

Cuestiones parte 1

1. ¿Qué función concreta es la encargada de realizar la detección?

La función dentro del código empleada expresamente para detectar las caras es “detectMultiScale”. Esta función se aplica sobre la variable “cascade”, la cual hemos pasado por un modelo de detección de caras ya entrenado.

```
# Carga del modelo de deteccion (previamente entrenado)
cascade_fn = 'haarcascades/haarcascade_frontalface_alt.xml'
cascade = cv2.CascadeClassifier(cascade_fn)

rects = cascade.detectMultiScale(
    img_gray,
    scaleFactor=1.1,
    minNeighbors=5,
    minSize=(30, 30),
    flags = cv2.CASCADE_SCALE_IMAGE
)
```

Captura [1]

2. ¿Qué parámetros recibe?

Esta función admite los siguientes parámetros, aunque hemos usado 5 de ellos:

Image: Almacena la imagen sobre la que vamos a buscar los objetos. En nuestro caso, los rostros.

Objects: Es una lista que almacena rectángulos. Cada uno de ellos contiene el objeto detectado, caras en nuestro caso. No lo hemos usado en la práctica.

ScaleFactor: Este parámetro especifica cuanto se reduce la imagen en cada escalado de imagen.

MinNeighbors: Este parámetro especifica cuántos vecinos debe tener cada rectángulo candidato para conservarlo. Es decir, define el requisito para conservar o rechazar un posible rectángulo.

Flags: variable para aplicar condiciones adicionales.

MinSize: Es el tamaño de objeto mínimo posible. Los objetos más pequeños a este valor se ignoran.

MaxSize: Similar al parámetro anterior, es el tamaño máximo de objeto posible. Los objetos más grandes que eso se ignoran. Si maxSize == minSize el modelo se evalúa en una sola escala. No lo hemos definido en la práctica.

3. ¿Qué modelo de detector pre-entrenado se está utilizando?

Tal y como se puede observar en la captura 1, el modelo pre-entrenado utilizado es el “haarcascade_frontalface_alt”. No obstante, a lo largo de la práctica lo he ido cambiando según convenía.

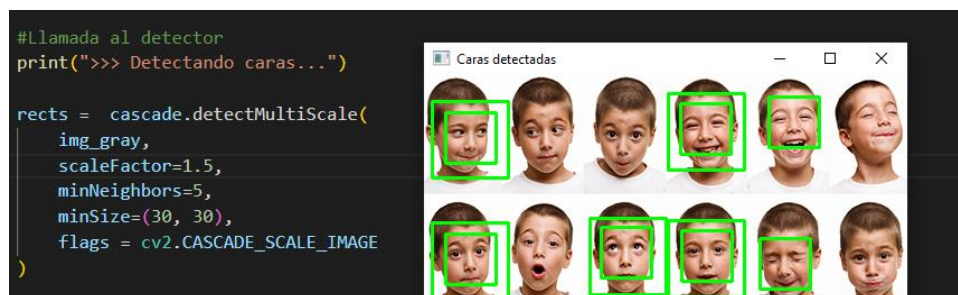
4. ¿Por qué cree que se realiza una ecualización de la imagen y una conversión de la misma a escala de gris?

El motivo por el cual interesa ecualizar y convertir la imagen a gris, es primero mejorar notablemente la calidad de la imagen con la ecualización, y segundo pasarlo a un espacio de colores cómodo para aplicar el método de detección de caras.

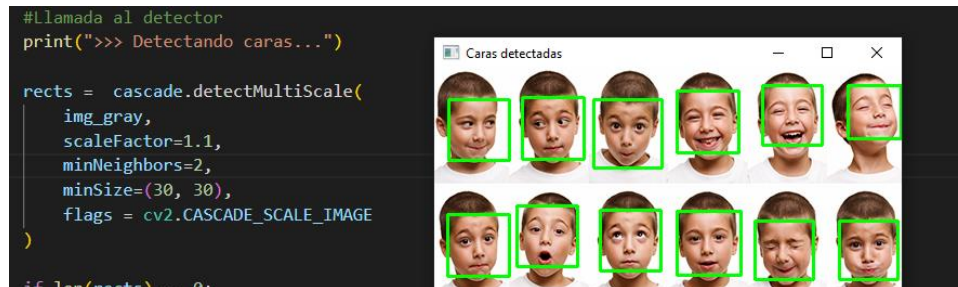
Como ya hemos comprobado en sesiones anteriores, elementos como los bordes o los cambios de intensidad se pueden detectar y manipular de forma eficiente en escala de grises. Por lo que parece razonable que el modelo trabaje sobre imágenes en gris.

