Detección de rostros usando técnicas de preprocesamiento y paralelización

Luis A. Selis,[1]

[1]Facultad de Ciencias
Universidad Nacional de Ingeniería
Av. Túpac Amaru 210, Lima, Perú

Introducción

La detección de rostros es un problema fundamental en la visión computacional, con aplicaciones en reconocimiento facial, vigilancia y análisis de imágenes. Existen diversos enfoques para abordar este problema, que van desde métodos clásicos hasta modelos basados en redes neuronales profundas. No obstante, estos métodos presentan limitaciones tanto en precisión como en tiempo de ejecución, especialmente cuando se enfrentan a imágenes en condiciones no controladas, como variaciones de iluminación, escala o calidad.

El objetivo de este proyecto es identificar el tiempo que toma cada método de detección de rostros, analizar las causas de sus fallos, comparar su rendimiento y proponer mejoras mediante técnicas de preprocesamiento, paralelización y modelos híbridos, haciendo uso de conocimientos previos en computación de alto desempeño y programación paralela.

Trabajos relacionados

Entre los métodos clásicos de detección de rostros se encuentran los clasificadores en cascada basados en características HAAR, los cuales destacan por su bajo costo computacional, aunque presentan dificultades para detectar rostros pequeños o imágenes con baja calidad. Métodos más recientes como MTCNN, DLIB y detectores basados en redes neuronales profundas (DNN) han demostrado una mayor precisión, a costa de un mayor tiempo de ejecución.

Por otro lado, los detectores de una sola pasada, como el Single Shot MultiBox Detector (SSD), permiten realizar predicciones de localización y clasificación de múltiples objetos en una sola evaluación de la red. Cuando estos detectores se combinan con redes convolucionales profundas como VGG16, preentrenadas en ImageNet, se obtiene una arquitectura eficiente y precisa para tareas de detección. Asimismo, el paradigma de paralelización *embarrassingly parallel* ha sido ampliamente utilizado para acelerar algoritmos de procesamiento de imágenes en plataformas multinúcleo, siempre que las tareas sean independientes entre sí.

Metodología

Para la evaluación se utilizaron distintos métodos de detección de rostros: HAAR, MTCNN, DLIB, DNN y una red neuronal convolucional (CNN) basada en SSD. Se trabajó inicialmente con un dataset original y posteriormente con un dataset extendido, al cual se le añadieron diez imágenes capturadas en entornos no controlados.

Se aplicaron técnicas de preprocesamiento de imágenes, tales como el reescalado para mejorar la detección de rostros pequeños y el ajuste de iluminación mediante aclarado u oscurecimiento de las imágenes. En el caso de los métodos MTCNN y CNN, se implementó paralelización utilizando el paradigma “embarrassingly parallel”, donde cada imagen es procesada de manera independiente por múltiples hilos de ejecución.

Este paradigma está compuesto por tres módulos principales: emitter, workers y collector. El emitter se encarga de distribuir cada elemento del flujo de entrada a uno de los workers de acuerdo con una política determinada, asegurando que cada elemento sea asignado a un único worker. Cada worker aplica de manera independiente la función F sobre los datos recibidos y envía el resultado al collector, el cual reúne las salidas y las transmite al flujo de salida. Para que el paradigma farm sea aplicable, la función F debe ser una función pura, es decir, no debe depender de un estado interno compartido. Este enfoque resulta adecuado para aplicaciones en las que el orden de los resultados no es crítico, ya que la velocidad de procesamiento puede variar entre los distintos workers.

