AUTENTICACIÓN POR RECONOCIMIENTO FACIAL

Fecha: 09/01/2025

Biometría

Memoria del trabajo final de Visión por Computador, trata la autenticación mediante reconocimiento facial haciendo uso de embeddings, uso de modelos entrenados con YOLO y algunas funcionalidades como similitud respecto a otros usuarios, predicciones de género, edad, etnia y emoción con Deepface, etc.

Alejandro Rodríguez Mesa
Alejandro.rodriguez145@alu.ulpgc.es

Índice

1. Motivación/argumentación del trabajo

La biometría es una disciplina que me despierta cierto interés desde hace tiempo. Probablemente debido a mi interés en la ciberseguridad, que ha ido creciendo recientemente, y entender un poco cómo funciona en el ámbito del reconocimiento facial me parecía interesante. En especial los embeddings, ya que me parece fascinante poder transformar características físicas en datos numéricos utilizables.

La autenticación basada en biometría facial es una herramienta poderosa para mejorar la seguridad y conveniencia en sistemas informáticos. La facilidad de utilizar un rostro en lugar de contraseñas o tokens físicos plantea un avance significativo en la experiencia del usuario, minimizando riesgos asociados con olvidos o robos de credenciales.

En términos de aplicaciones futuras, el reconocimiento facial puede revolucionar áreas como el control de accesos, la personalización de servicios, el monitoreo de seguridad y hasta el análisis demográfico en entornos urbanos. Sin embargo, al abordar estas posibilidades, surge la incertidumbre sobre cómo afectan factores externos, como la iluminación o las expresiones faciales. Y para mitigar esto surge el FQA (Face Quality Asessment), cuya aplicación también fue parte de las razones por las que elegí este trabajo.

Respecto a la ciberseguridad, un área de especial interés en este trabajo han sido las vulnerabilidades inherentes al reconocimiento facial, en particular los ataques de presentación (spoofing), donde un atacante intenta engañar al sistema utilizando fotografías, videos o máscaras. Este desafío resalta la necesidad de desarrollar mecanismos robustos para identificar y prevenir estos intentos maliciosos, lo que podría incluir la integración de técnicas de detección de vida (liveness detection) y análisis de textura facial. Debido a cuestiones de tiempo, no pude explotar demasiado esta faceta, pero mediante modelos entrenados para detectar objetos que podrían ser sujetos de spoofing, se obtuvieron resultados aceptables.

En conclusión, las razones principales por las que me decidí a realizar un proyecto relacionado con autenticación por reconocimiento facial, fueron las posibles futuras aplicaciones en ciberseguridad, mi interés por entender un poco más los embeddings, y la aplicación de FQA.

Por último, también quería entender un poco mejor qué tan lejos estamos de que las máquinas puedan identificar personas con la misma precisión y fiabilidad que un ser humano,

2. Objetivo de la Propuesta

El objetivo principal del proyecto era desarrollar una aplicación que permita:

- Registrar usuarios mediante la captura de datos biométricos faciales.
- Autenticar usuarios basándose en sus características faciales.
- Aplicar FQA (posición de la cabeza, iluminación…) y algún modelo de detección de objetos.
- Hacer una interfaz gráfica sencilla.
- Hacer alguna funcionalidad utilizando las caras guardadas.

Para el último punto, en el proceso me di cuenta de que sería interesante ofrecer análisis como simetría facial, predicciones acerca de la persona (edad, género, raza, etc) y estadísticas de similitud con otros usuarios.

3. Descripción Técnica del Trabajo Realizado

Antes de pasar a describir lo realizado mediante código, me gustaría dejar claro el concepto de embeddings, que son representaciones vectoriales de una cara, generadas por modelos de aprendizaje profundo, que encapsulan características únicas de una persona, transformando la cara detectada en un conjunto de números.

El mayor desafío a la hora de generar estos embeddings es que las imágenes se vean afectadas por efectos externos, como iluminación, objetos que ocultan parcialmente la cara, etc.