Modificación de parámetros

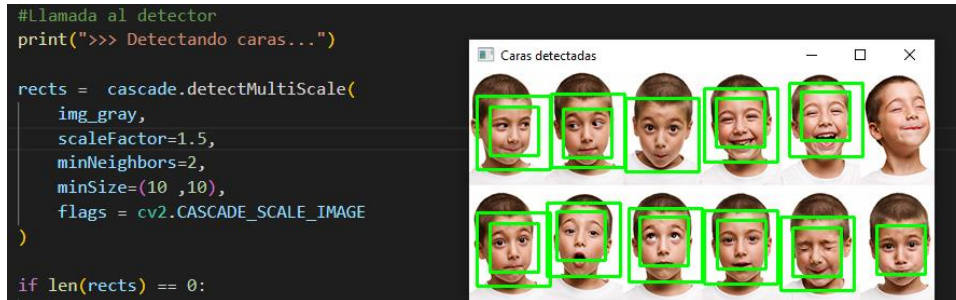
Si sobre test1.jpg aplicamos un “scaleFactor” de valor 1.5, obtenemos el siguiente resultado:



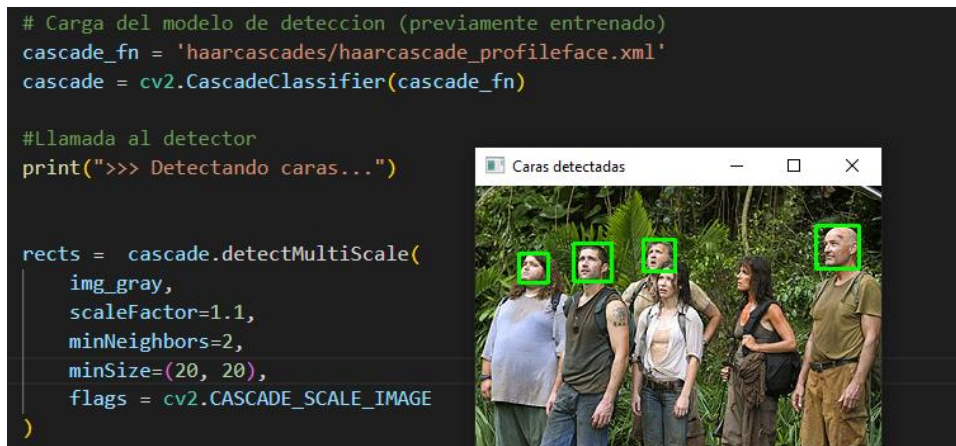
La explicación de esto es que al haber aumentado el factor de reducción y dejando constante los demás parámetros, hemos establecido las condiciones propicias para la detección de múltiples caras sobre una misma. Esto podemos corregirlo si jugamos con los otros parámetros.



Sobre la misma imagen varío el número mínimo de vecinos, cosa que nos permite detectar las caras que antes no nos detectaba.



Si ahora reduzco el valor “minSize”, de nuevo tenemos más de una detección sobre la misma cara.



Esta imagen en el modelo anterior no devuelve caras, pero si lo pasamos por el “haarcascade_profileface” sí que tenemos valores detectados.

Velocidad del código

Para poder valorar la rapidez de la detección, debemos introducir una variable para medir expresamente el tiempo de ejecución.

Para ello, voy a crear una variable global llamada "t", la cual voy a igualar a la función de openCV "cv2.getTickCount()". Esta función se emplea para medir los ticks desde la llamada a la función hasta el punto deseado. La propia documentación comenta que se emplea para establecer valores o semillas RNG, aunque también puede valer para calcular el tiempo de ejecución del programa.

Una vez seleccionada la función, vamos a medir el tiempo desde antes de pasar la imagen a escala de grises, hasta el punto en que la función nos devuelve el resultado mediante la variable "img_out"

Inicio la variable t

```
def detector(img_color):  
    global t  
    print(">>> Cargando imagen...")  
    t = cv2.getTickCount()
```

Captura [2]

Fin variable t

```
    flags = cv2.CASCADE_SCALE_IMAGE  
    )  
  
    if len(rects) == 0:  
        t = (cv2.getTickCount() - t)/cv2.getTickCount()  
        return img_color  
    print(rects)  
    rects[:, 2:] += rects[:, :2]  
    print("El numero de caras detectadas es:", len(rects))  
    img_out = img_color.copy()  
    draw_rects(img_out, rects, (0, 255, 0))  
    t = (cv2.getTickCount() - t)/cv2.getTickCount()  
    return img_out
```

Captura [3]

Fijándonos en captura 3, debemos hacer una serie de cuentas para obtener realmente el tiempo transcurrido. Esto se debe a que "cv2.getTickCount()" mide el número de ticks hasta el momento en que se llama la función. Concretamente hacemos lo siguiente:

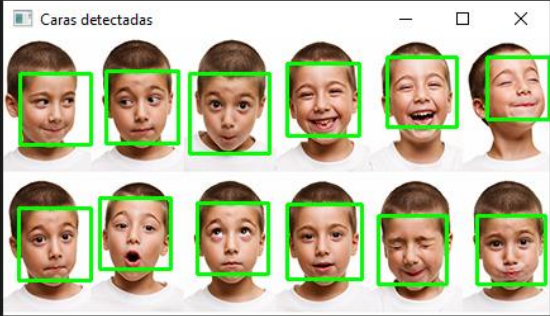
$$t = (cv2.getTickCount() - t) / cv2.getTickFrequency()$$

La explicación de esta operación es sencilla. Debemos restar el número de ticks actual con el inicial, lo cual nos proporciona la cantidad de ticks ocurridos en el bloque de código deseado. A continuación, como queremos que la medida sea en segundos, debemos restar la cantidad anterior entre la frecuencia de ticks, lo cual nos devuelve segundos como unidades.

Ejemplos de ejecución:

Primera medida: me detecta 12 caras en 0.145 segundos.

```
PS C:\Users\aitor\OneDrive\Desktop\Cuarto_de_carrera\VISION\BLOQUE 2\Sesion 1> & C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe eDrive/Desktop/Cuarto_de_carrera/VISION/BLOQUE 2/Sesion 1/dect_demo_timer.py"
>>> Cargando imagen...
(218, 432)
>>> Detectando caras...
[[224 19 57 57]
 [ 76 125 56 56]
 [ 12 133 57 57]
 [ 13 27 56 56]
 [303 14 55 55]
 [ 81 25 57 57]
 [296 139 54 54]
 [153 129 56 56]
 [374 139 54 54]
 [224 130 59 59]
 [382 14 49 49]
 [147 27 63 63]]
El numero de caras detectadas es: 12
El tiempo de ejecución han sido: 0.1458546 segundos
```



Segunda medida: Como la cara está de lado, y el modelo busca caras frontales, no la detecta. Aquí podemos ver la rapidez del código comparando el valor anterior con este, ya que este segundo tiempo se salta muchos cálculos empleados para la detección de la imagen, dando como resultado un tiempo de ejecución muy bajo.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.


Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/ps

PS C:\Users\aitor\OneDrive\Desktop\Cuarto_de_carrera\VISION\BLOQUE 2\Sesion 1> & C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe eDrive/Desktop/Cuarto_de_carrera/VISION/BLOQUE 2/Sesion 1/dect_demo_timer.py"
>>> Cargando imagen...
(287, 460)
>>> Detectando caras...
El tiempo de ejecución han sido: 1.7998987748302738e-07 segundos
```



Tercera medida: En este caso, detecta 4 de las 6 caras, y tarda más que en el caso de las 12. Esto se debe a que, para esta medida en concreto, he modificado el modelo a "haarcascade_profileface", y el minSize a (20x20).

```
PS C:\Users\aitor\OneDrive\Desktop\Cuarto_de_carrera\VISION\BLOQUE 2\Sesion 1> & C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe C:/Users/aitor/AppData/Local/Programs/Python/Python38-32/Scripts/python.exe eDrive/Desktop/Cuarto_de_carrera/VISION/BLOQUE 2/Sesion 1/dect_demo_timer.py"
>>> Cargando imagen...
(240, 320)
>>> Detectando caras...
[[132 42 25 25]
 [ 77 45 30 30]
 [ 34 53 23 23]
 [268 31 34 34]]
El numero de caras detectadas es: 4
El tiempo de ejecución han sido: 0.1650864 segundos
```



Ejercicio 2

Parte 1: Detección de caras en tiempo real.

Para poder detectar caras sobre un flujo de video, debemos modificar el script para poder tomar los frames del mismo.

```
#Creamos el objeto capturador de video
cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)

while(True):
    # Capturamos frame a frame
    ret, frame = cap.read()
    # Trabajamos con los frames
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Parte 2: Trabajamos frame a frame.

Una vez establecida la entrada de frames, debemos ir pasando uno a uno por la función de detección de frames.

```
while(True):
    # Captura frame a frame
    ret, frame = cap.read()
    # Trabajamos con los frames
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    copia_frame = frame.copy()
    # Llamo al detector
    img_out = detector(copia_frame)

    # Muestro el resultado
    cv2.imshow('resultado_deteccion', img_out)
```

Donde en lugar de trabajar con el frame actual, trabajo con una copia.

Parte 3: Cada cara detectada irá con un color diferente.

Este es el verdadero reto de este ejercicio, al cual, para darle cierto dinamismo, voy a agregar que los colores sean completamente aleatorios.

Para solucionar el problema de que los colores sean completamente aleatorios, y previendo que no tendremos más de 20 caras a la vez, sino un número mucho menor, he creado 20 colores aleatorios.

```
#Creamos aleatoriamente 20 caras
for i in range(20):
    color = tuple(np.random.choice(range(256), size=3))
    colores.append(color)
```

Este bucle “for” almacena en “color” una tupla de tamaño 3. Donde cada valor de la tupla está en el rango 0-255. Es decir, tenemos un código RGB aleatorio.

