DEPARTAMENTO DE
INGENIERÍA ELÉCTRICA
UNIVERSIDAD DE CONCEPCIÓN

IMPLEMENTACIÓN Y EVALUACIÓN DE ALGORITMOS DE SEGUIMIENTO PARA EL PROCESAMIENTO DE VIDEO

POR

Juan Pablo Lizardi Varas

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil en Telecomunicaciones.

Profesor Guía
Gabriel Saavedra Mondaca

Concepción,
11 de julio de 2025

© 2025 Juan Pablo Lizardi Varas

© 2025 Juan Pablo Lizardi Varas
Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica del documento.

A todos los estudiantes de Telecom...

Resumen

Este trabajo presenta un estudio comparativo entre tres sistemas de detección y seguimiento de rostros en video: Haarcascade + Tracking Universidad (propuesto en esta memoria), Haar + SORT, y YOLO + SORT. El objetivo principal fue evaluar el desempeño de cada sistema en términos de precisión, estabilidad del seguimiento, velocidad de procesamiento y consumo de recursos computacionales.

El sistema desarrollado por la universidad combina la detección facial mediante Haarcascade con un módulo de seguimiento personalizado, que incorpora un filtro de paso alto temporal para estabilizar las trayectorias. Por otro lado, los sistemas Haar + SORT y YOLO + SORT integran detectores conocidos con el algoritmo de seguimiento SORT, ampliamente utilizado en aplicaciones de visión por computador en tiempo real.

Los tres sistemas fueron implementados en Python utilizando bibliotecas como OpenCV, NumPy y psutil, y evaluados sobre videos reales con personas cuyos rostros estaban claramente visibles. Se registraron métricas como tiempo promedio de procesamiento por fotograma, uso de CPU y RAM, y cantidad de cuadros procesados por segundo (FPS).

Los resultados indican que el sistema Haarcascade + Tracking Universidad ofrece buena estabilidad visual con bajo consumo de recursos, siendo adecuado para dispositivos con capacidades limitadas. Haar + SORT mostró ser eficiente pero menos robusto ante oclusiones y condiciones adversas. Finalmente, YOLO + SORT alcanzó la mayor precisión y robustez en el seguimiento, aunque con mayor demanda computacional. Se concluye que la elección del sistema más apropiado depende del escenario de uso y las restricciones del hardware disponible.

Abstract

This thesis presents a comparative study of three systems for face detection and tracking in video sequences: Haarcascade + Tracking Universidad (a custom method), Haar + SORT, and YOLO + SORT. The goal is to evaluate each system's performance in terms of detection accuracy, tracking robustness, processing speed, and resource consumption under controlled conditions.

The first system was developed in-house and combines Haarcascade-based detection with a custom tracking module and a high-pass temporal filter to smooth object trajectories. The second and third systems integrate Haarcascade or YOLOv11n-face as detectors with the SORT tracking algorithm, a widely used method based on Kalman filtering and the Hungarian assignment algorithm.

All systems were implemented in Python using OpenCV and tested with real videos involving multiple visible faces. Performance metrics such as average processing time per frame, CPU/RAM usage, and frames per second (FPS) were recorded using the `psutil` library.

The results show that Haarcascade + Tracking Universidad offers good stability and low resource usage, making it suitable for low-power applications. Haar + SORT provides a simple and lightweight solution, though it is less robust to occlusions and lighting changes. YOLO + SORT achieves the highest accuracy and tracking reliability, but with increased computational demands. The study concludes that the optimal system depends on the target scenario and hardware limitations.

Agradecimientos

Primero que nada, empiezo agradeciendo a mi familia, que siempre estuvo allí en todo momento de mi carrera, en especial a mi madre, María Varas, la cual estuvo siempre presente en mi enseñanza básica, media y universitaria. A mi hermano mayor, Luis, que siempre estuvo en los momentos donde uno quería decaer frente a las dificultades; a mi hermana y a mi sobrino, quien fue mi motivación para poder seguir adelante y continuar con esta carrera. Los amo mucho.

Agradezco también a mi profesor Gabriel Saavedra, quien siempre estuvo al tanto de mis preguntas y dificultades para poder realizar la memoria.

Agradezco a mis amigos “Los Basados”, quienes siempre estuvieron ahí conmigo desde el comienzo de esta aventura llamada Universidad, los cuales fueron mi apoyo en todo momento

Finalmente, doy las gracias a todas las personas que estuvieron apoyándome de alguna manera u otra para poder lograr lo que he conseguido.

Índice General

Resumen I

Abstract II

Agradecimientos III

Índice de Figuras VI

Índice de Tablas VIII

1. Introducción **1**

1.1. Definición del problema 2

1.2. Objetivos 3

1.2.1. Objetivo general 3

1.2.2. Objetivos específicos 3

1.3. Metodología 3

1.4. Alcances y limitaciones 4

1.4.1. Alcances 4

1.4.2. Limitaciones 5

2. Marco Teórico **6**

2.1. Introducción 6

2.1.1. Deteccion de objetos 6

2.1.1.1. YOLO (You Only Look Once) 6

2.1.1.2. Caracteristicas de Haarcascade 7

2.1.1.3. MTCNN 9

2.1.2. Tracking de objetos 11

2.1.2.1. SORT 11

2.1.2.2. DeepSORT 11
2.1.2.3. MOOSE 13

3. Desarrollo **15**
3.1. Introducción 15
3.2. Implementos utilizados 15
3.3. Haarcascade + Tracking Universidad 16
3.3.1. Descripcion 16
3.3.2. Metodologia 16
3.3.3. Resultados 17
3.3.4. Discusion 20
3.4. YOLO + SORT 20
3.4.1. Descripcion 20
3.4.2. Metodología 21
3.4.3. Resultados 22
3.4.4. Discusion 25
3.5. Haar + SORT 25
3.5.1. Descripcion 25
3.5.2. Metodologia 26
3.5.3. Resultados 27
3.5.4. Discusion 30
3.6. Comparación Global de los Sistemas Evaluados 31

4. Conclusiones **32**
4.1. Sumario 32
4.2. Conclusión 32

Bibliografía **35**

Appendices **36**

A. ANEXO: Código **37**

Índice de Figuras

2.1. Esquema general de la arquitectura YOLO. 7
2.2. Esquema de funcionamiento del algoritmo Haar Cascade. 8
2.3. Diagrama del flujo de trabajo de MTCNN: detección jerárquica y predicción de puntos faciales clave. 10
2.4. Diagrama de Deep SORT: integración de detección, características visuales y seguimiento multi-modal. 12

3.1. Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad 17
3.2. Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad 18
3.3. Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad 19
3.4. Detección y seguimiento de múltiples rostros con el sistema YOLO + SORT. Se observan identificadores asignados en una escena con alta densidad de personas. 22
3.5. Seguimiento facial en una escena con alta densidad de personas (Video 2). El sistema YOLO + SORT asigna múltiples identificadores de forma simultánea. 23
3.6. Seguimiento facial con YOLO + SORT en un entorno denso y dinámico (Video 3). Se observa la correcta asignación de múltiples IDs incluso con oclusiones parciales. 24
3.7. Seguimiento facial con el sistema Haarcascade + SORT. Se observan varios rostros identificados correctamente en una escena peatonal. . . . 27

3.8. Seguimiento facial en Video 2 con el sistema Haarcascade + SORT. Se observa la correcta identificación de múltiples rostros en movimiento. . 28
3.9. Seguimiento de múltiples rostros en Video 3 utilizando Haarcascade + SORT. Se observa estabilidad en los identificadores asignados pese a la densidad visual. 29

Índice de Tablas

2.1. Comparación de algoritmos de detección de rostros. 10
2.2. Comparación de algoritmos de seguimiento de objetos. Se consideran precisión, uso de recursos, capacidad de re-identificación y principales características. 14

3.1. Métricas de rendimiento del sistema Haarcascade + Tracking Universidad. 18
3.2. Métricas de rendimiento del sistema Haarcascade + Tracking Universidad (video 2). 19
3.3. Métricas de rendimiento del sistema Haarcascade + Tracking Universidad (video 3). 20
3.4. Métricas de rendimiento del sistema YOLO + SORT (video 1). 23
3.5. Métricas de rendimiento del sistema YOLO + SORT (video 2). 24
3.6. Métricas de rendimiento del sistema YOLO + SORT (video 3). 25
3.7. Métricas de rendimiento del sistema Haarcascade + SORT (video 1). . 28
3.8. Métricas de rendimiento del sistema Haarcascade + SORT (video 2). . 29
3.9. Métricas de rendimiento del sistema Haarcascade + SORT (video 3). . 30

Siglas

AP Average Precision

BER Bit Error Rate

CPU Central Processing Unit

FCN Fully Convolutional Network

FER Facial Expression Recognition

FO Fibra(s) Óptica(s)

GPU Graphics Processing Unit

IOU Intersection Over Union

KLT Kanade-Lucas-Tomasi

NMS Non-Maximum Suppression

SFD Soft Failure Detection

SORT Simple Online and Realtime Tracking

TdeF Transformada de Fourier

YOLO You Only Look Once

Capítulo 1

Introducción

A lo largo del tiempo el reconocimiento y seguimiento de rostros en video ha experimentado un notable avance gracias a la incorporación de algoritmos basados en inteligencia artificial y visión por computador. Esta línea de investigación ha cobrado gran relevancia debido a su amplio rango de aplicaciones en ámbitos como la seguridad, el control de acceso, la autenticación biométrica y el análisis de comportamiento en entornos públicos y privados [1].

Investigaciones previas han abordado la detección facial utilizando distintos enfoques. Entre los más tradicionales se encuentran los algoritmos de detección basados en características geométricas, como Haarcascade [2], los cuales permiten detectar patrones faciales mediante clasificadores en cascada. Con el desarrollo de redes neuronales profundas, se introdujeron modelos como YOLO (You Only Look Once) [3], los cuales permiten una detección precisa y en tiempo real, incluso en condiciones de oclusión parcial o iluminación variable. Paralelamente, se han desarrollado algoritmos de seguimiento como SORT (Simple Online and Realtime Tracking) [4], que permiten mantener la identificación de un rostro a lo largo del tiempo, integrando métodos de predicción como el filtro de Kalman y el algoritmo de asignación húngaro.

La relevancia de este trabajo radica en su aporte a la evaluación comparativa de distintos algoritmos de detección y seguimiento de rostros, considerando no solo la precisión y la robustez frente a condiciones adversas, sino también el consumo de recursos computacionales y la velocidad de procesamiento. En un contexto donde la automatización de tareas de seguridad se vuelve cada vez más necesaria, esta

investigación contribuye al diseño de soluciones más eficientes y adaptables a distintas realidades operativas.

El problema abordado se centra en identificar qué combinación de algoritmos ofrece el mejor desempeño en tareas de seguimiento facial en video. El objetivo general fue comparar el rendimiento de tres sistemas de seguimiento de rostros: uno desarrollado en la universidad y dos modelos conocidos de la literatura (YOLO+SORT y Haarcascade+SORT).

Para lograr este objetivo, se realizó primero una revisión bibliográfica de los algoritmos más relevantes, luego se implementaron los modelos seleccionados utilizando herramientas como OpenCV y PyTorch [5, 6], y finalmente se realizaron pruebas controladas con videos de referencia. Los datos recolectados fueron analizados y comparados mediante métricas estándares en visión por computador [7].

La motivación para llevar a cabo este trabajo nace de la necesidad de contar con herramientas tecnológicas que permitan un monitoreo visual más efectivo, confiable y accesible. Esto cobra especial importancia en entornos donde la vigilancia en tiempo real es crítica y los recursos disponibles son limitados.

1.1. Definición del problema

En el contexto del procesamiento de video en tiempo real, la detección y seguimiento de rostros sigue presentando varios desafíos. Si bien existen algoritmos ampliamente utilizados en la literatura, como Haarcascade, YOLO y SORT, muchos de estos presentan limitaciones cuando se enfrentan a condiciones las cuales se deben adecuar los algoritmos , como cambios bruscos de iluminación, movimientos rápidos o escenarios con múltiples objetos.

Frente a estos problemas mencionados, surge la necesidad de realizar un estudio comparativo entre distintos sistemas de detección y seguimiento facial, incluyendo un modelo desarrollado en la universidad, con el objetivo de evaluar cuál entrega un mejor equilibrio entre precisión, velocidad, consumo de recursos y robustez. Este trabajo busca determinar qué combinación de algoritmos se adapta mejor a escenarios reales donde se requiere un monitoreo continuo y confiable con limitaciones técnicas y operativas.