Para mitigar estos desafíos, aparece el concepto de Face Quality Assessment (Evaluación de Calidad Facial), que permite descartar imágenes con condiciones no ideales, como mala iluminación o ángulos desfavorables, y elegir la que mejores métricas presenten. Esta técnica busca asegurar que los datos ingresados al sistema sean consistentes y de alta calidad, mejorando así su fiabilidad.

En el caso de este trabajo, como no se prevee que se haga al aire libre el inicio de sesión o registro, se puede asumir que la iluminación será estable. Aunque puede cambiar entre el login y el register. Por lo que será suficiente como FQA hacer detección de objetos para reducir casos en los que afecte la oclusión, además de mantener ojos abiertos y boca cerrada (evitando muecas que generen diferentes embeddings). Y aplicar enhancing para hacer lo más similares posibles las imágenes de register y login, a pesar de cambios de iluminación.

1. Video de Inicio (Intro)

Al lanzar la aplicación, se muestra inicialmente un video de introducción. A nivel técnico:

- Se carga un archivo de video (video.mp4) usando cv2.VideoCapture(video_path).
- Simultáneamente, se reproduce un archivo de audio (musica-inicial.mp3) a través de un canal de audio específico de Pygame (self.intro_channel).
- Para mostrar el video en un Label de tkinter, se lee cada fotograma (frame) usando cap.read(), se hace un resize a (800, 600) y se convierte a RGB para luego integrarse en un PhotoImage de la librería PIL.
- El botón "Saltar Intro" permite al usuario omitir esta secuencia. Al pulsarlo, se detiene el audio y se libera el recurso de captura de video, pasando directamente a la Pantalla de Inicio.

Este video fue hecho en Glitch con three.js en la asignatura de Informática Gráfica, es una animación con shaders. Se puede acceder al código aquí

Aspectos clave:

- Uso de *cv2.VideoCapture* para leer el video fotograma a fotograma.
- Reproducción de audio en paralelo con *pygame.mixer.Sound*.
- Conversión de fotogramas OpenCV (BGR) a PIL (RGB) para incrustarlos en un widget de tkinter.
- Al finalizar el video (o al saltarlo), se detiene el canal de audio de la intro y se pasa a la siguiente pantalla.

2. Pantalla Inicial

Tras finalizar (o saltar) la intro, se muestra la pantalla inicial, en la que el usuario elige entre "Log In" o "Register".

- Se carga una imagen de fondo y se ajusta al Frame con *bg_label.place(...)*.
- Se ubican dos botones principales:
  - Login: Llama al método *go_to_login()*.
  - Sign Up / Register: Llama al método *go_to_signup()*.

  Traté de usar imágenes png sin fondo como botones de login y register, pero tras reiterados intentos no pude conseguirlo. Parece que Tkinter no ofrece transparencia en imágenes, siempre aplica un fondo a modo recuadro en la posición del botón. Se puede cambiar el color del fondo, pero no se puede trabajar con la transparencia, o al menos no encontré manera de hacerlo.

Esta pantalla es simplemente la UI base y la puerta de entrada a las otras funcionalidades.

3. Proceso de Registro (Sign Up)

Si el usuario pulsa en Register, se sigue la siguiente secuencia:

1. **Ingreso de nombre de usuario**
    - Se solicita un nombre de usuario en un Entry.
    - Se valida que el Entry no esté vacío.
    - Se comprueba que no exista ya una carpeta con el mismo nombre en usuarios/. Si existe, se lanza un *messagebox.showerror(...)* para avisar que debe elegir otro nombre.
2. **Creación del entorno de usuario**
    - Si el nombre no existe, se crea una carpeta usuarios/<username> para almacenar las fotos y el archivo de embeddings (*embeddings.json*).
3. **Verificación de la cámara**
    - Se muestra un loader/gif hasta que la cámara esté lista.
    - Cuando la cámara está OK, se empieza la captura en vivo.
4. **Reproducción de audio inicial**
    - Se pausa la música de fondo y se reproduce un audio de instrucción (self.audio_creador) cuando el registro comienza. Luego se reanuda la música de fondo al terminar ese audio.
5. **Captura de imágenes (Se aplican varias condiciones para obtener las mejores posibles, a modo FQA)**
    - Se muestra la cámara en pantalla. Para ello, la clase App abre la cámara (cv2.VideoCapture(0)) en un hilo independiente.
    - Se despliega un óvalo (ellipse) en la pantalla para guiar la posición de la cabeza. Esto es simplemente para conseguir en distintas posiciones la cara, al principio había hecho que deban estar las 4 esquinas de la bounding box dentro del ovalo, pero en ocasiones, aunque la face mesh esté dentro. Se hace un bucle de lectura de fotogramas:
    - Se realiza detección de la cara usando Mediapipe FaceDetection.
    - Se comprueba que haya una sola cara.
    - Se ejecuta un modelo YOLO para verificar si la persona tiene gafas o mascarilla y, de ser así, se pide que se las retire. En el register no añadí la detección de spoofing.
    - Se calcula la pose de la cabeza (head pose) mediante resolución de PnP en 3D (solvePnP). Debe estar de frente.