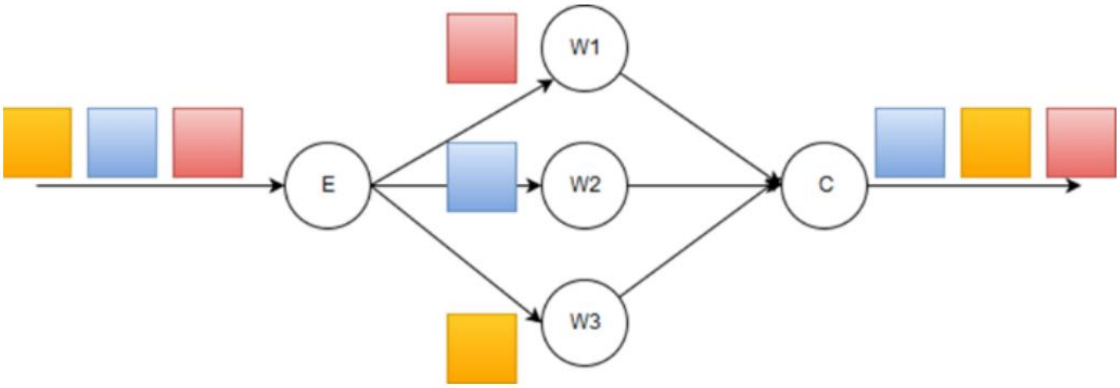


Figura 1. Ilustración del funcionamiento de paradigma “embarrassingly parallel” donde se paralelizan varias tareas y el tiempo de ejecución lo determina la tarea que tome más tiempo.

Para la CNN se utilizó una arquitectura basada en VGG16 como backbone de un Single Shot MultiBox Detector (SSD). La red VGG16, preentrenada en ImageNet, actúa como extractor de características profundas, mientras que el SSD añade capas adicionales para la predicción de cajas delimitadoras y probabilidades de detección. Las imágenes se redimensionaron a un tamaño fijo de 360 × 360 píxeles y se normalizaron según los valores promedio de VGG16. La red fue paralelizada usando cuatro hilos.

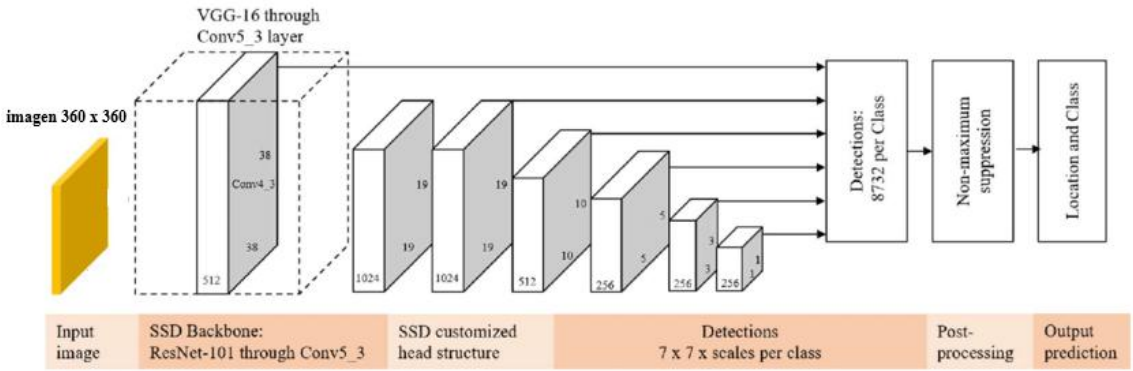


Figura 2. Estructura de la CNN usada, usa una VGG-16 como backbone de un Single Shot MultiBox Detector (SSD).

Algo que se agrego fue el threshold para la detección de rostros, esta red CNN devuelve un porcentaje de probabilidad de detectar un rostro y se notó que en algunas imágenes esta probabilidad era baja incluso para el rostro más notorio mientras que en otras imágenes usar un bajo threshold daba falso positivos

Para solucionar esto se usa un threshold dinámico definido por la ecuación:

$$\text{threshold} = \max(0.2, p^4/1.6)$$

Donde max es una función de maximización y p es la máxima probabilidad de un rostro detectado en la imagen analizada. Este umbral dinámico permite reducir falsos positivos en imágenes con baja confianza y mejorar la robustez del detector.