1.2. Objetivos

1.2.1. Objetivo general

El objetivo general de este trabajo es estudiar y comparar el desempeño de distintos tipos de algoritmos de procesamiento de video enfocados en la detección de rostros. Se harán distintas pruebas de precisión de los algoritmos, el consumo de recursos, robustez ante variaciones y velocidad de procesamiento. Esta investigación tiene como finalidad identificar cuál de los métodos proporciona el mejor desempeño, contribuyendo así al avance del algoritmo creado por la universidad

1.2.2. Objetivos específicos

1. Investigación y elección de los diferentes algoritmos de procesamiento de video para la detección de rostros y tracking, incluyendo sus ventajas y desventajas.
2. Implementación de los algoritmos de detección y tracking seleccionados en un entorno de prueba (YOLO+SORT y Haarcascade+SORT).
3. Realización de pruebas entre los algoritmos mencionados y el desarrollado en la universidad, evaluando su rendimiento en términos de precisión, consumo de recursos, robustez ante variaciones y velocidad de procesamiento.
4. Análisis comparativo de los resultados obtenidos.

1.3. Metodología

1. **Investigación y elección de los diferentes algoritmos de procesamiento de video para la detección de rostros y tracking incluyendo sus ventajas y desventajas**:Se llevará a cabo una revisión de la literatura relacionada con la detección de rostros y tracking para la selección de lo os algoritmos más prometedores que se alineen con los objetivos del trabajo.
2. **Implementación de los algoritmos de detección y tracking seleccionados en un entorno de prueba (YOLO+SORT y Haarcascade+SORT)**:Los

algoritmos seleccionados serán implementados en un entorno de desarrollo adecuado, utilizando herramientas y bibliotecas como OpenCV, TensorFlow o PyTorch. Se documentará el proceso de implementación para asegurar la reproducibilidad de los resultados.

3. **Realización de pruebas entre los algoritmos mencionados y el desarrollado en la universidad, evaluando su rendimiento en términos de precisión, consumo de recursos, robustez ante variaciones y velocidad de procesamiento**: Se realizarán pruebas comparativas utilizando conjuntos de datos estándar y previamente establecidos. Se evaluará el rendimiento de cada algoritmo en términos de precisión, velocidad de procesamiento, consumo de recursos y robustez ante varacioness. Para esto, se diseñará un protocolo de pruebas que garantice la consistencia y validez de los resultados.

4. **Análisis comparativo de los resultados obtenidos.**:Los datos obtenidos se analizarán estadísticamente para determinar las diferencias significativas entre los algoritmos. Se generarán gráficos y tablas que permitan visualizar los resultados y facilitar la comparación.

1.4. Alcances y limitaciones

1.4.1. Alcances

Este trabajo se enfoca en la evaluación comparativa de tres sistemas de detección y seguimiento de rostros en video (Haarcascade+SORT, YOLO+SORT y un modelo universitario), abordando temas de visión por computador y procesamiento de imágenes. Cubre desde la revisión bibliográfica hasta la implementación y prueba de los algoritmos, evaluando precisión, velocidad, robustez y uso de recursos. Se espera obtener un análisis técnico que identifique ventajas y limitaciones de cada sistema, acompañado de gráficos, código y recomendaciones. El estudio se realiza bajo condiciones controladas con videos de orientación frontal, movimientos moderados y sin GPU, por lo que los resultados son válidos en contextos de hardware convencional. La propuesta está orientada a aplicaciones en videovigilancia y monitoreo en entornos con recursos limitados.

1.4.2. Limitaciones

Las limitaciones de este proyecto incluyen la velocidad a la que opera el tracking, que puede afectar la capacidad de los algoritmos para seguir rostros en situaciones de movimiento rápido. Además, los recursos computacionales disponibles pueden restringir el uso de algoritmos más complejos o la capacidad de procesar múltiples rostros simultáneamente. También se debe con- siderar el desafío que presenta el movimiento rápido de las personas en el video, ya que esto puede dificultar el seguimiento preciso y generar errores de detección. Asimismo, cualquier obs- trucción que impida la visualización completa del rostro, como sombras o elementos que pasen por delante, podría afectar negativamente el rendimiento de los algoritmos evaluados

Capítulo 2

Marco Teórico

2.1. Introducción

2.1.1. Deteccion de objetos

2.1.1.1. YOLO (You Only Look Once)

YOLO (You Only Look Once) es una arquitectura de detección de objetos basada en redes neuronales convolucionales que trata el problema como una única tarea de regresión. A diferencia de otros métodos como R-CNN o SSD, YOLO predice directamente las coordenadas de las cajas delimitadoras y las clases asociadas desde una imagen completa, logrando un rendimiento en tiempo real incluso en hardware limitado [3, 8].

El modelo ha evolucionado desde su versión original (YOLOv1) hasta variantes más precisas y eficientes como YOLOv5 y YOLOv8. Estas versiones incorporan mejoras como activaciones SiLU, detección de objetos pequeños, arquitectura CSPNet y entrenamiento optimizado por medio de PyTorch, lo que las hace ideales para tareas críticas como vigilancia, reconocimiento facial y conducción autónoma [9].

La arquitectura divide la imagen en una cuadrícula y predice múltiples bounding boxes por celda. Su capacidad para hacer detección múltiple por frame con baja latencia lo convierte en un detector ideal para combinar con algoritmos de tracking como SORT o

DeepSORT.

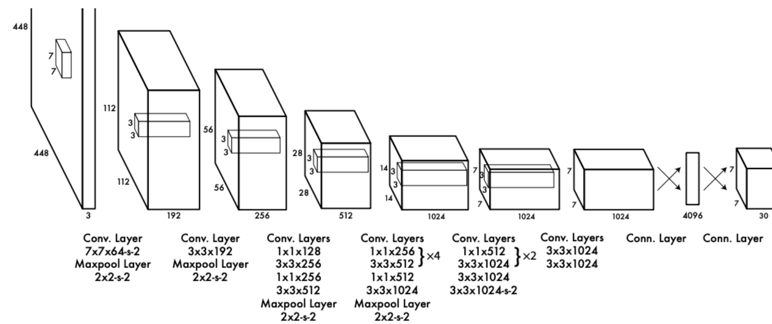

Fig. 2.1: Esquema general de la arquitectura YOLO.

2.1.1.2. Caracteristicas de Haarcascade

El algoritmo Haar Cascade, desarrollado por Viola y Jones en 2001, constituye una de las metodologías más clásicas y utilizadas para la detección de objetos, especialmente rostros humanos, en imágenes digitales [2]. Este método se basa en la extracción de características Haar, que representan diferencias de intensidad entre regiones adyacentes, similares a los filtros utilizados para detección de bordes.

Una de las principales ventajas de Haarcascade es su alta eficiencia computacional, lograda mediante el uso de ïmágenes integralesz la organización jerárquica de clasificadores débiles en una estructura en cascada. Esta arquitectura permite descartar rápidamente regiones irrelevantes en las primeras etapas y refinar las detecciones en etapas posteriores, logrando detección en tiempo real incluso en dispositivos con recursos limitados [10].

La figura 2.2 muestra un esquema general del funcionamiento del algoritmo Haarcascade, desde la aplicación de los filtros hasta la decisión final del clasificador.

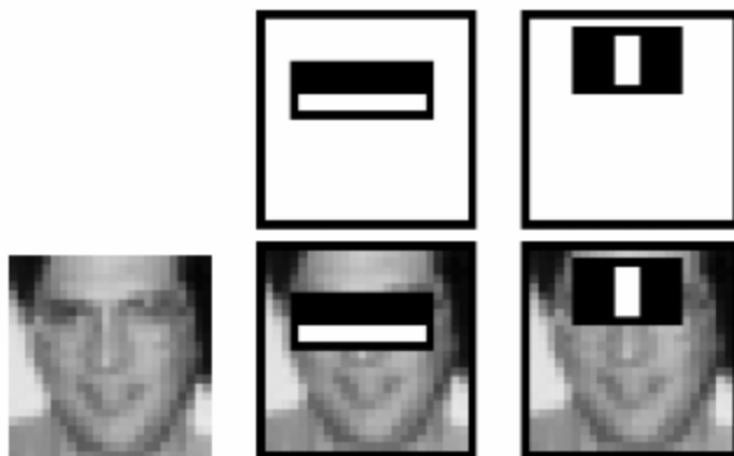

Fig. 2.2: Esquema de funcionamiento del algoritmo Haar Cascade.

A pesar de haber sido superado en precisión por técnicas modernas como las redes neuronales convolucionales (CNNs), Haarcascade sigue siendo ampliamente utilizado en aplicaciones donde la eficiencia y simplicidad son prioritarias. Su implementación está disponible de forma nativa en bibliotecas como OpenCV, lo que facilita su integración en sistemas embebidos, cámaras de vigilancia o aplicaciones móviles [5].

El algoritmo *Haar Cascade*, propuesto por Paul Viola y Michael Jones en 2001, es uno de los métodos más conocidos para la detección de objetos, especialmente rostros, en imágenes digitales. Se basa en un enfoque de aprendizaje supervisado y detección en tiempo real que combina múltiples clasificadores simples en una estructura en cascada, evaluando subventanas de una imagen a distintas escalas y posiciones

El método utiliza características de Haar, que consisten en diferencias de intensidades entre regiones adyacentes de una imagen, similares a filtros simples de bordes. Estas características se extraen mediante la técnica de *integral images*, lo que permite calcularlas de forma extremadamente eficiente.

El algoritmo se entrena utilizando un gran conjunto de imágenes positivas (con el objeto) y negativas (sin el objeto). A partir de este conjunto, se seleccionan las características más relevantes mediante el algoritmo AdaBoost, que construye clasificadores fuertes a partir de clasificadores débiles.

Estos clasificadores son luego organizados en una cascada jerárquica: las primeras etapas eliminan rápidamente regiones sin información relevante, mientras que las

últimas realizan verificaciones más exhaustivas, permitiendo una alta velocidad de procesamiento.

Aunque ha sido en gran medida reemplazado por métodos basados en redes neuronales profundas en aplicaciones de alto rendimiento, Haar Cascade sigue siendo ampliamente utilizado debido a su simplicidad, rapidez y disponibilidad en bibliotecas como *OpenCV*. Es particularmente útil en entornos con recursos computacionales limitados o donde se requiere una implementación sencilla y ligera.

Se ha utilizado tradicionalmente para tareas de detección facial en tiempo real, como en cámaras web, sistemas de vigilancia y dispositivos móviles. Además, su integración con algoritmos de seguimiento como KLT o MOSSE permite aplicaciones completas de detección y seguimiento con bajo costo computacional.

2.1.1.3. MTCNN

MTCNN (Multi-task Cascaded Convolutional Neural Network) es un algoritmo diseñado específicamente para la detección de rostros y localización precisa de puntos clave faciales (landmarks). Fue propuesto por Zhang et al. en 2016 como una solución robusta frente a variaciones de iluminación, poses, expresiones faciales y oclusiones parciales [11].

La arquitectura de MTCNN consta de tres redes convolucionales organizadas en cascada:

- **P-Net (Proposal Network)**: Genera regiones candidatas que posiblemente contienen un rostro.
- **R-Net (Refine Network)**: Filtra y refina las regiones candidatas, ajustando sus límites.
- **O-Net (Output Network)**: Realiza una refinación final y predice los cinco puntos clave del rostro (ojos, nariz y comisuras de la boca).

Cada una de estas redes está entrenada para realizar simultáneamente la detección del rostro y la regresión de los landmarks, lo que mejora notablemente la precisión del sistema. Su capacidad para alinear rostros de forma automática lo convierte en

una herramienta esencial para tareas de reconocimiento facial, seguimiento en video y análisis emocional [11, 12].

En la figura 2.3 se muestra el flujo de procesamiento de MTCNN, desde la detección inicial hasta la predicción final de landmarks.

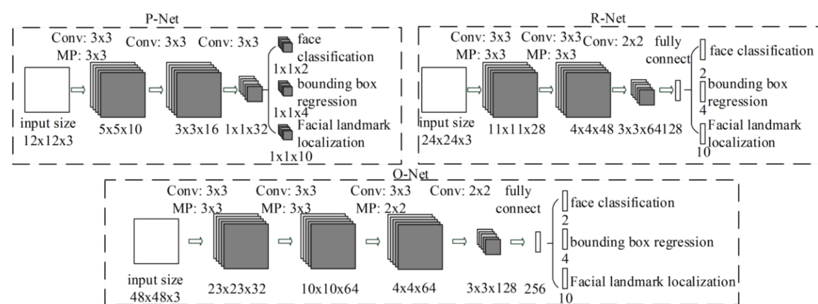

Fig. 2.3: Diagrama del flujo de trabajo de MTCNN: detección jerárquica y predicción de puntos faciales clave.