- Se revisa la condición de ojos abiertos y boca cerrada. Esto se logra con los landmarks de Face Mesh y el cálculo de EAR (Eye Aspect Ratio) para los ojos, y la distancia entre landmarks superior e inferior de la boca.
- Si alguna condición no se cumple, se muestran mensajes de error en pantalla y/o se reproducen audios de error:
- Se produce el audio de “ojos” si están cerrados. O el de “boca” si la boca está abierta. Son audios que avisan de un comportamiento errático.
- Solo cuando todas las condiciones se cumplen de forma estable durante 2 segundos (self.required_stable_seconds), se captura la imagen, se recorta la cara, se aplica “enhancing” o ecualización de histograma, y se guardan dos archivos:
- Una foto sin procesar (foto_n_unproc.png). No se usa como tal para ninguna funcionalidad posterior, pero me parecía correcto mantenerla para compararla con la procesada.
- Una foto procesada (foto_n.png). Usada para comparación de embeddings posteriormente.
- El óvalo va cambiando de posición en cada iteración (cinco en total, definidas en self.ellipse_offsets).

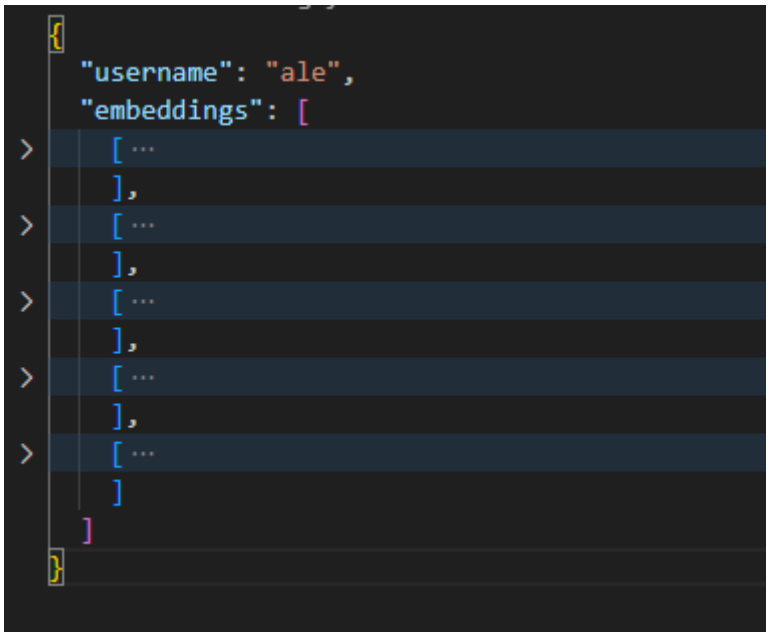
6. Control de cancelación de registro

- Si durante el proceso de captura el usuario cierra la app o pulsa Cancelar, el método *cancel_signup()* elimina la carpeta creada para ese usuario, de modo que no quede un usuario a medio registrar sin tener todas las fotos y embeddings generados.

7. Generación de embeddings

- Una vez que se han capturado las 5 imágenes o las que definamos como self.max_images, se llama a generate_embeddings().
- Para cada foto procesada, se usa DeepFace.represent(...) con el modelo "Facenet" para obtener un vector de 128 dimensiones.

