas chicas se intenta con S=1.05 y tamaño mínimo 10x10. Se mejora el nivel de detección a 4/4 posibles. Aparece sin embargo dos falsos positivos. El primero a la izquierda es sorprendentemente real y tiene sentido, el segundo no tanto.

Ampliación del primer falso positivo, aquí es fácil adivinar dos ojos y una nariz.

Ejemplo 4

En este ejemplo se hace más evidente la relación con el tamaño mínimo de detección, cambiando la escala y el tamaño mínimo de detección, cambia en forma drástica la performance del detector.

Detección con S=1.1 y T=30X30. Se obtuvo un 0% de detección con 1 falsos positivos, observar que el tamaño de las caras es relativamente pequeño respecto al tamaño de la foto, por lo que al tener T=30x30 hace que no sea posible detectarlas.

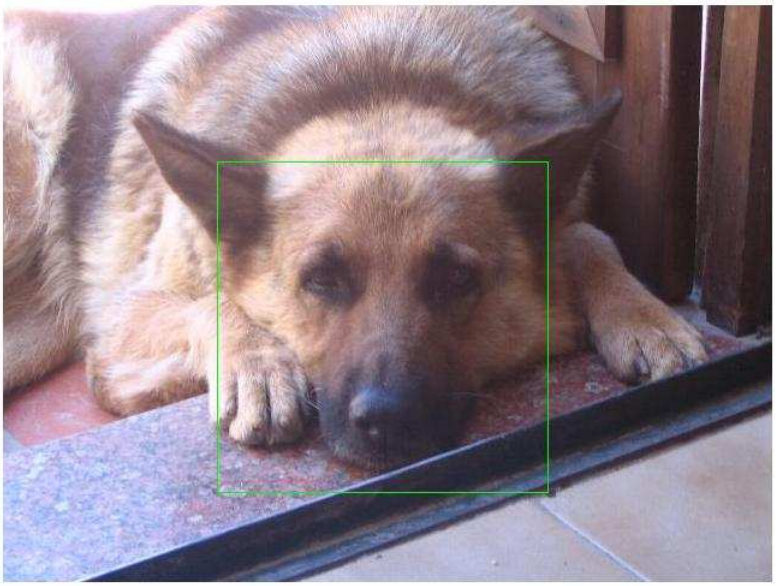
Detección con S=1.1 y T=10X10. Se obtuvo un 13/21 de detección con 1 falsos positivos, observar que el solo cambio del parámetro de tamaño mínimo resulta en un cambio de la performance muy alto.

Detección con S=1.05 T=10x10. La performance mejora subiendo la detección a 17/21. Se nota la aparición de un nuevo falso positivo que también puede ser interpretado como una cara.

Con un poco de imaginación podemos entenderlo como un rostro.

Ejemplo 5

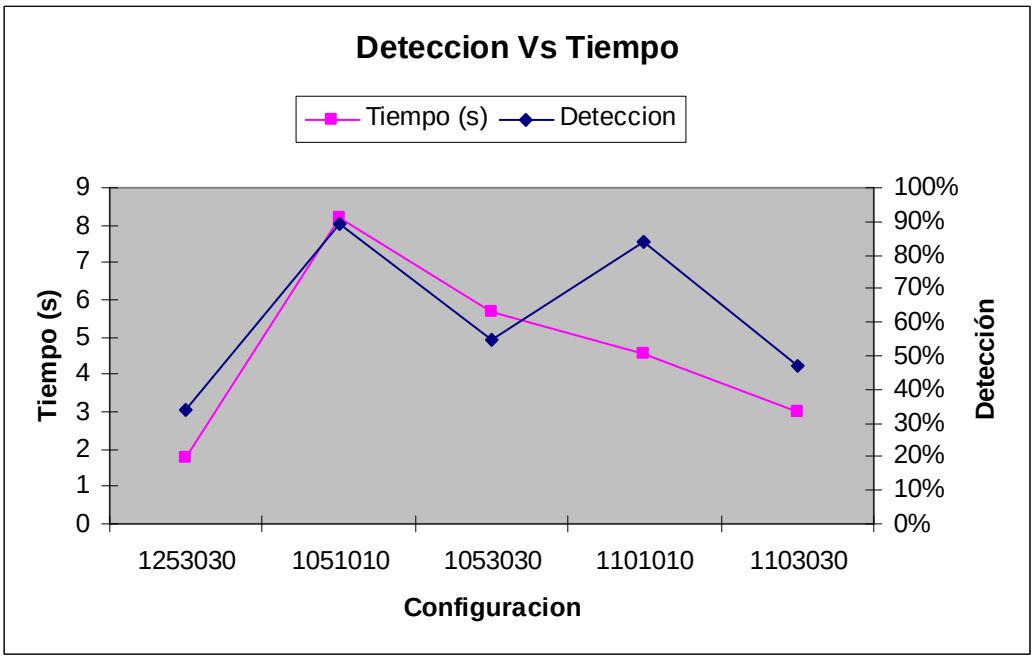
Con este ejemplo podríamos concluir que hace falta el agregado de más etapas y volver a la fase de entrenamiento.