Comparado con métodos tradicionales como Haarcascade o HOG+SVM, MTCNN ofrece una precisión superior, especialmente en condiciones no controladas. Aunque su velocidad puede ser menor que la de modelos livianos como YOLO, su precisión lo hace adecuado para aplicaciones que requieren alta confiabilidad, siempre que se cuente con soporte por GPU.

Tabla 2.1: Comparación de algoritmos de detección de rostros.

Algoritmo	Precisión	Velocidad	Robustez	Uso típico
Haarcascade	Media	Alta	Baja	Dispositivos con bajo cómputo
YOLO	Alta	Alta	Alta	Sistemas en tiempo real
MTCNN	Muy Alta	Media	Alta	Reconocimiento facial fino

Como se observa en la Tabla 2.1, el algoritmo Haarcascade destaca por su rapidez y bajo consumo, pero tiene limitaciones frente a condiciones no controladas. En contraste, YOLO presenta alta precisión y velocidad, siendo adecuado para sistemas en tiempo real. Por otro lado, MTCNN ofrece la mayor precisión, aunque a un mayor costo computacional.

2.1.2. Tracking de objetos

2.1.2.1. SORT

SORT (Simple Online and Realtime Tracking) es un algoritmo de seguimiento de objetos propuesto por Bewley et al. en 2016. Está diseñado para operar en tiempo real, siendo especialmente adecuado para escenarios donde se requiere seguimiento múltiple de objetos con bajo costo computacional [4].

El enfoque de SORT se basa en el paradigma tracking-by-detection, donde las detecciones en cada cuadro (frame) del video son proporcionadas por un detector externo (por ejemplo, YOLO o Haarcascade), y luego asociadas entre cuadros sucesivos mediante técnicas clásicas de estimación y optimización.

SORT emplea dos componentes fundamentales:

- **Filtro de Kalman**: Estima el estado futuro de cada objeto (posición y velocidad), basándose en un modelo de movimiento lineal con ruido gaussiano.
- **Algoritmo húngaro**: Resuelve el problema de asignación óptima entre las nuevas detecciones y las predicciones previas, maximizando la coherencia espacial (típicamente mediante el uso del IoU como métrica de distancia).

SORT se ha implementado con éxito en sistemas de videovigilancia, análisis de tráfico vehicular y seguimiento de personas en tiempo real. Sin embargo, presenta limitaciones en escenarios con oclusiones prolongadas o cambios abruptos de apariencia, lo que motivó el desarrollo de algoritmos más robustos como Deep SORT [13].

2.1.2.2. DeepSORT

Deep SORT (Deep Simple Online and Realtime Tracking) es una extensión del algoritmo SORT que incorpora descripciones visuales profundas (deep features) para mejorar la asociación de objetos entre cuadros consecutivos, especialmente en condiciones desafiantes como oclusiones, desapariciones temporales y cruces entre objetos [13].

Mientras SORT se basa únicamente en información espacial (posición, velocidad), Deep SORT agrega un vector de características visuales (embedding) generado por

una red neuronal convolucional preentrenada. Este vector describe la apariencia del objeto, permitiendo que el sistema mantenga la identidad incluso si el objeto sale temporalmente del campo visual.

El pipeline de Deep SORT conserva la estructura base de SORT (filtro de Kalman + algoritmo húngaro), pero modifica el paso de asignación incluyendo una métrica visual basada en distancia coseno o distancia de Mahalanobis.

- **Extracción de características**: Una CNN genera un embedding de cada objeto detectado.
- **Asociación multi-modal**: Combina distancia de ubicación (IoU) y distancia de apariencia para decidir la correspondencia entre detecciones actuales y trayectorias previas.

La figura 2.4 muestra un esquema general del funcionamiento de Deep SORT.

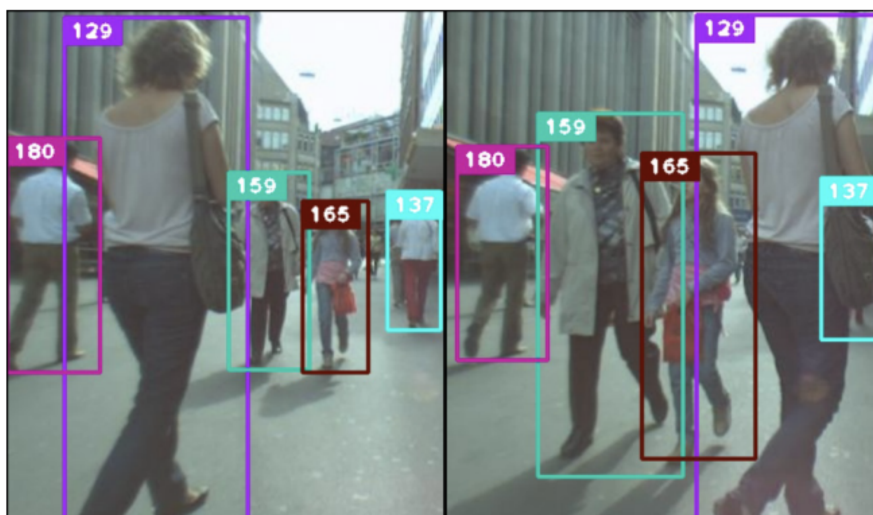

Fig. 2.4: Diagrama de Deep SORT: integración de detección, características visuales y seguimiento multi-modal.

Gracias a esta integración visual, Deep SORT ha sido adoptado en numerosos sistemas de vigilancia, análisis de multitudes y seguimiento vehicular. También ha demostrado una mayor estabilidad en escenarios reales, donde la sola información de posición no es suficiente para una asociación precisa.

El diseño modular de Deep SORT permite su integración con diversos detectores como

YOLO, MTCNN o Faster R-CNN, lo que le otorga gran versatilidad en aplicaciones de visión por computador [14].

2.1.2.3. MOOSE

El algoritmo MOOSE (Minimum Output Sum of Squared Error), propuesto por Bolme et al. en 2010, es un método de seguimiento visual basado en filtros de correlación adaptativos. Su principal fortaleza radica en su extrema eficiencia computacional, lo que lo hace ideal para aplicaciones en tiempo real y sistemas con recursos limitados [15].

A diferencia de los enfoques que dependen de redes neuronales o características visuales profundas, MOOSE emplea operaciones en el dominio de Fourier para entrenar filtros que maximizan la respuesta en la región del objeto a seguir, minimizando el error cuadrático medio respecto a una salida deseada (generalmente una distribución gaussiana centrada en el objetivo).

- **Entrenamiento**: Se construye un filtro en frecuencia que maximiza la correlación con el objeto objetivo.
- **Actualización en línea**: El filtro se ajusta progresivamente a medida que el objeto cambia de apariencia o iluminación.
- **Correlación rápida**: El uso de la Transformada Rápida de Fourier (FFT) reduce el costo computacional a $\mathcal{O}(n \log n)$.

MOOSE ha sido ampliamente utilizado en entornos donde la velocidad y el bajo consumo de recursos son prioritarios, como en sistemas embebidos, cámaras de seguridad, drones y robótica móvil. No obstante, presenta ciertas limitaciones al enfrentar oclusiones prolongadas, reidentificación de objetos o cambios drásticos de apariencia. Por ello, suele emplearse como base de comparación frente a algoritmos más complejos como Deep SORT [16].

Tabla 2.2: Comparación de algoritmos de seguimiento de objetos. Se consideran precisión, uso de recursos, capacidad de re-identificación y principales características.

Algoritmo	**Precisión**	**Recursos**	**Re-ID**	**Notas**
SORT	Media	Bajo	No	Rápido y simple. No distingue objetos visualmente similares.
DeepSORT	Alta	Medio	Sí	Incorpora embeddings visuales para mantener la identidad.
MOOSE	Baja	Muy Bajo	No	Uso mínimo de procesamiento. Basado en filtros de correlación.

En la Tabla 2.2 se puede observar que el algoritmo SORT es muy utiles ,pero no muy precisos en entornos complejos. DeepSORT mejora significativamente el seguimiento al incorporar descriptores visuales, mientras que MOOSE se orienta a sistemas embebidos con restricciones severas.

Capítulo 3

Desarrollo

3.1. Introducción

A continuación, se presentan detalles de lo realizado durante el proyecto en base a lo mencionado en los objetivos de esta Memoria de Título. Estas pruebas son realizadas para investigar

3.2. Implementos utilizados

Para el desarrollo, prueba y evaluación de los sistemas presentados en esta memoria, se utilizó un equipo MacBook Pro de 15 pulgadas, modelo 2018. Este equipo cuenta con un procesador Intel Core i7 de 6 núcleos a 2.6 GHz, 16 GB de memoria RAM DDR4 a 2400 MHz y el chip gráfico integrado Intel UHD Graphics 630. El sistema operativo utilizado fue macOS Sequiola (versión 15.5), y el entorno de desarrollo consistió en Python 3.11 con bibliotecas como OpenCV, NumPy, psutil y `ultralytics` para modelos YOLO. Todas las pruebas fueron realizadas sin aceleración por GPU, privilegiando la reproducibilidad en entornos de cómputo moderado.

3.3. Haarcascade + Tracking Universidad

3.3.1. Descripcion

Este sistema fue desarrollado por la Universidad como una alternativa a los métodos tradicionales de detección y seguimiento de rostros. Se basa en el uso del clasificador Haarcascade para la detección facial y en un esquema personalizado de seguimiento que incorpora un filtro de paso alto temporal. Este filtro permite estabilizar las trayectorias de los objetos detectados a lo largo del tiempo, reduciendo fluctuaciones erráticas. A través de esta arquitectura modular, se busca mantener la identidad de los rostros en secuencias de video sin depender de algoritmos externos como SORT o DeepSORT.

3.3.2. Metodologia

El sistema se compone de tres módulos principales implementados en los archivos `Haar1.py`, `full_tracking.py` y `high_pass_filter.py`, los cuales trabajan de forma conjunta:

El sistema se compone de tres módulos principales implementados en los archivos `Haar1.py`, `full_tracking.py` y `high_pass_filter.py`, los cuales trabajan de forma conjunta:

- **Detección de rostros:** En el script que implementa el sistema Haar + Tracking Universidad (ver Listado A.2), se utiliza el clasificador preentrenado `haarcascade_frontalface_default.xml`, disponible en OpenCV. Este clasificador se aplica sobre imágenes en escala de grises mediante la función `detectMultiScale()` para obtener las coordenadas de los rostros en cada frame del video.

- **Seguimiento personalizado:** En el módulo `full_tracking.py` se implementa la clase `FullTracking`, encargada de asignar un ID único a cada rostro detectado. Este componente mantiene la identidad de los objetos a lo largo del tiempo, gestionando el historial de detecciones. La lógica de seguimiento está diseñada específicamente para operar con las coordenadas provenientes del clasificador Haarcascade.

- **Filtrado de trayectorias:** En el módulo `high_pass_filter.py`, se encuentra la clase `HighPassFilter`, presentada en el Listado A.4. Este filtro actúa sobre las posiciones detectadas en el tiempo, eliminando pequeñas oscilaciones causadas por ruido o variabilidad del clasificador. Con ello se mejora considerablemente la estabilidad visual de los identificadores en pantalla.
- **Evaluación en videos reales:** El sistema fue probado sobre tres videos reales en los que se observaban claramente rostros humanos. Para cada uno, se utilizó OpenCV para procesar los fotogramas, aplicar el detector y mostrar los identificadores asignados, mientras se registraba el rendimiento del sistema mediante la biblioteca `psutil`. Se midió el uso promedio de CPU, RAM, FPS y tiempo de procesamiento por cuadro.

3.3.3. Resultados

En el primer video aprecia el seguimiento de la mujer en especifico El sistema mantuvo un seguimiento estable, con asignación continua del ID. La Figura 3.1 muestra un fotograma típico del seguimiento, y la Tabla 3.1 resume las métricas asociadas.