- Se guardan todos los embeddings en un archivo JSON llamado embeddings.json en la carpeta del usuario, con la estructura:



8. **Fin del registro**
   - Se notifica al usuario con un messagebox.showinfo(...) que el registro finalizó.
   - Se regresa a la pantalla inicial.

4. Proceso de Login

Si el usuario selecciona Login en la pantalla principal, la secuencia es la siguiente:

1. **Verificación de la cámara**
   - Se muestra un loader/gif hasta que la cámara esté lista.
   - Cuando la cámara está OK, se empieza la captura en vivo.
2. **Detección de restricciones**
   - Similar al registro, se comprueba:
   - Spoofing usando un modelo YOLO distinto (self.yolo_model2), entrenado para detectar si hay un dispositivo o fotografía usada para engañar al sistema.
   - Gafas o mascarilla usando el primer modelo YOLO (self.yolo_model).
   - Número de caras (debe ser una sola).
   - Cabeza de frente (pose “Frente”).
   - Si se detecta spoofing, se avisa en la pantalla con “Spoofing Detectado”.
   - Si hay gafas o mascarilla, se avisa que se las quite.
   - Todo esto se hace en cada fotograma. Mientras alguna restricción no se cumpla, no se avanza.

3. **Verificación de parpadeos (liveness)**
    - Se lleva un contador de parpadeos (self.blink_count). Cada vez que se detecta un cambio de estado (de ojos cerrados a abiertos), se incrementa.
    - El usuario debe parpadear unas 3 veces para demostrar que es una persona real interactuando (liveness).
    - Debe estar mirando al frente para que cuenten los pestañeos, se reinician si no está la cabeza mirando a esa dirección.
4. **Comprobación de boca cerrada y ojos abiertos**
    - Después de los 3 parpadeos, se verifica de nuevo que la persona tenga ojos abiertos y boca cerrada. Si no se cumplen, se muestra un mensaje en pantalla indicando lo que falta.
5. **Captura final**
    - Cuando las condiciones se cumplen (3 parpadeos, ojos abiertos y boca cerrada, cabeza de frente, sin spoofing, etc.), se captura el fotograma y se realiza una rotación para alinear los ojos horizontalmente. Esto se hace calculando el ángulo entre los landmarks de los dos ojos y aplicando cv2.getRotationMatrix2D().
    - Se recorta la cara con cierto margen.
    - Se guarda una imagen sin procesar internamente (como login_image_pil_unprocessed) para posteriormente poder hacer predicciones más acertadas y otra con enhancing (ecualizada), que se almacena en login_image_pil para hacer la comparación de embeddings.
6. **Generación de embedding y comparación**
    - Se genera un embedding con el modelo "*Facenet*" a partir de la foto de login, usando *DeepFace.represent(...)*.
    - Se buscan todos los usuarios en usuarios/, se lee el archivo '*embeddings.json*' de cada uno, y se compara el embedding recién obtenido con todos los embeddings de cada usuario mediante distancia *coseno (df_verification.find_cosine_distance(...))*. Probé con la distancia Euclidiana y la de Manhattan, pero al final me decidí por usar la distancia coseno a la hora de hacer el inicio de sesión.
    - Se identifica el "mejor match" con la distancia coseno más baja.
    - Se verifica un umbral de 0.25, con lo que la similitud debe ser de al menos 75%. Si la distancia es menor, se considera el usuario coincidente; de lo contrario, es "Desconocido".
7. **Registro de la sesión (log)**