Resultados

Evaluación inicial con el dataset original

Tiempo total: 1.94 segundos

Tiempo de solo HAAR: 0.28 segundos

Tiempo de solo MTCNN: 1.73 segundos

Tiempo de solo DLIB: 0.76 segundos

Tiempo de solo DNN: 0.33 segundos

HAAR muestra 10 falsos negativos y 1 falso positivo

MTCNN no muestra ningún falso negativo o falso positivo

DLIB muestra 6 falsos negativos y ningún falso positivo

DNN muestra 3 falsos negativos y ningún falso positivo, la mayoría de los errores son en imágenes blanco y negro

Fortalecimiento de debilidades y código modificado

Se usaron técnicas de preprocesamiento para mejorar los resultados tal como reescala la imagen para mejorar la detección de rostros pequeños y aclarar u oscurecer la imagen. En el caso de MTCNN ya que este método era muy preciso de por si se usó técnicas de paralelización para mejorar el tiempo de ejecución

Rendimiento del código Modificado con el dateset original:

Tiempo de solo HAAR: 0.31 segundos

Tiempo de solo MTCNN: 0.7 segundos

Tiempo de solo DLIB: 1.72 segundos

Tiempo de solo DNN: 0.41 segundos

Tiempo de solo el CNN: 0.29 segundos

HAAR muestra 8 falsos negativos y 1 falso positivo

MTCNN no muestra ningún falso negativo o falso positivo

DLIB muestra 5 falsos negativos y ningún falso positivo

DNN muestra 1 falsos negativos y ningún falso positivo, la mayoría de los errores son en imágenes blanco y negro

CNN muestra 1 falsos negativos y ningún falso positivo

Nuevo código para HAAR:

```
from mtcnn.mtcnn import MTCNN
import cv2
import dlib
import numpy as np
import os
import time # LS
start_time = time.time() # LS
```

```
classifier2 =
cv2.CascadeClassifier('models/haarcascade_frontalface2.xml')
faces_dir = os.path.join(os.path.dirname(__file__), 'faces') # LS
images = os.listdir(faces_dir) #LS
os.makedirs('faces/haar')
for image in images:
    img_path = os.path.join(faces_dir, image) #LS
    img = cv2.imread(img_path) # LS
    print("Leyendo:", img_path, "->", "OK" if img is not None else
"FALLO") # LS
    if img is None: # LS
        continue #LS
    height, width = img.shape[:2]
    img1 = img.copy()
    img2 = img.copy()
    img3 = img.copy()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    gamma = 2.4  # cambio
    lookUpTable = np.array([((i / 255.0) ** gamma) * 255 for i in
np.arange(0, 256)]).astype("uint8") #cambio
    img = cv2.LUT(img, lookUpTable) #cambio
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # cambio
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))  #
cambio: más ligero y local
    gray = clahe.apply(gray)
    scaled = cv2.resize(gray, None, fx=1.5, fy=1.5,
interpolation=cv2.INTER_LINEAR)
    f_scaled = classifier2.detectMultiScale(scaled, minNeighbors=4)  #
cambio
    faces4 = [(int(x/1.5), int(y/1.5), int(w/1.5), int(h/1.5)) for (x,
y, w, h) in f_scaled]

    #HAAR
    for result in faces4:
        x, y, w, h = result
        x1, y1 = x + w, y + h
        cv2.rectangle(img3, (x, y), (x1, y1), (0, 0, 255), 2)

    cv2.imwrite(os.path.join('faces', 'haar', image), img3)
    cv2.destroyAllWindows()

end_time = time.time() #LS
execution_time = end_time - start_time #LS
print(f"\nTiempo total de ejecución: {execution_time:.2f} segundos")
#LS
```