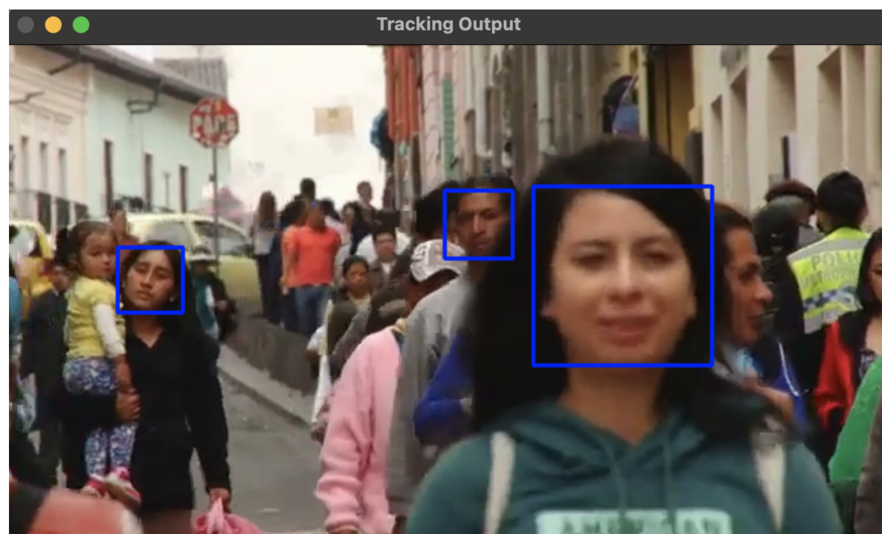

Fig. 3.1: Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad .

Tabla 3.1: Métricas de rendimiento del sistema Haarcascade + Tracking Universidad.

Métrica	**Valor**	**Unidad**
Total de cuadros procesados	300	frames
Tiempo promedio por frame	0.0172	segundos
FPS promedio	18.63	frames/s
Uso promedio de CPU	29.58	%
Uso promedio de RAM	60.70	%
Uso máximo de RAM	61.10	%
Tiempo total de ejecución	16.11	segundos

En el siguiente video se puede ver dos personas caminando y cruzándose frente a la cámara. Se asignaron dos identificadores que se mantuvieron coherentes. La Figura 3.2 muestra el seguimiento simultáneo, y la Tabla 3.2 incluye los valores medidos.

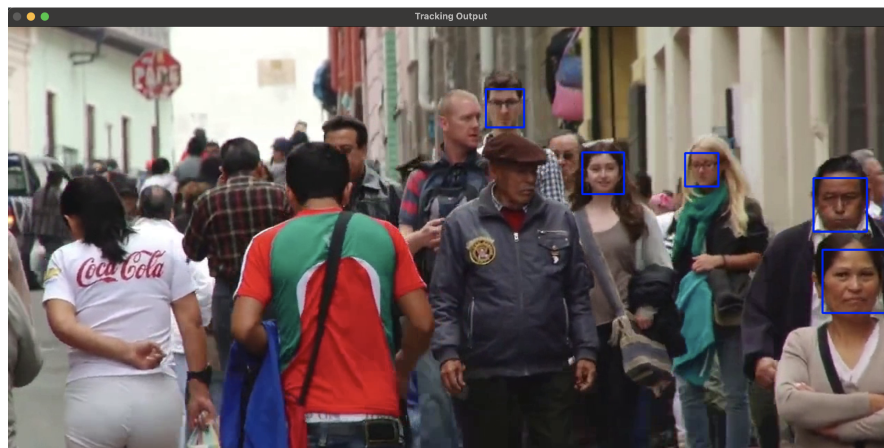

Fig. 3.2: Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad .

Tabla 3.2: Métricas de rendimiento del sistema Haarcascade + Tracking Universidad (video 2).

Métrica	**Valor**	**Unidad**
Total de cuadros procesados	374	frames
Tiempo promedio por frame	0.0467	segundos
FPS promedio	11.24	frames/s
Uso promedio de CPU	47.26	%
Uso promedio de RAM	60.83	%
Uso máximo de RAM	61.30	%
Tiempo total de ejecución	33.27	segundos

En el ultimo video se intenta verificar entornos diferentes a los demas . Se mantuvo la identificación, aunque con menor FPS. La Figura 3.3 representa un ejemplo de detección en ese contexto, y la Tabla 3.3 entrega el resumen numérico.

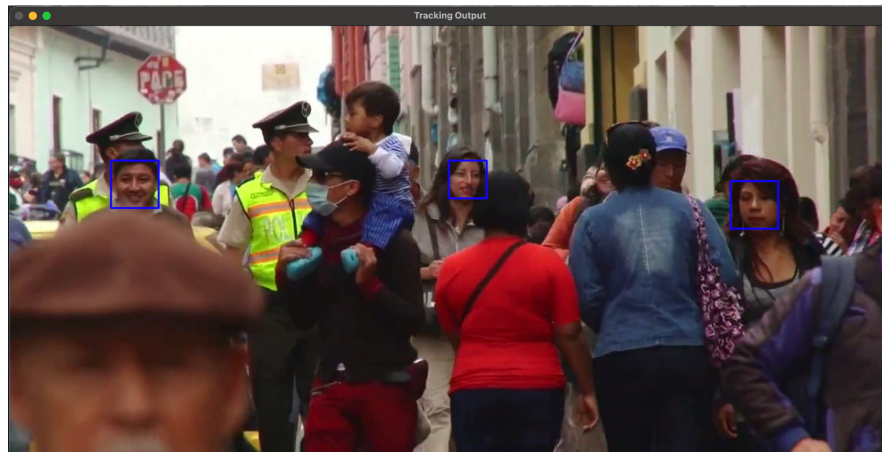

Fig. 3.3: Captura de pantalla con las métricas de rendimiento obtenidas por el sistema Haarcascade + Tracking Universidad .

Tabla 3.3: Métricas de rendimiento del sistema Haarcascade + Tracking Universidad (video 3).

Métrica	**Valor**	**Unidad**
Total de cuadros procesados	254	frames
Tiempo promedio por frame	0.0426	segundos
FPS promedio	11.74	frames/s
Uso promedio de CPU	46.32	%
Uso promedio de RAM	60.62	%
Uso máximo de RAM	61.20	%
Tiempo total de ejecución	21.64	segundos

3.3.4. Discusion

El sistema Haarcascade + Tracking Universidad mostró ser una alternativa eficiente para contextos con restricciones computacionales. Su diseño modular y su implementación simple en Python sin necesidad de GPU lo convierten en una opción accesible. El filtro de paso alto fue clave para suavizar las trayectorias, reduciendo el parpadeo de los IDs y mejorando la experiencia visual.

A lo largo de los tres videos, el sistema mantuvo un rendimiento estable con valores de FPS entre 11 y 18, y un consumo de CPU moderado. Sin embargo, su dependencia exclusiva del clasificador Haarcascade —sensible a variaciones de iluminación, oclusiones y ángulos faciales— representa una debilidad frente a entornos más complejos. Pese a ello, logró mantener la continuidad de los identificadores en todos los casos, cumpliendo su objetivo de seguimiento básico con estabilidad.

3.4. YOLO + SORT

3.4.1. Descripcion

En este apartado se implementó un sistema de detección y seguimiento de rostros en video utilizando el modelo YOLOv11n-face y el algoritmo de tracking SORT. El objetivo

fue evaluar el rendimiento de esta combinación en condiciones de prueba controladas. YOLOv11n-face es un modelo liviano optimizado para la detección de rostros, mientras que SORT permite realizar un seguimiento en tiempo real mediante el uso de filtros de Kalman y el algoritmo húngaro. Esta configuración permitió asignar un identificador único a cada rostro detectado, preservando su identidad entre frames consecutivos

3.4.2. Metodología

El sistema fue desarrollado en Python utilizando las bibliotecas `ultralytics`, `OpenCV`, `NumPy` y `psutil`. A continuación, se describe el procedimiento seguido:

- **Carga del video:** Se utilizaron distintos videos en los que se observaban varias personas con el rostro visible. Estos archivos fueron procesados utilizando la clase `cv2.VideoCapture` de OpenCV, lo que permitió acceder a los cuadros (frames) individuales.
- **Detección de rostros:** Se empleó el modelo preentrenado `yolov11n-face.pt` para realizar detección de rostros en cada fotograma del video. Los frames fueron redimensionados a una resolución de 640×360 píxeles para reducir la carga computacional. Se aplicó un umbral de confianza de 0,5, descartando detecciones con baja probabilidad.
- **Seguimiento de rostros:** Las coordenadas de las cajas delimitadoras (bounding boxes) obtenidas con YOLO fueron entregadas al algoritmo SORT (*Simple Online and Realtime Tracking*). Este algoritmo se encargó de asociar las detecciones entre cuadros sucesivos y asignar un identificador único a cada rostro detectado, preservando su identidad a lo largo del tiempo.
- **Visualización:** Se dibujaron sobre cada frame las cajas de detección y los respectivos identificadores de seguimiento mediante funciones de `cv2.rectangle()` y `cv2.putText()`.
- **Monitoreo de rendimiento:** Para cada fotograma se registraron métricas de uso de CPU y RAM antes y después del procesamiento, utilizando la biblioteca `psutil`. También se midió el tiempo de procesamiento por frame.
- **Cálculo de métricas:** Al finalizar la ejecución, se calcularon promedios de

tiempo por fotograma, tasa de cuadros por segundo (FPS), uso medio y máximo de RAM, y porcentaje promedio de uso de CPU.

3.4.3. Resultados

El sistema YOLO + SORT fue evaluado utilizando un video con alta densidad de personas en movimiento. En la Figura 3.4, se observa un fotograma representativo en el que se asignan múltiples identificadores a diferentes rostros de forma simultánea. A pesar de la complejidad de la escena, el sistema logró mantener de forma estable los IDs, evidenciando la capacidad de SORT para realizar asociaciones en tiempo real.

Fig. 3.4: Detección y seguimiento de múltiples rostros con el sistema YOLO + SORT. Se observan identificadores asignados en una escena con alta densidad de personas.

Según se observa en la Tabla 3.4, el sistema procesó 230 cuadros con un promedio de 7.15 FPS y un tiempo medio de procesamiento por cuadro de 0.1246 segundos. El uso de CPU fue moderado (30.18 %), mientras que el uso de memoria se mantuvo estable cerca del 60 %. Estos resultados confirman que, aunque el sistema YOLO + SORT ofrece buena precisión en escenarios complejos, su velocidad es menor en comparación con otros métodos evaluados, debido principalmente a la carga computacional del modelo YOLO.

Tabla 3.4: Métricas de rendimiento del sistema YOLO + SORT (video 1).

Métrica	**Valor**	**Unidad**
Frames procesados	230	–
Tiempo por frame	0.1246	segundos
FPS promedio	7.15	frames/s
CPU promedio	30.18	%
RAM promedio	59.40	%
RAM máxima	60.10	%
Tiempo total	32.15	segundos

En el segundo video evaluado con el sistema YOLO + SORT, se observó un entorno aún más desafiante, con mayor densidad de personas y múltiples objetos en movimiento dentro del mismo plano. La Figura 3.5 muestra un ejemplo del seguimiento realizado, donde el sistema logra identificar correctamente numerosos rostros a pesar de la superposición visual y oclusiones parciales.

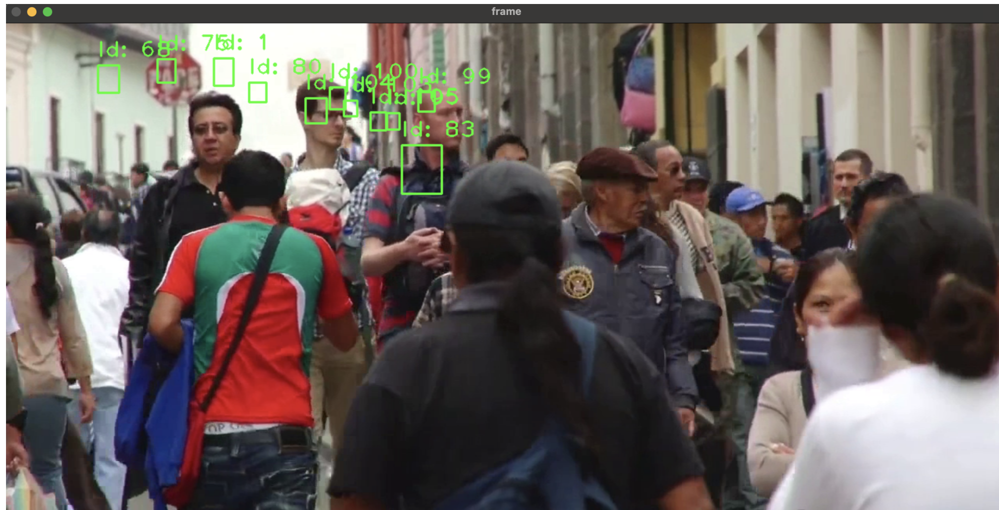

Fig. 3.5: Seguimiento facial en una escena con alta densidad de personas (Video 2). El sistema YOLO + SORT asigna múltiples identificadores de forma simultánea.