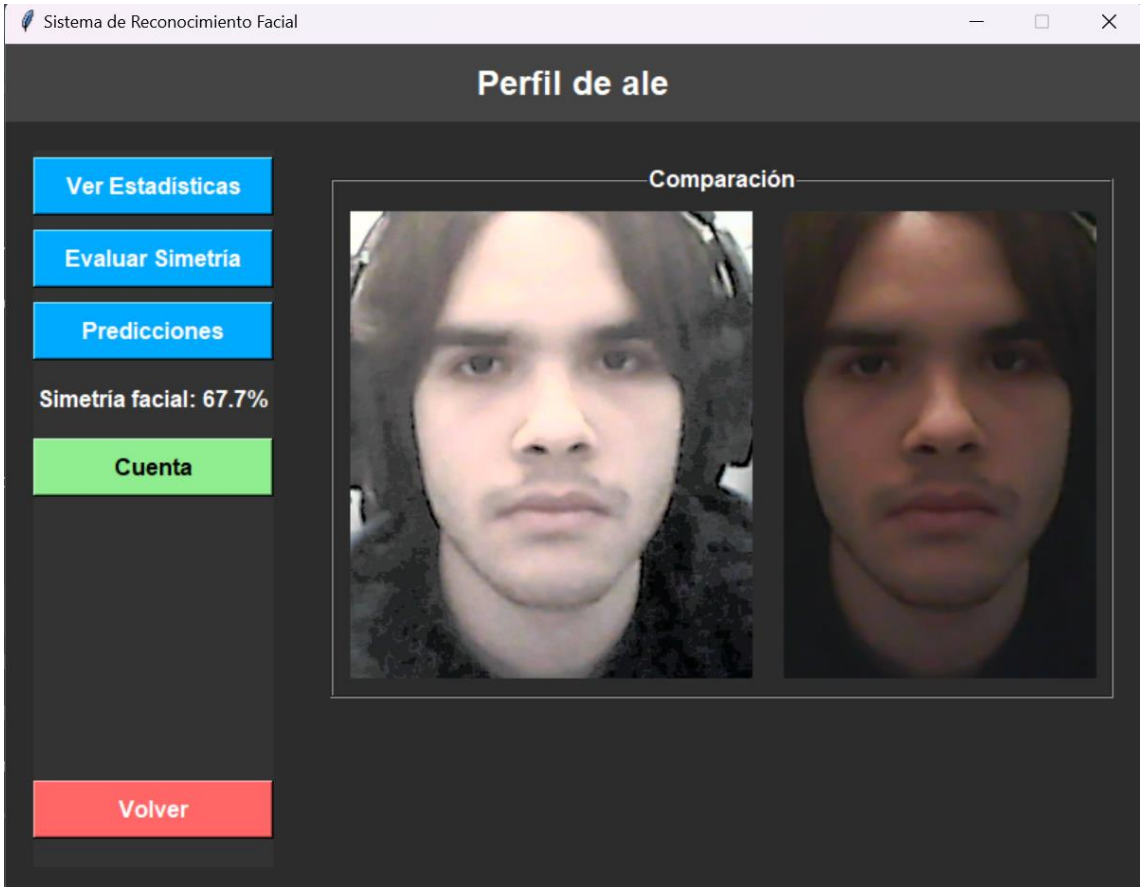
- Se llama *a log_user_login(...),* que escribe una nueva línea (al inicio) en login_history.log con la forma <usuario>,<fecha-hora> para tener histórico de accesos.

8. **Acceso al Perfil**
    - Si se encuentra un match, se muestra un messagebox.showinfo("Login", f"¡Bienvenido, {best_user}!").
    - De lo contrario, messagebox.showinfo("Login", "No se encontró coincidencia. Serás 'Desconocido'."), y se setea self.current_logged_in_user = "Desconocido".
    - Se pasa a la pantalla de perfil.

5. Pantalla de Perfil

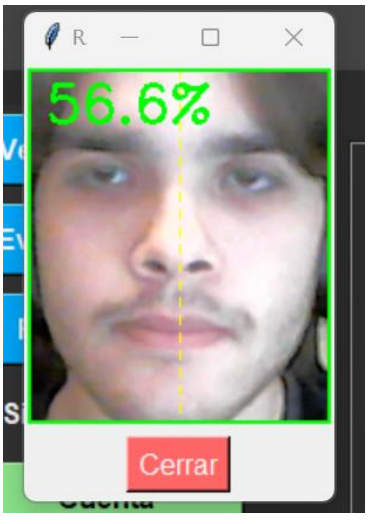
En esta pantalla se muestra:

- Un Label con texto “Perfil de X” donde X es el usuario reconocido o “Desconocido”.
- Dos imágenes, lado a lado:
    - Imagen procesada (enhanced).
    - Imagen sin procesar (unprocessed).
- Botones a la izquierda:
    - Ver Estadísticas (para ver la similitud de embeddings con otros usuarios).
    - Evaluar Simetría (calcula y muestra un porcentaje de cuán simétrica es la imagen del login).
    - Predicciones (estimación de edad, género, raza y emoción con DeepFace.analyze).
    - Cuenta (si el usuario no es “Desconocido”), para ver cuántas veces ha iniciado sesión y cuál fue la última fecha de conexión previa (se lee de login_history.log).
    - En caso de usuario = “Desconocido”, sale un botón “Registrar” que vuelve a la fase de registro.
      Volver, que regresa a la pantalla inicial.



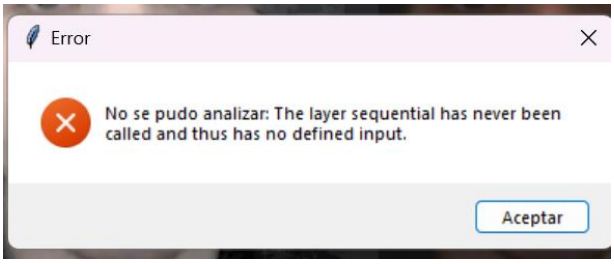
5.1 Simetría

- Se toma la imagen de login procesada (login_image_pil). Se genera la face mesh de la imagen.
- Se recorta la cara mediante los puntos x_left, x_right, y_top, y_bottom encontrados en los landmarks de la malla.
- Se busca el punto medio en el eje X, se corta la imagen en dos mitades y se hace un flip de la mitad derecha para compararla con la izquierda.
- Se calcula la Media Squared Error (MSE) entre ambas mitades, mostrando una aproximación de porcentaje de simetría en pantalla.
- Se abre una ventana (Toplevel) con la imagen y la línea divisoria, mostrando el valor calculado.



5.2 Predicciones (edad, género, raza, emoción)

- Se usa DeepFace.analyze(img_path=face_rgb, actions=['age','gender','race','emotion']), con la imagen sin procesar, para no alterar rasgos. Como nota importante, se debe haber probado primero el código con esta funcionalidad separada para comprobar que funciona, ya que hay ciertas dependencias que se descargan al ejecutarse. Si falla, volver a ejecutar, luego reiniciar el enviroment y volver a ejecutarlo. En caso de que no se haya hecho esto, probablemente saldrá este error:



- Se traducen mediante diccionarios los resultados al español
- Se muestran en un Label los resultados de la predicción.

5.3 Estadísticas (Similitud con otros usuarios)

- Se crea una nueva pantalla (stats_frame) con un encabezado para mostrar:
  - Nombre de usuario
  - Similitud por Distancia Coseno (se calcula 1 - dist_cos y se transforma a %)
  - Similitud usando Distancia Euclidiana (se aplica una escala heurística para convertir a %)
  - Similitud usando Distancia Manhattan (también escalada a %)
- Para cada usuario, se busca el mejor embedding en su lista de embeddings comparado con el embedding de login.
- La línea se formatea con espacios fijos (font=("Courier",12)) para simular una pequeña tabla.
- Se resalta la fila correspondiente al usuario al que corresponde la sesión.

5.4 Botón “Cuenta”

- Muestra cuántas veces ha iniciado sesión el usuario en particular leyendo login_history.log.
- Se cuentan las líneas que empiecen con <username>,.
- Se extrae además la fecha-hora de la última conexión.

5.5 Usuario “Desconocido”

- Se ve el botón “Registrar” que redirecciona allí en vez de “Cuenta”

6. Manejo de Cierre de Aplicación

- Si el usuario cierra la ventana principal (WM_DELETE_WINDOW), se llama a on_closing().
    - Se detienen todos los canales de audio activos.
    - Se verifica si está en proceso de registro (self.in_signup_process); si es así, se llama a cancel_signup() para no dejar la carpeta del usuario a medias.
    - Se libera la cámara con cap.release() y finalmente se destruye la ventana con self.destroy().