Nuevo código para MTCNN:

```
from mtcnn.mtcnn import MTCNN
import cv2
import dlib
import numpy as np
import os
import time # LS
from concurrent.futures import ThreadPoolExecutor, as_completed  # LS
```

```
start_time = time.time() # LS

detector1 = MTCNN()
faces_dir = os.path.join(os.path.dirname(__file__), 'faces') # LS
images = os.listdir(faces_dir) #LS
os.makedirs('faces/mtcnn')
num_threads = 4  # LS
def detect_faces(img):
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return detector1.detect_faces(img_rgb)
imgs = []
img_names = []
for image in images:
    img_path = os.path.join(faces_dir, image)
    img = cv2.imread(img_path)
    print("Leyendo:", img_path, "->", "OK" if img is not None else
"FALLO")
    if img is not None:
        imgs.append(img)
        img_names.append(image)
with ThreadPoolExecutor(max_workers=num_threads) as executor:
    futures = [executor.submit(detect_faces, img) for img in imgs]
    results = [f.result() for f in futures]

    #MTCNN
for img, faces, name in zip(imgs, results, img_names):
    for result in faces:
        x, y, w, h = result['box']
        x1, y1 = x + w, y + h
        cv2.rectangle(img, (x, y), (x1, y1), (0, 0, 255), 2)
    cv2.imwrite(os.path.join('faces', 'mtcnn', name), img)

    cv2.imwrite(os.path.join('faces', 'mtcnn', image), img)
    cv2.destroyAllWindows()

end_time = time.time() #LS
execution_time = end_time - start_time #LS
print(f"\nTiempo total de ejecución: {execution_time:.2f} segundos")
#LS
```

Nuevo código para DLIB:

```
from mtcnn.mtcnn import MTCNN
import cv2
import dlib
import numpy as np
import os
import time # LS
start_time = time.time() # LS

detector2 = dlib.get_frontal_face_detector()
faces_dir = os.path.join(os.path.dirname(__file__), 'faces') # LS
images = os.listdir(faces_dir) #LS
os.makedirs('faces/dlib')
scale = 1.5
```

```
for image in images:
    img_path = os.path.join(faces_dir, image) #LS
    img = cv2.imread(img_path) # LS
    print("Leyendo:", img_path, "->", "OK" if img is not None else
"FALLO") # LS
    if img is None: # LS
        continue #LS
    if scale != 1.0:
        img = cv2.resize(img, None, fx=scale, fy=scale,
interpolation=cv2.INTER_LINEAR)
    height, width = img.shape[:2]
    img1 = img.copy()
    img2 = img.copy()
    img3 = img.copy()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    faces2 = detector2(gray, 2)

    #DLIB
    for result in faces2:
        x = result.left()
        y = result.top()
        x1 = result.right()
        y1 = result.bottom()
        cv2.rectangle(img1, (x, y), (x1, y1), (0, 0, 255), 2)

    cv2.imwrite(os.path.join('faces', 'dlib', image), img1)
    cv2.destroyAllWindows()

end_time = time.time() #LS
execution_time = end_time - start_time #LS
print(f"\nTiempo total de ejecución: {execution_time:.2f} segundos")
#LS
```