Como se indica en la Tabla 3.5, el sistema procesó 340 cuadros con un rendimiento promedio de 5.89 FPS, un tiempo de procesamiento por cuadro de 0.1448 segundos, y un uso de CPU más bajo que en el video anterior (24.55 %). Esto puede atribuirse a

una menor velocidad de movimiento en la escena, lo que reduce la carga del algoritmo SORT. El uso de memoria se mantuvo estable en torno al 60 %.

Tabla 3.5: Métricas de rendimiento del sistema YOLO + SORT (video 2).

Métrica	**Valor**	**Unidad**
Frames procesados	340	–
Tiempo por frame	0.1448	segundos
FPS promedio	5.89	frames/s
CPU promedio	24.55	%
RAM promedio	60.20	%
RAM máxima	60.80	%
Tiempo total	57.75	segundos

En el tercer video evaluado, el sistema YOLO + SORT enfrentó una escena de alta densidad peatonal con sujetos en movimiento en diversas direcciones y varias oclusiones. Tal como se aprecia en la Figura 3.6, el sistema fue capaz de asignar correctamente múltiples identificadores, incluso en presencia de superposición entre personas.

Fig. 3.6: Seguimiento facial con YOLO + SORT en un entorno denso y dinámico (Video 3). Se observa la correcta asignación de múltiples IDs incluso con oclusiones parciales.

La Tabla 3.6 muestra que el sistema procesó 377 cuadros a un ritmo de 6.32 FPS, con un tiempo promedio de procesamiento por cuadro de 0.1345 segundos. El uso promedio

de CPU fue de 28.55 %, mientras que el uso de RAM se mantuvo estable (60.48 %), consistente con los resultados de los otros videos.

Tabla 3.6: Métricas de rendimiento del sistema YOLO + SORT (video 3).

Métrica	**Valor**	**Unidad**
Frames procesados	377	–
Tiempo por frame	0.1345	segundos
FPS promedio	6.32	frames/s
CPU promedio	28.55	%
RAM promedio	60.48	%
RAM máxima	61.20	%
Tiempo total	59.68	segundos

3.4.4. Discusion

YOLO + SORT fue el sistema más robusto en términos de detección y seguimiento preciso, especialmente en entornos complejos. Gracias al modelo YOLOv11n-face, se logró mantener la identificación de múltiples sujetos incluso ante oclusiones, cruces y cambios de iluminación.

No obstante, este rendimiento tuvo un costo en términos de velocidad: con valores de FPS entre 5.8 y 7.1, fue el sistema más lento evaluado. El uso de CPU y RAM fue constante, aunque más exigente que Haar + Universidad. La mayor ventaja del sistema fue la coherencia en la asignación de IDs, sin errores visibles, lo que lo hace ideal para contextos donde la precisión es prioritaria por sobre la eficiencia.

3.5. Haar + SORT

3.5.1. Descripcion

En este apartado se implementó un sistema de detección y seguimiento de rostros utilizando el clasificador Haarcascade para la detección facial y el algoritmo SORT (*Simple Online and Realtime Tracking*) para el seguimiento. El objetivo fue evaluar esta

combinación en escenarios reales mediante videos que contenían personas con el rostro visible. Haarcascade es un detector basado en características simples de intensidad, mientras que SORT emplea predicción con filtro de Kalman y asignación óptima con el algoritmo húngaro. Esta integración permite detectar y mantener la identidad de múltiples rostros a lo largo del tiempo.

El programa utiliza un modelo Haarcascade de OpenCV para detectar los rostros y el algoritmo SORT (Simple Online and Realtime Tracking) para mantener la identidad de cada rostro a través de múltiples fotogramas. Además, se implementa un módulo de monitoreo de rendimiento usando la biblioteca psutil para registrar métricas del sistema durante el procesamiento.

3.5.2. Metodologia

La implementación del sistema Haar+SORT se realizó en Python, utilizando las bibliotecas `OpenCV`, `NumPy`, `psutil` y el módulo `Sort` para el seguimiento. A continuación, se describe el procedimiento seguido:

- **Carga del video:** Se utilizaron videos en los que se observaban personas con el rostro visible. Estos videos fueron procesados mediante la clase `cv2.VideoCapture()`, permitiendo acceder a los fotogramas de forma secuencial.
- **Conversión a escala de grises:** Cada fotograma fue convertido a escala de grises utilizando la función `cv2.cvtColor()`, ya que el clasificador Haarcascade opera sobre imágenes monocromáticas.
- **Detección de rostros:** Se utilizó el modelo preentrenado `haarcascade_frontalface_default.xml` provisto por OpenCV. La detección se realizó mediante la función `detectMultiScale()`, ajustando los parámetros `scaleFactor`, `minNeighbors` y `minSize` para lograr un equilibrio entre precisión y velocidad.
- **Seguimiento de rostros:** Las coordenadas de las detecciones fueron convertidas al formato requerido por el algoritmo SORT: `[xmin, ymin, xmax, ymax]`. Estas fueron ingresadas al tracker `Sort(max_age=50, iou_threshold=0.3)`, el cual asigna un identificador único a cada rostro detectado y lo mantiene a lo largo del tiempo, incluso ante pérdidas momentáneas de detección.

- **Visualización:** Se dibujaron rectángulos en torno a cada rostro detectado y se añadió el ID correspondiente mediante funciones de OpenCV como `rectangle()` y `putText()`.
- **Monitoreo de rendimiento:** En cada ciclo se midió el uso instantáneo de CPU y RAM, así como el tiempo de procesamiento por frame, utilizando funciones de la biblioteca `psutil`.
- **Cálculo de métricas:** Al finalizar la ejecución se calcularon indicadores globales, como el promedio de tiempo de procesamiento por frame, FPS alcanzado, uso promedio y máximo de RAM, y porcentaje promedio de CPU.

3.5.3. Resultados

En el primer video evaluado, el sistema Haarcascade + SORT mostró un rendimiento notablemente eficiente. Como se aprecia en la Figura 3.7, el sistema logró asignar correctamente identificadores a varios rostros presentes en la escena, manteniendo la identidad visual de cada sujeto durante su trayecto.

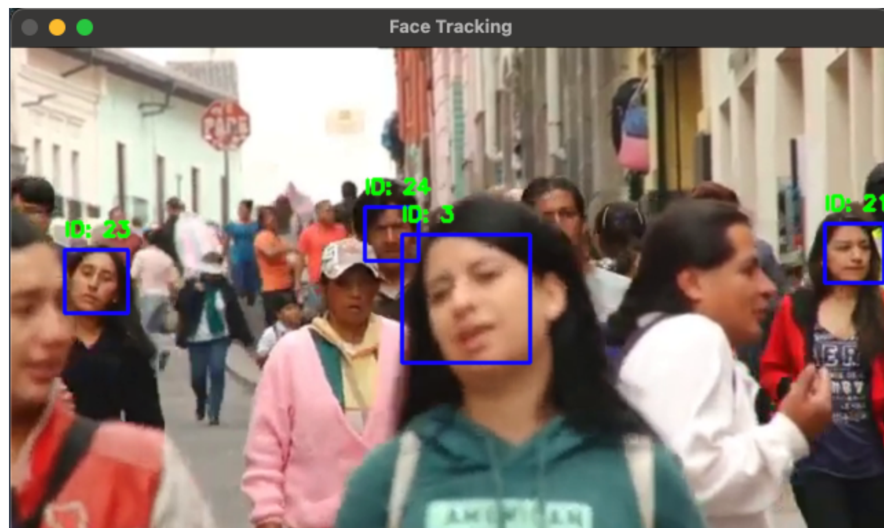

Fig. 3.7: Seguimiento facial con el sistema Haarcascade + SORT. Se observan varios rostros identificados correctamente en una escena peatonal.

La Tabla 3.7 presenta las métricas de rendimiento obtenidas. Se alcanzó un promedio de 15.83 FPS, con un tiempo de procesamiento por cuadro de solo 0.0272 segundos. El

uso de CPU fue particularmente bajo (21.63 %), lo cual lo posiciona como un sistema adecuado para entornos de bajo consumo de recursos. El uso de memoria RAM se mantuvo estable, con un promedio de 60.61 %.

Tabla 3.7: Métricas de rendimiento del sistema Haarcascade + SORT (video 1).

Métrica	**Valor**	**Unidad**
Frames procesados	300	–
Tiempo por frame	0.0272	segundos
FPS promedio	15.83	frames/s
CPU promedio	21.63	%
RAM promedio	60.61	%
RAM máxima	61.40	%
Tiempo total	18.95	segundos

En el segundo video evaluado con el sistema Haarcascade + SORT, se presentó un escenario más exigente, con mayor densidad de personas y desplazamiento simultáneo. Tal como se muestra en la Figura 3.8, el sistema logró realizar un seguimiento estable y coherente para los rostros visibles, incluso en condiciones de oclusión parcial.

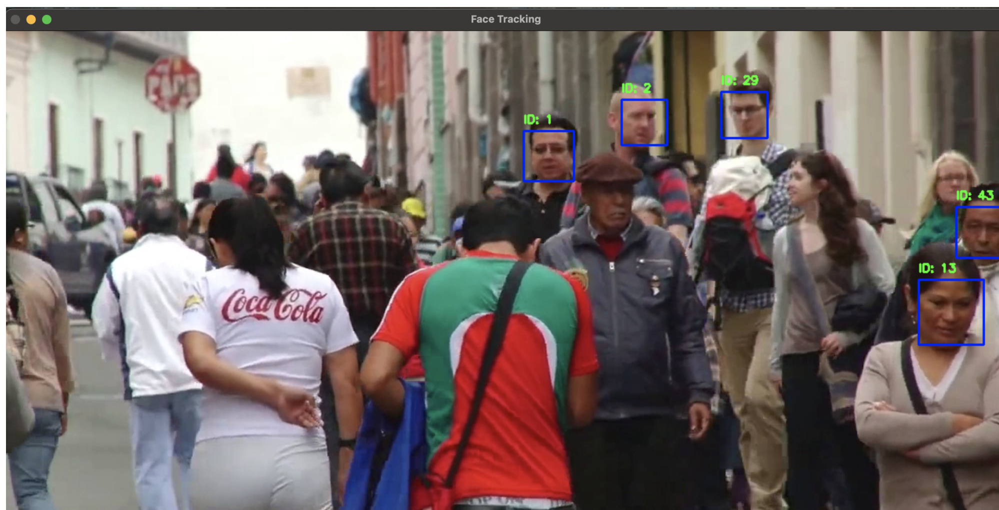

Fig. 3.8: Seguimiento facial en Video 2 con el sistema Haarcascade + SORT. Se observa la correcta identificación de múltiples rostros en movimiento.

La Tabla 3.8 indica que se procesaron 160 cuadros con un promedio de 7.20 FPS. El tiempo de procesamiento por fotograma fue de 0.0964 segundos. A diferencia del primer

video, aquí se observó un aumento considerable en el uso de CPU (44.06 %), producto del procesamiento adicional requerido por el algoritmo SORT en escenas dinámicas. La RAM utilizada se mantuvo en valores similares al resto de las pruebas.

Tabla 3.8: Métricas de rendimiento del sistema Haarcascade + SORT (video 2).

Métrica	**Valor**	**Unidad**
Frames procesados	160	–
Tiempo por frame	0.0964	segundos
FPS promedio	7.20	frames/s
CPU promedio	44.06	%
RAM promedio	60.78	%
RAM máxima	61.30	%
Tiempo total	22.23	segundos

El tercer video representó un entorno especialmente complejo para el sistema Haarcascade + SORT, debido a la alta cantidad de personas, presencia de oclusiones y movimiento en profundidad. Como se muestra en la Figura 3.9, el sistema mantuvo un seguimiento funcional, logrando asignar y sostener múltiples identificadores.