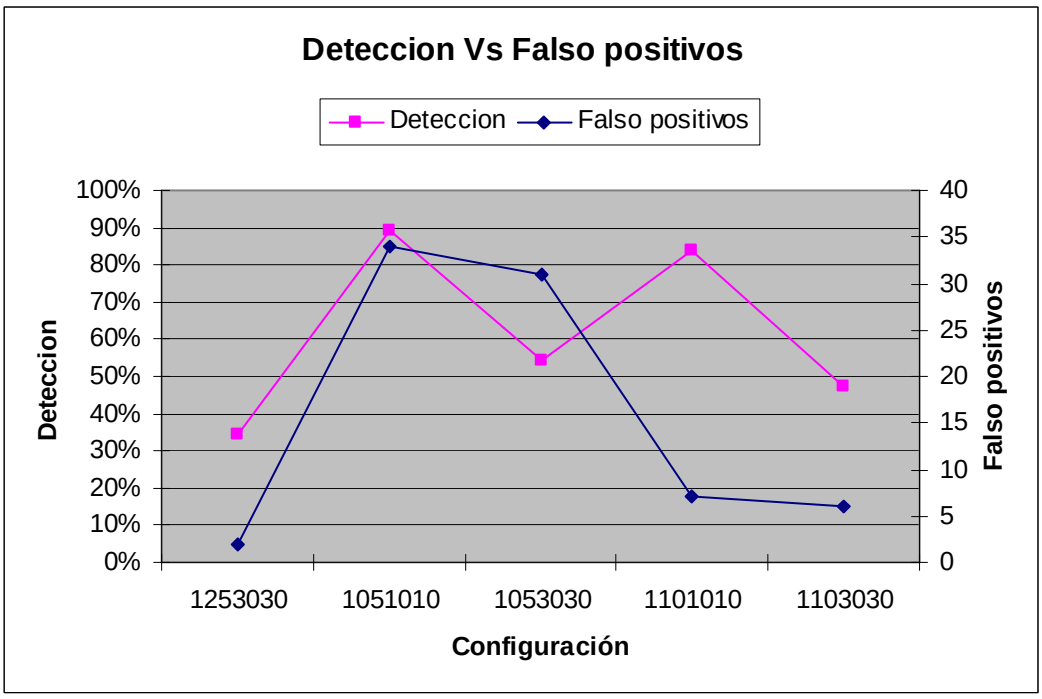


Detección con S=1.1 y T=30x30, está corrida con los parámetros que daban el menor numero de falsos positivos, evidencia las carencias de este detector.

Cuadro con valores promedio para cada una de las configuraciones y las 20 fotos elegidas:

Configuracion	Tiempo (s)	Deteccion	Detectadas	Total	Falso positivos
1253030	1,752	34,15%	42	123	2
1051010	8,203	89,43%	110	123	34
1053030	5,691	54,47%	67	123	31
1101010	4,531	83,74%	103	123	7
1103030	2,985	47,15%	58	123	6





4.3 Conclusiones sobre la implementación probada

Para fotos donde las caras aparecen bien definidas y de tamaño relativamente grande, el detector se comporta en forma aceptable con sus parámetros por defecto S=1.1 y T=30x30. Cuando las caras son relativamente chicas se debe bajar la escala y el tamaño mínimo para obtener detección, al efectuar estas modificaciones aparecen falsos positivos que evidencia la falta de mas etapas y nuevo entrenamiento.

Referencias

1.- V. G. Vovk, A game of prediction with expert advine, in “Proceedings of the Eighth Annual Conference on Computational Learnig Theory”,1995

2.- K. Sung and T. Poggio. Example-based learning for view-based face detection. In IEEE Patt. Anal.Mach. Intell., volume 20, pages 39–51, 1998.

3.- H. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. In *IEEE Patt. Anal. Mach. Intell.*, volume 20, pages 22–38, 1998.

4.- H. Schneiderman and T. Kanade. A statistical method for 3D object detection applied to faces and cars. In *International Conference on Computer Vision*, 2000.

5.- D. Roth, M. Yang, and N. Ahuja. A snowbased face detector. In *Neural Information Processing 12*, 2000.

6.- Y. Amit, D. Geman, and K. Wilder. Joint induction of shape features and tree classifiers, 1997.

7.- C. Papageorgiou,M. Oren, and T. Poggio. A general framework for object detection. In *International Conference on Computer Vision*, 1998.