Nuevo código para DNN:

```
from mtcnn.mtcnn import MTCNN
import cv2
import dlib
import numpy as np
import os
import time # LS
start_time = time.time() # LS

modelFile = "models/res10_300x300_ssd_iter_140000.caffemodel"
configFile = "models/deploy.prototxt.txt"
net = cv2.dnn.readNetFromCaffe(configFile, modelFile)

faces_dir = os.path.join(os.path.dirname(__file__), 'faces') # LS
images = os.listdir(faces_dir) #LS
os.makedirs('faces/dnn')
scale = 1.0  # LS

for image in images:
    img_path = os.path.join(faces_dir, image) #LS
    img = cv2.imread(img_path) # LS
```

```
    print("Leyendo:", img_path, "->", "OK" if img is not None else
"FALLO") # LS
    if img is None: # LS
        continue #LS

    gamma = 2.0  # cambio
    lookUpTable = np.array([((i / 255.0) ** gamma) * 255
                            for i in np.arange(0,
256)]).astype("uint8")
    if len(img.shape) == 3:
        img = cv2.LUT(img, lookUpTable)
    else:
        img = cv2.LUT(cv2.cvtColor(img, cv2.COLOR_GRAY2BGR),
lookUpTable)

    height0, width0 = img.shape[:2]
    img_scaled = cv2.resize(img, None, fx=scale, fy=scale,
interpolation=cv2.INTER_LINEAR)
    height, width = img_scaled.shape[:2]
    img1 = img.copy()
    img2 = img.copy()
    img3 = img.copy()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    blob = cv2.dnn.blobFromImage(cv2.resize(img_scaled, (450, 450)),
                                 1.0, (450, 450), (104.0, 117.0,
123.0))
    net.setInput(blob)
    faces3 = net.forward()

    #OPENCV DNN
    for i in range(faces3.shape[2]):
        confidence = faces3[0, 0, i, 2]
        if confidence > 0.5:
            box = faces3[0, 0, i, 3:7] * np.array([width, height,
width, height])
            (x, y, x1, y1) = box.astype("int")
            x = int(x / scale)
            y = int(y / scale)
            x1 = int(x1 / scale)
            y1 = int(y1 / scale)
            cv2.rectangle(img2, (x, y), (x1, y1), (0, 0, 255), 2)

    img_out = cv2.resize(img, (width0, height0),
interpolation=cv2.INTER_LINEAR)
    cv2.imwrite(os.path.join('faces', 'dnn', image), img2)
    cv2.destroyAllWindows()

end_time = time.time() #LS
execution_time = end_time - start_time #LS
print(f"\nTiempo total de ejecución: {execution_time:.2f} segundos")
#LS
```

Código para CNN:

```
import cv2
```

```
import numpy as np
import os
import time
import threading
from concurrent.futures import ThreadPoolExecutor
start_time = time.time()

# Modelo
vgg_dir = os.path.join(os.path.dirname(__file__), 'VGG')
prototxt_path = os.path.join(vgg_dir, "deploy.prototxt")
weights_path  = os.path.join(vgg_dir,
"res10_300x300_ssd_iter_140000.caffemodel")
print("Verificando existencia de archivos de modelo...")
print(" prototxt:", prototxt_path, "->",
os.path.exists(prototxt_path))
print(" weights :", weights_path, "->", os.path.exists(weights_path))

# Cada hilo tendrá su propia instancia de Net
thread_local = threading.local()
def get_thread_net():
    if not hasattr(thread_local, "net"):
        print(f"[thread {threading.get_ident()}] Cargando modelo en
hilo...")
        thread_local.net = cv2.dnn.readNetFromCaffe(prototxt_path,
weights_path)
    return thread_local.net

# Directorios
faces_dir = os.path.join(os.path.dirname(__file__), 'faces')
images = sorted(os.listdir(faces_dir))
output_dir = os.path.join('faces', 'vgg')
os.makedirs(output_dir, exist_ok=True)
num_threads = 4
input_size = (360, 360)

# Función de detección
def detect_faces_vgg(img):
    net = get_thread_net()
    (h, w) = img.shape[:2]
    blob = cv2.dnn.blobFromImage(cv2.resize(img,
input_size),1.0,input_size,(104.0, 177.0, 123.0))
    net.setInput(blob)
    detections = net.forward()
    boxes = []
    confidences = []
    all_conf = [float(detections[0,0,i,2]) for i in
range(detections.shape[2])
                if not np.isnan(detections[0,0,i,2]) and
detections[0,0,i,2] > 0]
    if len(all_conf) == 0:
        return [], [], img
    max_conf = max(all_conf)
    dyn_thresh = max(0.2, max_conf*max_conf*max_conf*max_conf/ 1.6)
    for i in range(detections.shape[2]):
        conf = float(detections[0,0,i,2])
        if np.isnan(conf) or conf < dyn_thresh:
            continue
```

```
            box = detections[0,0,i,3:7]
            if np.any(np.isnan(box)):
                continue
            box = box * np.array([w, h, w, h])
            x1 = int(max(0, min(w - 1, box[0])))
            y1 = int(max(0, min(h - 1, box[1])))
            x2 = int(max(0, min(w - 1, box[2])))
            y2 = int(max(0, min(h - 1, box[3])))
            if x2 <= x1 or y2 <= y1:
                continue
            boxes.append((x1, y1, x2, y2))
            confidences.append(conf)
    return boxes, confidences, img

# Cargar imágenes
imgs = []
img_names = []
for image in images:
    img_path = os.path.join(faces_dir, image)
    img = cv2.imread(img_path)
    print("Leyendo:", img_path, "->", "OK" if img is not None else
"FALLO")
    if img is not None:
        imgs.append(img)
        img_names.append(image)

# Procesamiento paralelo
with ThreadPoolExecutor(max_workers=num_threads) as executor:
    futures = [executor.submit(detect_faces_vgg, img) for img in imgs]
    results = [f.result() for f in futures]

# Guardar resultados
for (orig, res, name) in zip(imgs, results, img_names):
    boxes, confidences, out_img = res
    out = out_img.copy()
    if len(boxes) == 0:
        print(f"[SSD-VGG] No detectó rostros en {name}")
    else:
        for (box, conf) in zip(boxes, confidences):
            x1, y1, x2, y2 = box
            cv2.rectangle(out, (x1, y1), (x2, y2), (0,0,255), 2)
            text = f"{conf*100:.1f}%"
            cv2.putText(out, text, (x1, max(15, y1-5)),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0,0,255), 1)
    cv2.imwrite(os.path.join(output_dir, name), out)
print("\nDetecciones SSD+VGG16 guardadas en:", output_dir)
end_time = time.time()
print(f"\nTiempo total de ejecución: {end_time - start_time:.2f}
segundos")
```