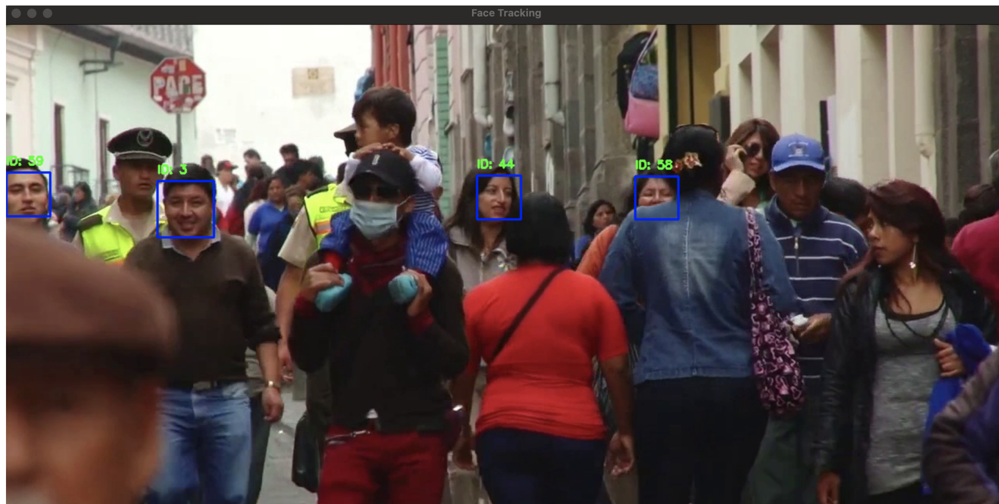

Fig. 3.9: Seguimiento de múltiples rostros en Video 3 utilizando Haarcascade + SORT. Se observa estabilidad en los identificadores asignados pese a la densidad visual.

Según la Tabla 3.9, el sistema procesó 237 cuadros, alcanzando un promedio de 6.95 FPS y un tiempo de procesamiento por cuadro de 0.1011 segundos. Se registró un

uso promedio de CPU de 48.97 %, el más alto entre los tres videos evaluados para este sistema, lo que refleja el esfuerzo computacional adicional requerido ante la mayor complejidad visual. El uso de memoria se mantuvo dentro de márgenes esperables, cercanos al 60 %.

Tabla 3.9: Métricas de rendimiento del sistema Haarcascade + SORT (video 3).

Métrica	**Valor**	**Unidad**
Frames procesados	237	–
Tiempo por frame	0.1011	segundos
FPS promedio	6.95	frames/s
CPU promedio	48.97	%
RAM promedio	59.92	%
RAM máxima	60.80	%
Tiempo total	34.11	segundos

3.5.4. Discusion

El sistema Haar + SORT representó un punto intermedio entre los extremos evaluados. Logró un rendimiento eficiente en términos de velocidad (hasta 15.8 FPS), con bajo consumo de CPU en escenas simples. Sin embargo, al aumentar la complejidad del entorno (más personas, oclusiones), el uso de CPU se elevó significativamente, alcanzando casi el 49 %.

Aunque el clasificador Haarcascade limita la calidad de la detección, SORT permitió compensar en parte estas deficiencias al mantener los IDs con mayor estabilidad que en el sistema Haar + Universidad. La combinación fue efectiva siempre que los rostros fueran claramente visibles y el número de sujetos no fuera excesivo.

3.6. Comparación Global de los Sistemas Evaluados

Tras la evaluación de los tres sistemas propuestos para detección y seguimiento facial Haarcascade + Tracking Universidad, YOLO + SORT y Haarcascade + SORT, se realizó una comparación considerando métricas de rendimiento, robustez en la asignación de identificadores y consumo de recursos computacionales.

El sistema Haarcascade + Tracking Universidad destacó por su simplicidad y eficiencia, logrando un alto número de FPS (hasta 18.6) y bajo consumo de CPU, lo que lo hace especialmente adecuado para implementaciones en tiempo real en dispositivos sin aceleración por hardware. Su principal fortaleza fue la integración del filtro de paso alto, que permitió una visualización más estable. No obstante, su precisión depende en gran medida de la calidad de las detecciones Haarcascade.

Por otro lado, el sistema YOLO + SORT fue el más robusto en términos de continuidad de IDs y detección precisa, manteniendo el seguimiento incluso ante oclusiones y cruces entre personas. Esta capacidad se reflejó en la ausencia de errores en la asignación de identificadores en todos los videos probados. Sin embargo, esta robustez implicó una reducción significativa en los FPS (entre 5.8 y 7.1) y un mayor tiempo por cuadro.

Finalmente, Haarcascade + SORT representó una solución intermedia. En escenas simples logró un excelente rendimiento, pero su estabilidad en el seguimiento se vio afectada en entornos complejos, al depender completamente del clasificador Haarcascade para mantener la identidad.

Capítulo 4

Conclusiones

4.1. Sumario

En este trabajo se desarrollaron y evaluaron tres sistemas de detección y seguimiento facial: Haarcascade + Tracking Universidad, YOLO + SORT y Haarcascade + SORT. Se realizaron pruebas sobre tres videos con condiciones variables (cantidad de personas, iluminación, movimiento), midiendo parámetros como FPS, uso de CPU y RAM, y estabilidad del seguimiento.

El sistema Haar + Universidad destacó por su eficiencia y bajo consumo, manteniendo una buena estabilidad gracias al filtro de paso alto. YOLO + SORT fue el más robusto en seguimiento, pero el más exigente computacionalmente. Haar + SORT ofreció un balance aceptable, destacando por su rapidez en escenas simples. Los resultados permiten visualizar claramente el compromiso entre precisión, velocidad y uso de recursos que cada sistema implica, lo que orienta su aplicación en distintos contextos reales.

4.2. Conclusión

Los sistemas evaluados permitieron estudiar diferentes enfoques de detección y seguimiento facial, abarcando desde soluciones ligeras basadas en Haarcascade hasta combinaciones más avanzadas como YOLO + SORT. A través de la experimentación

en tres escenarios reales, se logró caracterizar su desempeño desde una perspectiva cuantitativa y cualitativa.

Se comprobó que el sistema Haarcascade + Tracking Universidad, si bien limitado en precisión por el clasificador base, es efectivo y estable en escenarios simples, destacando por su eficiencia y facilidad de implementación. Por otro lado, YOLO + SORT mostró ser el sistema más robusto en términos de continuidad de identificación, logrando una detección precisa incluso en escenas complejas, aunque a costa de mayor uso de recursos. Haar + SORT se ubicó como una solución intermedia, con buen rendimiento en condiciones moderadas, pero susceptible a fallas si las detecciones Haarcascade se ven comprometidas.

En general, se cumplió el objetivo planteado: analizar e implementar distintos sistemas de seguimiento facial y compararlos en base a métricas objetivas. Se concluye que no existe un único sistema óptimo, sino que su elección debe depender del contexto específico: disponibilidad de hardware, necesidad de precisión, cantidad de sujetos, etc.

Bibliografía

[1] W. Zhao, R. Chellappa, P. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *ACM computing surveys (CSUR)*, vol. 35, no. 4, pp. 399–458, 2003.

[2] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[4] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in *2016 IEEE International Conference on Image Processing (ICIP)*, 2016, pp. 3464–3468.

[5] G. Bradski, "The opencv library," 2000, https://opencv.org.

[6] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[7] M. Everingham, L. V. Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[8] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[9] G. Jocher, "Yolov5 by ultralytics," https://github.com/ultralytics/yolov5, 2023.

[10] R. Lienhart and J. Maydt, “Extended set of haar-like features for rapid object detection,” *Proceedings. International Conference on Image Processing*, pp. I–900, 2002.

[11] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, “Joint face detection and alignment using multitask cascaded convolutional networks,” in *IEEE Signal Processing Letters*, vol. 23, no. 10, 2016, pp. 1499–1503.

[12] X. Song, X. Li, and P. Zhang, “Face recognition via improved mtcnn and cnn models,” *Procedia Computer Science*, vol. 199, pp. 646–653, 2022.

[13] N. Wojke, A. Bewley, and D. Paulus, “Simple online and realtime tracking with a deep association metric,” *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 3645–3649, 2017.

[14] S. Rudinac, C. Liu, and M. Worring, “Visual tracking for video surveillance: A performance evaluation,” *Multimedia Tools and Applications*, vol. 77, pp. 8889–8912, 2018.

[15] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, “Visual object tracking using adaptive correlation filters,” in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2544–2550.

[16] M. Danelljan, G. Hager, F. S. Khan, and M. Felsberg, “Accurate scale estimation for robust visual tracking,” in *British Machine Vision Conference (BMVC)*, 2014.

Appendices

Apéndice A

ANEXO: Código

```
import numpy as np
import cv2
from sort import Sort
import time
import psutil

if __name__ == '__main__':
    # Cargar el modelo Haarcascade para detección de rostros
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + '
        haarcascade_frontalface_default.xml')
    cap = cv2.VideoCapture("videos/corto.mp4")

    # Inicializar el tracker SORT
    tracker = Sort(max_age=50, iou_threshold=0.3)  # Mantener rastros
        más tiempo

    # --- Variables para el análisis de rendimiento ---
    frame_count = 0
    total_processing_time = 0
    cpu_usage_list = []
    ram_usage_list = []
    start_time_total = time.time()  # Tiempo total de ejecución

    # --- Obtener información del video ---
    fps = cap.get(cv2.CAP_PROP_FPS)
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
```

```
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    print(f"Video FPS: {fps}, Resolution: {width}x{height}")


    # Procesar cada frame
    while cap.isOpened():
        status, frame = cap.read()
        if not status:
            break

        frame_count += 1

        # --- Medición de recursos ANTES del procesamiento ---
        start_time = time.time()
        cpu_percent = psutil.cpu_percent(interval=None)  # Uso de CPU instantáneo
        ram_percent = psutil.virtual_memory().percent  # Uso de RAM

        # Convertir el frame a escala de grises
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Detectar rostros en el frame
        faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))

        # Convertir detecciones al formato esperado por SORT [xmin, ymin, xmax, ymax]
        if len(faces) > 0:  # Validar si hay detecciones
            boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in faces])
        else:
            boxes = np.empty((0, 4))  # Array vacío para SORT

        # Actualizar el tracker
        if boxes.shape[0] > 0:
            tracks = tracker.update(boxes)
        else:
            tracks = np.empty((0, 5))  # Array vacío con 5 columnas (SORT espera este formato)

        # Dibujar las detecciones y los IDs en el frame
```

```
61          for xmin, ymin, xmax, ymax, track_id in tracks:
62              cv2.rectangle(frame, (int(xmin), int(ymin)), (int(xmax),
                    int(ymax)), (255, 0, 0), 2)
63              cv2.putText(frame, f"ID:␣{int(track_id)}", (int(xmin), int
                    (ymin) - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255,
                    0), 2)
64
65
66          #  Medición de recursos DESPU S del procesamiento ---
67          end_time = time.time()
68          processing_time = end_time - start_time
69          total_processing_time += processing_time
70
71          # --- Almacenar datos de uso de recursos ---
72          cpu_usage_list.append(cpu_percent)
73          ram_usage_list.append(ram_percent)
74
75
76          # Mostrar el frame procesado
77          cv2.imshow("Face␣Tracking", frame)
78
79          # Salir al presionar 'q'
80          if cv2.waitKey(int(1000/fps)) & 0xFF == ord('q'): #Control con
                el FPS real del video
81              break
82
83
84      cap.release()
85      cv2.destroyAllWindows()
86
87      # --- Cálculo de métricas de rendimiento ---
88      end_time_total = time.time()  # Tiempo total de finalización
89      total_execution_time = end_time_total - start_time_total
90
91      avg_processing_time = total_processing_time / frame_count if
            frame_count > 0 else 0
92      avg_fps = frame_count / total_execution_time if
            total_execution_time > 0 else 0
93      avg_cpu_usage = sum(cpu_usage_list) / len(cpu_usage_list) if
            cpu_usage_list else 0
```

```
94      avg_ram_usage = sum(ram_usage_list) / len(ram_usage_list) if
            ram_usage_list else 0
95      max_ram_usage = max(ram_usage_list) if ram_usage_list else 0
96
97      # DATOS RECOPILADOS A TRAVES DE LOS VIDEOS
98      print("\n--- Performance Metrics ---")
99      print(f"Total frames processed: {frame_count}")
100     print(f"Average processing time per frame: {avg_processing_time:.4
            f} seconds")
101     print(f"Average FPS: {avg_fps:.2f}")
102     print(f"Average CPU Usage: {avg_cpu_usage:.2f}%")
103     print(f"Average RAM Usage: {avg_ram_usage:.2f}%")
104     print(f"Max RAM Usage: {max_ram_usage:.2f}%")
105     print(f"Total execution time: {total_execution_time:.2f} seconds")
```