A continuación, agrega al final del array global “colores” el color creado, y hace esto 20 veces. De esta forma, tengo 20 colores completamente aleatorios accesibles desde cualquier parte del código. Sin embargo, no tengo un sistema que me asegure que esos 20 colores no sean repetidos. Aunque con un bucle “while” podríamos hacerlo, no he visto que sea necesario al ser una probabilidad muy baja.

Solucionado el problema de los colores, debemos establecer un color a cada cara detectada. Para esto, nos vamos a basar en la variable proporcionada “rects”, la cual almacena una lista que guarda información de todas las caras detectadas.

Con esto en mente, podemos hacer un bucle “for” que realice una iteración por cada cara detectada. A estas caras, se les irá asignando en orden un valor del array “colores”. Para medir las iteraciones, empleo la variable “vueltas”, que emplearé para seleccionar la posición del color aleatorio.

```
for (x, y, w, h) in rects:
    cv2.rectangle(img_color, (x, y), (x + w, y + h), (int(colores[vueltas][0]),int(colores[vueltas][1]),
                                                         int(colores[vueltas][2])), 2)
    vueltas+=1

return img_color
```

Es importante comentar que el recuadro detectado sobre cada cara puede cambiar de color momentáneamente en el caso en que se detecten menos caras. Esto se explica porque los colores se asignan en orden según las caras detectadas, por lo que, si tengo 3 caras, y la mía es la tercera, se me asignará el color 3. Si en este mismo caso, una de las otras 2 caras se deja de detectar, se me va a asignar otro color, ya que ahora sólo hay 2 caras y por tanto asignamos 2 colores. Por lo que mi cara heredará uno de los colores previamente asignados.

Ejercicio 3

Para proporcionar el efecto pixelado a nuestra cara, me he apoyado en la función “draw_rects”. El motivo de esto es que por parámetros tenemos los píxeles en los cuales se dibuja el rectángulo en torno a la cara detectada. O lo que es lo mismo, las coordenadas de la zona que quiero pixelar.

En la función “draw_rects” primero establezco el recuadro verde, luego desenfoco la zona de interés (ROI a partir de ahora), y llamo a la función “pixelado”.

```
#Funcion para pintar los bounding boxes detectados
def draw_rects(img, rects, color):
    for x1, y1, x2, y2 in rects:
        cv2.rectangle(img, (x1, y1), (x2, y2), color, 2)

        # Difumino con un desenfoque gaussiano la zona de la cara
        roi = img[y1:y2, x1:x2]
        roi = cv2.GaussianBlur(roi, (23, 23), 30)

        # Llamo a la función pixelate_face
        roi_pixelado = pixelado (roi,rects,tam_bloque)

        # Superpongo en la imagen a la original la roi con desenfoque y pixelado
        img[y1:y1+roi_pixelado.shape[0], x1:x1+roi_pixelado.shape[1]] = roi_pixelado
```

La función “pixelado” trabaja sobre nuestra ROI desenfocada. Esta función hace lo siguiente:

1. Dividimos la imagen en bloques de (tam_bloque x tam_bloque).
2. Utilizo un bucle que recorre los bloques tanto en la dirección X como en Y.
3. Extraemos una ROI sobre la cual hago la media en sus componentes RGB.
4. Superponemos esta nueva imagen pixelada a la dada por parámetro.

Foto de la función

```
def pixelado(img, rects,tam_bloque):
    for x1, y1 in rects:
        # Dividimos la imagen en bloques de tam_bloque x tam_bloque
        (h,w) = img.shape[:2]
        pasos_X = np.linspace(0, w, tam_bloque + 1, dtype="int")
        pasos_Y = np.linspace(0, h, tam_bloque + 1, dtype="int")
        # Bucle que recorre los bloques tanto en la dirección x como en y
        for i in range(1, len(pasos_Y)):
            for j in range(1, len(pasos_X)):
                inicio_X = pasos_X[j - 1]
                inicio_Y = pasos_Y[i - 1]
                fin_X = pasos_X[j]
                fin_Y = pasos_Y[i]
                # Extraemos una ROI sobre la cual hacer "mean" en sus componentes RGB
                roi = img[inicio_Y:fin_Y, inicio_X:fin_X]
                (B, G, R) = [int(x) for x in cv2.mean(roi)[:3]]
                cv2.rectangle(img, (inicio_X, inicio_Y), (fin_X, fin_Y),
                              (B, G, R), -1)
            # Superponemos esta nueva imagen pixelada a la dada por parámetro
            img[y1:y1+roi.shape[0], x1:x1+roi.shape[1]] = roi
        # Devolvemos la imagen
    return img
```