Evaluación con dataset extendido

Se agregaron 10 nuevas imágenes tal como en el caso anterior se usaron técnicas de preprocesamiento para mejorar los resultados tal como reescala la imagen para mejorar la detección de rostros pequeños y aclarar u oscurecer la imagen. En el caso de MTCNN ya que este

método era muy preciso de por si se usó técnicas de paralelización para mejorar el tiempo de ejecución

Rendimiento del código original con el dateset extendido:

Tiempo total: 19.66 segundos

Tiempo de solo HAAR: 1.67 segundos

Tiempo de solo MTCNN: 6.70 segundos

Tiempo de solo DLIB: 11.66 segundos

Tiempo de solo DNN: 1.56 segundos

HAAR muestra 38 falsos negativos y 5 falso positivo, tiene problemas detectando rostros pequeños

MTCNN muestra ningún falso positivo y ningún falso negativo

DLIB muestra 30 falsos negativos y 1 falso positivo, tiene problemas detectando rostros pequeños

DNN muestra 20 falsos negativos y 1 falso positivo

Rendimiento del código modificado con el dateset extendido

Tiempo de solo HAAR: 3.41 segundos

Tiempo de solo MTCNN: 3.28 segundos

Tiempo de solo DLIB: 52.80 segundos

Tiempo de solo DNN: 4.74 segundos

Tiempo de solo CNN: 1.13 segundos

HAAR muestra 29 falsos negativos y 5 falso positivo

MTCNN no muestra ningún falso negativo o falso positivo

DLIB muestra 26 falsos negativos y ningún falso positivo

DNN muestra 4 falsos negativos y 2 falso positivo

CNN muestra 7 falsos negativos y 1 falso positivo

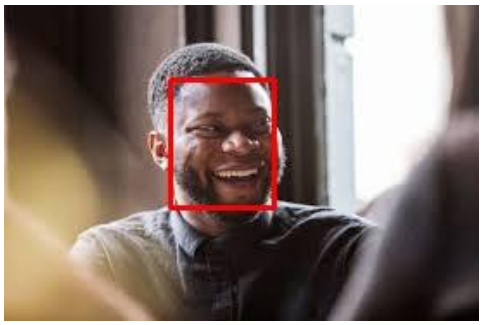
Ejemplos de imágenes agregadas:





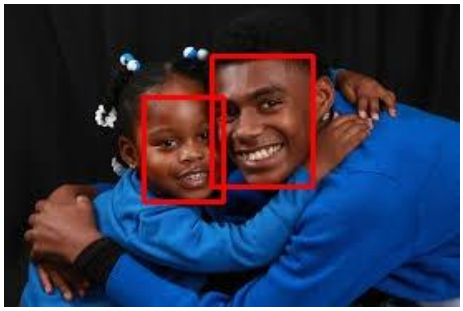








Resumen de resultados

En resumen, el cambio de iluminación y el escalamiento de las imágenes mejoraron el nivel de precisión de los modelos en especial para el modelo DNN que experimentos las mayores mejoras (Tabla 1), y la paralelización mostros mejoras notorias en el tiempo de ejecución y el modelo que encuentra los rostros más rápido es el CNN (Tabla 2).

Tabla 1. Numero de falsos negativos (FN) y falso positivos (FP) de cada uno de los programas ya sea con el dateset original y el dataset extendido

	Código original		Código modificado	
	dataset original	dataset extendido	dataset original	dataset extendido
	FN; FP	FN; FP	FN; FP	FN; FP
HAAR	10; 1	38; 5	8; 1	29, 5
MTCNN	0; 0	0; 0	0; 0	0; 0
DLIB	6; 0	30; 1	5; 0	26; 0
DNN	3; 0	20; 1	1; 0	4; 2
CNN	--	--	1; 0	7; 1

Tabla 2. Tiempo de ejecución en segundo de cada uno de los programas ya sea con el dateset original y el dataset extendido

	Código original		Código modificado	
	dataset original	dataset extendido	dataset original	dataset extendido
HAAR	0.28	1.67	0.31	3.14
MTCNN	1.73	6.70	0.70	3.28
DLIB	0.76	11.66	1.72	52.80
DNN	0.33	1.56	0.41	4.74
CNN	--	--	0.29	1.13

Conclusión

Los resultados obtenidos muestran que la aplicación de técnicas de preprocesamiento de imágenes mejora significativamente la precisión de los métodos de detección de rostros, especialmente en modelos basados en redes neuronales profundas. El modelo DNN fue el que presentó las mayores mejoras en términos de reducción de falsos negativos tras la modificación del código.

Asimismo, la paralelización mediante el paradigma "embarrassingly Parallel" permitió reducir de manera notable los tiempos de ejecución, destacando la CNN basada en SSD como el método más rápido entre los evaluados, siendo de los 5 métodos probados en este trabajo el método más rápido para la detección de rostros. En conjunto, este trabajo demuestra que la combinación de preprocesamiento, paralelización y modelos modernos constituye una estrategia efectiva para mejorar sistemas de detección de rostros en entornos no controlados.

Referencias

1. Mia, K.; Islam, T.; Assaduzzaman, M.; Akhund, T. M. N. U.; Saha, A.; Shaha, S. P.; Razzak, M. A.; Dhar, A., Parallelizing image processing algorithms for face recognition on multicore platforms. *International Journal of Advanced Computer Science and Applications* **2022,** *13* (11).
2. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.-Y.; Berg, A. C. In *SSD: Single Shot MultiBox Detector*, Computer Vision – ECCV 2016, Cham, 2016//; Leibe, B.; Matas, J.; Sebe, N.; Welling, M., Eds. Springer International Publishing: Cham, 2016; pp 21-37.
3. Qassim, H.; Verma, A.; Feinzimer, D. In *Compressed residual-VGG16 CNN model for big data places image recognition*, 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), 8-10 Jan. 2018; 2018; pp 169-175.
4. Xie, X.; Han, X.; Liao, Q.; Shi, G., Visualization and Pruning of SSD with the base network VGG16. In *Proceedings of the 2017 International Conference on Deep Learning Technologies*, Association for Computing Machinery: Chengdu, China, 2017; pp 90–94.