Listing A.1: Script principal con Haar+SORT

```
1   import cv2
2   import time
3   import psutil
4   from full_tracking import FullTracking
5   from some_face_classifier import FaceClassifier
6
7
8   if __name__ == '__main__':
9       # Inicializa el clasificador de rostros
10      face_classifier = FaceClassifier()
11
12      # Parámetros para FullTracking
13      radius_threshold = 50  # Umbral de radio
14      fc_fd_distance = 30    # Distancia entre el detector y el
            clasificador
15      elem_in_memory = 10    # Número de elementos en memoria
16      buffer_size = 5        # Tamaño del buffer para el filtro
17      run_hpf_flag = True    # Activar filtro de alta frecuencia
18
19      # Crea una instancia de FullTracking
20      tracking = FullTracking(face_classifier, radius_threshold,
            fc_fd_distance, elem_in_memory, buffer_size, run_hpf_flag)
21
22      # Ruta del video
```

```
23      video_path = 'videos/corto.mp4'
24      cap = cv2.VideoCapture(video_path)
25
26      # Verifica si el video se abrió correctamente
27      if not cap.isOpened():
28          print("No␣se␣pudo␣abrir␣el␣archivo␣de␣video.␣Verifica␣la␣ruta.
               ")
29          exit()
30
31      # --- Variables para el análisis de rendimiento ---
32      frame_count = 0
33      total_processing_time = 0
34      cpu_usage_list = []
35      ram_usage_list = []
36      start_time_total = time.time()  # Tiempo total de ejecución
37
38      # --- Obtener información del video ---
39      fps = cap.get(cv2.CAP_PROP_FPS)
40      width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
41      height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
42      print(f"Video␣FPS:␣{fps},␣Resolution:␣{width}x{height}")
43
44
45      # Procesamiento del video
46      while True:
47          ret, frame = cap.read()
48          if not ret:
49              print("No␣se␣puede␣leer␣el␣marco.␣Terminando.")
50              break
51
52          frame_count += 1
53
54          # --- Medición de recursos ANTES del procesamiento ---
55          start_time = time.time()
56          cpu_percent = psutil.cpu_percent(interval=None)  # Uso de CPU
               instantáneo
57          ram_percent = psutil.virtual_memory().percent  # Uso de RAM
58
59          # Detecta las caras en el fotograma
60          faces = face_classifier.detect_faces(frame)
61
```

```
62        # Dibuja cuadros delimitadores en las caras detectadas
63        # (Esto también forma parte del procesamiento, así que se mide
              su tiempo)
64        for (x, y, w, h) in faces:
65            cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0),
                  2)
66
67        # Simulación de parámetros para FullTracking
68        subject_names = ['ID1', 'ID2'] * len(faces)  # Nombres
              simulados, * len(faces) para tener nombres para cada cara
              detectada.
69        text_location = [(x, y) for (x, y, w, h) in faces]  # Coloca
              texto en la esquina superior izquierda
70        face_crops = [frame[y:y + h, x:x + w] for (x, y, w, h) in
              faces]  # Recortes de caras
71        start_point_list = [(x, y) for (x, y, w, h) in faces]  #
              Coordenadas de inicio
72        end_point_list = [(x + w, y + h) for (x, y, w, h) in faces]  #
               Coordenadas de fin
73
74        # Llama a run_full_tracking
75        output_frame, n_subject_names, n_text_location, n_face_crops,
              fc_id = tracking.run_full_tracking(
76            frame, subject_names, text_location, face_crops,
                  start_point_list, end_point_list, run_ia_flag=True
77        )
78
79        # --- Medición de recursos DESPU S del procesamiento ---
80        end_time = time.time()
81        processing_time = end_time - start_time
82        total_processing_time += processing_time
83
84        # --- Almacenar datos de uso de recursos ---
85        cpu_usage_list.append(cpu_percent)
86        ram_usage_list.append(ram_percent)
87
88        # Muestra el video con los cuadros y resultados
89        cv2.imshow('Tracking␣Output', output_frame)
90
91        # Controla la velocidad del video y permite salir con 'q'
92        if cv2.waitKey(int(1000/fps)) & 0xFF == ord('q'):  # Usar FPS
```

```
            del video para un delay más preciso.
93              break
94
95      # Libera los recursos
96      cap.release()
97      cv2.destroyAllWindows()
98
99      # --- Cálculo de métricas de rendimiento ---
100     end_time_total = time.time()  # Tiempo total de finalización
101     total_execution_time = end_time_total - start_time_total
102
103     avg_processing_time = total_processing_time / frame_count if
            frame_count > 0 else 0
104     avg_fps = frame_count / total_execution_time if
            total_execution_time > 0 else 0
105     avg_cpu_usage = sum(cpu_usage_list) / len(cpu_usage_list) if
            cpu_usage_list else 0
106     avg_ram_usage = sum(ram_usage_list) / len(ram_usage_list) if
            ram_usage_list else 0
107     max_ram_usage = max(ram_usage_list) if ram_usage_list else 0
108
109     # --- Impresión de resultados ---
110     print("\n---␣Performance␣Metrics␣---")
111     print(f"Total␣frames␣processed:␣{frame_count}")
112     print(f"Average␣processing␣time␣per␣frame:␣{avg_processing_time:.4
            f}␣seconds")
113     print(f"Average␣FPS:␣{avg_fps:.2f}")
114     print(f"Average␣CPU␣Usage:␣{avg_cpu_usage:.2f}%")
115     print(f"Average␣RAM␣Usage:␣{avg_ram_usage:.2f}%")
116     print(f"Max␣RAM␣Usage:␣{max_ram_usage:.2f}%")
117     print(f"Total␣execution␣time:␣{total_execution_time:.2f}␣seconds")
```

Listing A.2: Script principal con FullTracking y análisis de rendimiento.

```
1  from collections import Counter
2  from typing import Tuple
3
4
5  class HighPassFilter:
6      """ Computes current output from historic
7          results, ignoring low frequency predictions
8          over time for the same id.
9      """
10
11     def __init__(self,
12                  parent,
13                  buffer_size,
14                  run_hpf_flag):
15         # Initiate parent reference
16         self.full_tracking = parent
17         # Number of predictions saved on the dictionary by id
18         self.buffer_size = buffer_size
19         # Initiate dictionary to save historic results
20         self.historic_results = {0: [""]}  # {'id': 'prediction'}
21         # hpf run flag
22         self.run_hpf_flag = run_hpf_flag
23
24     def run_hpf(self, input_id, input_prediction) -> Tuple[int, str]:
25         """ Performs the HighPassFilter algorithm, removing the
26         predictions that have low frequency through time.
27
28         :param input_id: List with current ids in predictions.
29         :param input_prediction: List of current predictions.
30         :return: Tuple with id and predictions filtered.
31         """
32
33         # Add new input key and value to historic results
34         self.update_historic_results(input_id=input_id,
35                                      input_prediction=input_prediction
                                           )
36         # Get most common prediction for input_id key
37         output_id, output_prediction = self.get_mode(input_id=input_id
               )
38
```

```
39         return output_id, output_prediction
40 
41     def update_historic_results(self, input_id, input_prediction) ->
       None:
42         """ Checks if input_id exists in the dictionary
43             that holds historical results. If it exists, its
44             prediction value is added to the list, if it doesn't
45             then the id key is added to the dictionary long with
46             its prediction value.
47 
48         :param input_id: Int with current id key
49         :param input_prediction: Current prediction value for this key
50         :return: Nothing
51         """
52 
53         # Verify if input_id exists in dictionary
54         if input_id in self.historic_results:
55             # Iterate Historic Results list
56             for _id, prediction in self.historic_results.items():
57                 # find input_id key in dictionary
58                 if _id == input_id:
59                     # Add input_prediction to historic list
60                     prediction.append(input_prediction)
61                     # check if list is bigger than limit
62                     if len(prediction) > self.buffer_size:
63                         # Remove first element from list
64                         prediction.pop(0)
65         # if input_key is not on dictionary, then it's added to it
66         else:
67             self.historic_results.update({input_id: [input_prediction
               ]})
68 
69     def get_mode(self, input_id) -> Tuple[int, str]:
70         """ Gets the most frequent prediction for
71             a certain id, from the historical
72             prediction results.
73 
74         :param input_id: Current id for dictionary key
75         :return: output_id is the same as input_id,
76                  and output_prediction is the most
77                  frequent prediction for this id.
```

```
78          """
79
80          # Initiate return values
81          output_id = 0
82          output_prediction = ""
83          # Iterate historic results list
84          for _id, prediction in self.historic_results.items():
85              # find input_id key in dictionary
86              if _id == input_id:
87                  # Get most common value for this id key in dictionary
88                  output_prediction = Counter(prediction).most_common(1)
                        [0][0]
89                  output_id = _id
90                  # Stop the for loop
91                  break
92
93          return output_id, output_prediction
```

Listing A.3: Clase fulltracking.

```
1  import numpy #Operaciones matematicas y manipulacion de matrices
2  from typing import Tuple, Any, List
3  from high_pass_filter import HighPassFilter
4
5
6  class FullTracking: #El init se utiliza para inicializar los
       parametros que estan aqui
7      def __init__(self,
8                   face_classifier_object,
9                   radius_threshold: int,
10                  fc_fd_distance,
11                  elem_in_memory,
12                  buffer_size,
13                  run_hpf_flag):
14         """ Constructor function for the Full Tracking algorithms.
15
16         :param face_classifier_object: Face Classifier instance.
17         :param radius_threshold: Int with radius threshold in pexels.
18         :param fc_fd_distance: Int with the distance (in pexels)
               between the bbox of the face detector and classifier.
19         :param elem_in_memory: Int with the number people in the
               memory.
20         :param buffer_size: Int with the size of the buffer for the
               HighPassFilter.
21         :param run_hpf_flag: Boolean to enable or disable the
               HighPassFilter.
22         """
23
24         # Initiate attributes
25         self.last_run_ia_flag_state = False  # Run ia flag
26         self.last_master_switch_state = True  # Switch to enable or
               disable the AI algorithms
27         self.face_classifier = face_classifier_object  # Reference to
               face_classifier_object for id counter
28         # Define tracking algorithm parameters
29         self.elements_in_memory = elem_in_memory  # in people count
30         self.radius_threshold = radius_threshold  # in pixels
31         self.radius_threshold_fd_fc = fc_fd_distance  # in pixels
32         self.bbox_id_counter_fc = 0
33         # Memory variables to hold historic results
```

```
34          self.fc_historic_centers = [(0, 0)]
35          self.fc_historic_ids = [0]
36          self.fd_historic_centers = [(0, 0)]
37          self.fd_historic_ids = [0]
38          self.buffer_size = buffer_size
39          self.run_hpf_flag = run_hpf_flag
40          # Initiate High Pass Filter object
41          self.hpf = HighPassFilter(parent=self,
42                                    buffer_size=self.buffer_size,
43                                    run_hpf_flag=self.run_hpf_flag)
44
45      def reset_memory(self):
46          # Memory variables to hold historic results
47          self.fc_historic_centers = [(0, 0)]
48          self.fc_historic_ids = [0]
49          self.fd_historic_centers = [(0, 0)]
50          self.fd_historic_ids = [0]
51          # re Initiate High Pass Filter object
52          self.hpf = HighPassFilter(parent=self,
53                                    buffer_size=self.buffer_size,
54                                    run_hpf_flag=self.run_hpf_flag)
55
56  #ESTO LO CAMBIE OJO
57      def run_full_tracking(self, frame: numpy.array, subject_names:
            list, text_location: list, face_crops: list, start_point_list:
            list, end_point_list: list, run_ia_flag: bool) -> tuple:
58
59          """ Performs the algorithm of tracking.
60
61          :param frame: Image in cv2 format.
62          :param subject_names: List with strings with the recognized
               subject names.
63          :param text_location: List with coordinates tuples for the
               text tag location.
64          :param face_crops: List of face crops from the bboxes, in cv2
               format.
65          :param start_point_list: List of coordinates of the top-left
               vertex of the bbox.
66          :param end_point_list: List of coordinates of the bottom-right
                vertex of the bbox.
67          :param run_ia_flag: Boolean flag to enable or disable the AI
```

```
            algorithms.
68      :return: Tuple with several parameters in the format
69      (frame, n_subject_names, n_text_location, n_face_crops, fc_id)
            .
70      """
71
72      # Check current state of run_ia flag
73      if self.last_run_ia_flag_state == run_ia_flag:
74          # Compute center of bbox (fd results)
75          fd_center_list = self.get_fd_centers(start_point_list,
                end_point_list)
76          # Compute center of bbox (fd results)
77          fc_center_list = self.get_fc_centers(face_crops,
                text_location)
78          # Compare fd results with historic fd results
79          frame, fd_id = self.compare_fd_centers(frame,
                fd_center_list)
80          # Compare fc results with historic fc results
81          frame, fc_id = self.compare_fc_centers(frame,
                fc_center_list)
82          # Compare fc and fd results
83          frame, n_subject_names, fc_id =\
84               self.compare_fc_fd_results(frame, fd_id,
                     fd_center_list, fc_id, fc_center_list,
                     subject_names)
85          n_text_location = text_location
86          n_face_crops = face_crops
87      # If run_ia flag was toggled, reset memory
88      else:
89          # Reset memory variables
90          self.fc_historic_centers = [(0, 0)]
91          self.fc_historic_ids = [0]
92          self.fd_historic_centers = [(0, 0)]
93          self.fd_historic_ids = [0]
94          # Reset result variables
95          n_subject_names, n_text_location, n_face_crops, fc_id =
                [], [], [], []
96
97      # Update the run_ia flag variable
98      self.last_run_ia_flag_state = run_ia_flag
99      self.last_master_switch_state = self.face_classifier.
```

```
            master_switch
100
101         return frame, n_subject_names, n_text_location, n_face_crops,
            fc_id
102
103     def compare_fd_centers(self,
104                            frame: numpy.array,
105                            fd_center_list: list) -> Tuple[numpy.array,
                                   list]:
106         """ Compare the bbox centers that result from the face
            detector with its historical
107         results.
108
109         :param frame: Image in cv2 format.
110         :param fd_center_list: List of tuples with the bbox centers.
111         :return: Tuple with the input frame and a list with the
            current ids.
112         """
113
114         # Initiate list
115         fd_id = []
116         # Iterate current center list
117         for fd_center in fd_center_list:
118             coincidence_flag = False
119             # Iterate historic center list
120             for index, fd_historic_center in enumerate(self.
                fd_historic_centers):
121                 # Compute euclidean distance
122                 dist = numpy.linalg.norm(numpy.array(fd_center)-numpy.
                    array(fd_historic_center))
123                 # Compare distance
124                 if dist <= self.radius_threshold:
125                     # Assign historic id to current id
126                     fd_current_id = self.fd_historic_ids[index]
127                     fd_id.append(fd_current_id)
128                     coincidence_flag = True
129                     break
130             if not coincidence_flag:
131                 fd_current_id = self.face_classifier.id_counter
132                 fd_id.append(fd_current_id)
133                 self.face_classifier.id_counter += 1
```

```
134         for index, fd_center in enumerate(fd_center_list):
135             self.fd_historic_centers.append(fd_center)
136             self.fd_historic_ids.append(fd_id[index])
137             if len(self.fd_historic_centers) >= self.
                elements_in_memory:
138                 self.fd_historic_centers.pop(0)
139                 self.fd_historic_ids.pop(0)
140
141         return frame, fd_id
142
143     def compare_fc_centers(self,
144                            frame,
145                            fc_center_list) -> Tuple[numpy.array, list
                               ]:
146         """ Compare the bbox centers that result from the face
                classifier with its historical
147         results.
148
149         :param frame: Image in cv2 format.
150         :param fc_center_list: List of tuples with the centers of the
                bboxes from the face classifier.
151         :return: Tuple with the input frame and a list with the
                current ids.
152         """
153
154         fc_id = []
155         # Iterate current center list
156         for fc_center in fc_center_list:
157             coincidence_flag = False
158             # Iterate historic center list
159             for index, fc_historic_center in enumerate(self.
                fc_historic_centers):
160                 # Compute euclidean distance
161                 dist = numpy.linalg.norm(numpy.array(fc_center)-numpy.
                    array(fc_historic_center))
162                 # Compare distance
163                 if dist <= self.radius_threshold:
164                     # Assign historic id to current id
165                     fc_current_id = self.fc_historic_ids[index]
166                     fc_id.append(fc_current_id)
167                     # Draw current id in center of bbox
```

```
168                     coincidence_flag = True
169                     break
170             if not coincidence_flag:
171                 fc_current_id = self.bbox_id_counter_fc
172                 fc_id.append(fc_current_id)
173                 self.bbox_id_counter_fc += 1
174         for index, fc_center in enumerate(fc_center_list):
175             self.fc_historic_centers.append(fc_center)
176             self.fc_historic_ids.append(fc_id[index])
177             if len(self.fc_historic_centers) >= self.
                elements_in_memory:
178                 self.fc_historic_centers.pop(0)
179                 self.fc_historic_ids.pop(0)
180 
181         return frame, fc_id
182 
183     def compare_fc_fd_results(self,
184                               frame,
185                               fd_id,
186                               fd_center_list,
187                               fc_id,
188                               fc_center_list,
189                               subject_names) -> Tuple[numpy.array,
                                  list[str], list[int]]:
190         """ Compare the bbox centers that result from the face
                classifier and the face detector.
191 
192         :param frame: Image in cv2 format.
193         :param fd_id: List of current ids of the face detector
194         :param fd_center_list: List of centers of the current ids from
                 the face detector
195         :param fc_id: List of current ids of the face classifier
196         :param fc_center_list: List of centers of the current ids from
                 the face classifier
197         :param subject_names: List of names of the subjects detected
                and identified
198         :return: Tuple with the input image in cv2 format (frame), the
                 list of subject names
199         after the HighPassFilter (hpf_subject_names), and the list of
                ids of these subjects
200         after the HighPassFilter (hpf_fc_id).
```

```
201         """
202 
203         # Compute distance between current fc and fd centers
204         for index, fc_center in enumerate(fc_center_list):
205             for index2, fd_center in enumerate(fd_center_list):
206                 # Compute euclidean distance
207                 dist = numpy.linalg.norm(numpy.array(fc_center) -
                        numpy.array(fd_center))
208                 # Compare distance with threshold
209                 if dist <= self.radius_threshold_fd_fc:
210                     # fc_id follows fd_id
211                     fc_id[index] = fd_id[index2]
212                     break
213 
214         # Initiate high-pass-filtered-results lists
215         hpf_subject_names = []
216         hpf_fc_id = []
217         # Apply hpf algorithm to tracking results
218         for index, pred_id in enumerate(fc_id):
219             # verify subject_names has this index
220             if len(subject_names) > index:
221                 # Run hpf
222                 hpf_id, hpf_pred = self.hpf.run_hpf(pred_id,
                        subject_names[index])
223                 # Append results to list
224                 hpf_fc_id.append(hpf_id)
225                 hpf_subject_names.append(hpf_pred)
226 
227         return frame, hpf_subject_names, hpf_fc_id
228 
229     # noinspection PyMethodMayBeStatic
230     def get_fc_centers(self, face_crops, text_location):
231         """ Computes a list with the center points of the bboxes
232             obtained with the face classifier algorithm, which
233             returns a set of start and end points for such bboxes.
234 
235         :param face_crops: List of face crops from the bboxes.
236         :param text_location:  List with coordinates of the bboxes.
237         :return: List with center points of the bboxes.
238         """
239 
```

```
240         # Initiate empty list for center points
241         center_point_list = []
242         # Iterate list with face_crops
243         for index in range(len(face_crops)):
244             # Get start point from list with index
245             start_point = text_location[index]
246             # Get end point from list with index
247             end_point = text_location[index]
248             # Compute center point from start and end bbox points
249             center_point = (end_point[0] + int(0.5 * (start_point[0] -
                    end_point[0])),
250                             start_point[1] + int(0.5 * (end_point[1] -
                                    start_point[1])))
251             # Append center point calculated to the list
252             center_point_list.append(center_point)
253 
254         return center_point_list
255 
256     # noinspection PyMethodMayBeStatic
257     def get_fd_centers(self, start_point_list, end_point_list):
258         """ Computes a list with the center points of the bboxes
259             obtained with the face detection algorithm, which
260             returns a set of start and end points for such bboxes.
261 
262         :param start_point_list: List with start points of bbox
263         :param end_point_list:  List with end points of bbox
264         :return: List with center points of bbox
265         """
266 
267         # TODO: Put code to add null center when no bbox are detected
                so
268         #       the memory doesn't hold results that are too old!!
269 
270         # Initiate empty list for center points
271         center_point_list = []
272 
273         if not len(start_point_list):
274             center_point_list.append((-1000, -1000))
275             return center_point_list
276 
277         # Iterate start point list (a.k.a. fd results)
```

```
278         for index in range(len(start_point_list)):
279             # Get start point from list with index
280             start_point = start_point_list[index]
281             # Get end point from list with index
282             end_point = end_point_list[index]
283             # Compute center point from start and end bbox points
284             center_point = (end_point[0] + int(0.5 * (start_point[0] -
                    end_point[0])),
285                             start_point[1] + int(0.5 * (end_point[1] -
                                    start_point[1])))
286             # Append center point calculated to the list
287             center_point_list.append(center_point)
288
289         return center_point_list
```

Listing A.4: Clase HighPassFilter

```
1
2  import numpy as np
3  import cv2
4  from ultralytics import YOLO
5  from sort import Sort
6  import time
7  import psutil  # Importante para el monitoreo de recursos
8  import os
9
10 if __name__ == '__main__':
11     cap = cv2.VideoCapture("videos/people.mp4")
12
13     model = YOLO("yolov11n-face.pt")  # Cargar modelo v8
14
15     tracker = Sort()  # Inicializar el tracker
16
17     # --- Variables para el análisis de rendimiento ---
18     frame_count = 0
19     total_processing_time = 0
20     cpu_usage_list = []
21     ram_usage_list = []
22     start_time_total = time.time() #Para el tiempo total de ejecución
23
24     # --- Obtener información del video ---
```

```
25      fps = cap.get(cv2.CAP_PROP_FPS)
26      width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
27      height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
28      print(f"Video FPS: {fps}, Resolution: {width}x{height}")
29
30      while cap.isOpened():
31          status, frame = cap.read()
32
33          if not status:
34              break
35
36          frame_count += 1
37
38          # --- Medición de recursos ANTES del procesamiento ---
39          start_time = time.time()
40          cpu_percent = psutil.cpu_percent(interval=None)  # Uso de CPU
               instantáneo
41          ram_percent = psutil.virtual_memory().percent      # Uso de
               RAM
42
43          # Redimensionado de frames
44          frame_resized = cv2.resize(frame, (640, 360))
45          results = model(frame_resized, verbose=False)  # verbose=False
                para evitar salida extra
46
47          for res in results:
48              filtered_indices = np.where(res.boxes.conf.cpu().numpy() >
                    0.5)[0]
49              boxes = res.boxes.xyxy.cpu().numpy()[filtered_indices].
                   astype(int)
50
51              tracks = tracker.update(boxes)
52              tracks = tracks.astype(int)
53
54              for xmin, ymin, xmax, ymax, track_id in tracks:
55                  cv2.putText(img=frame, text=f"Id: {track_id}", org=(
                        xmin, ymin - 10),
56                              fontFace=cv2.FONT_HERSHEY_PLAIN, fontScale
                                =2, color=(0, 255, 0), thickness=2)
57                  cv2.rectangle(img=frame, pt1=(xmin, ymin), pt2=(xmax,
                        ymax), color=(0, 255, 0), thickness=2)
```

```
58
59          # --- Medición de recursos DESPU S del procesamiento ---
60          end_time = time.time()
61          processing_time = end_time - start_time
62          total_processing_time += processing_time
63
64          # --- Almacenar datos de uso de recursos ---
65          cpu_usage_list.append(cpu_percent)
66          ram_usage_list.append(ram_percent)
67
68
69
70          cv2.imshow("frame", frame)
71
72          if cv2.waitKey(1) & 0xFF == ord("q"):
73              break
74
75      cap.release()
76      cv2.destroyAllWindows()
77
78      # --- Cálculo de métricas de rendimiento ---
79      end_time_total = time.time() #Tiempo total de finalizacion
80      total_execution_time = end_time_total - start_time_total
81
82      avg_processing_time = total_processing_time / frame_count if
            frame_count > 0 else 0
83      avg_fps = frame_count / total_execution_time if
            total_execution_time>0 else 0
84      avg_cpu_usage = sum(cpu_usage_list) / len(cpu_usage_list) if
            cpu_usage_list else 0
85      avg_ram_usage = sum(ram_usage_list) / len(ram_usage_list) if
            ram_usage_list else 0
86      max_ram_usage = max(ram_usage_list) if ram_usage_list else 0
87
88      # --- Impresión de resultados ---
89      print("\n--- Performance Metrics ---")
90      print(f"Total frames processed: {frame_count}")
91      print(f"Average processing time per frame: {avg_processing_time:.4
            f} seconds")
92      print(f"Average FPS: {avg_fps:.2f}")
93      print(f"Average CPU Usage: {avg_cpu_usage:.2f}%")
```

```
94     print(f"Average RAM Usage: {avg_ram_usage:.2f}%")
95     print(f"Max RAM Usage: {max_ram_usage:.2f}%")
96     print(f"Total execution time: {total_execution_time:.2f} seconds")
```

Listing A.5: Script completo de YOLO+SORT