4. Fuentes y Tecnologías Utilizadas

- **Lenguaje:** Python.
- **Librerías:**
    - **tkinter**: Para la construcción de la interfaz gráfica (ventanas, Label, Button, Entry, etc.).
    - **cv2** (OpenCV): Captura de video, procesamiento y transformaciones (rotaciones, escalados, conversiones de color).
    - **mediapipe**: Detección de caras y extracción de landmarks (FaceDetection, FaceMesh).
    - **numpy**: Operaciones numéricas y manejo de arrays para distancias y transformaciones.
    - **json**: Lectura y escritura de archivos con embeddings.
    - **shutil**: Para eliminar carpetas de usuarios incompletos.
    - **deepface**: Generación de embeddings (Facenet), verificación de similitud (distancia coseno), y análisis de atributos (edad, género, raza, emoción).
    - **pygame**: Para la reproducción de audio.
    - **Ultralytics.YOLO:** Modelos YOLO para detección de gafas, mascarilla y spoofing.

Recursos Web O Tecnologías:

- **Roboflow**, usado para manejar datasets obtenidos en Roboflow universe, aplicado escalado previo a imágenes, data augmentation, se hizo merge de datasets, y se exportó en formato YOLO 11, para posteriormente hacer el entrenamiento con CUDA.

- **Roboflow Universe,** utilizado para encontrar datasets con imágenes anotadas para entrenar modelos de detección de objetos.
- **CUDA,** Utilizado para el entrenamiento de modelos YOLO

5. Conclusiones y Propuestas de Ampliación

Conclusiones:

- Se cumplieron los objetivos propuestos inicialmente.
- El sistema cumple con los objetivos de registro y autenticación biométrica, ofreciendo resultados precisos y consistentes, aplicando FQA mediante modelos de detección de objetos, y condiciones para hacer las capturas usadas en los embeddings.
- Se aplicó procesado de imágenes (enhancing) para reducir efectos externos en la luz, ecualizando el canal Y, que es el de luminancia o brillo, corrigiendo desequilibrios en la iluminación y resaltando mejor los rasgos faciales, reduciendo el efecto de sombras o exceso de claridad que pueden alterar el reconocimiento.
- La interfaz gráfica facilita el uso del sistema incluso para usuarios sin experiencia técnica, aunque es posible que sea necesario añadir algún audio explicativo, o más mensajes a lo largo del proceso de login o register.

Propuestas de Ampliación:

- **Mejoras en la detección:** Implementar modelos entrenados para detectar oclusión en caras, es decir, si la cara no se ve completa por algún objeto que la oculta parcialmente.
- **Extensión del sistema:** Extender el sistema para que funcione en dispositivos móviles, por ejemplo. Además de utilizar bases de datos como sql en vez de guardar localmente los datos.
- **Autenticación multi-factor:** Integrar el reconocimiento facial con otros factores biométricos, como reconocimiento de voz.
- **Optimización del rendimiento:** Reducir los tiempos de procesamiento mediante hardware dedicado como GPUs o TPUs. En ocasiones la aplicación va a pocos FPS, lo que se nota mucho en la cámara.
- **Mejoras en la interfaz gráfica:** Tkinter es fácil de utilizar, y puede mejorarse la interfaz gráfica utilizándolo, pero tiene varias limitaciones. Habría que valorar si cambiar de librería para mejorar la estética, o continuar usándola.

- **Cambiar la manera en la que se hacen las predicciones con el Deepface.**Analyze(), probablemente se puedan descargar localmente las build para que funcione sin necesidad de ejecutar el código previamente.

6. Indicación de Herramientas/Tecnologías con las que Hubiera Gustado Contar

- Principalmente lo que más me costó fue encontrar datasets con imágenes anotadas adecuadamente. En especial para la oclusión. Traté de usar algunos que encontré, pero los resultados no fueron satisfactorios. Entre desbalanceo de clases e imágenes mal anotadas, los entrenamientos no eran los ideales. En muchos casos se veían durante cientos de imágenes la misma cara (probablemente de los propietarios del dataset), que pueden ser útiles para detectar sus propias caras, pero me preocupa que no funcione correctamente para multitud de caras diferentes.
- GPUs dedicadas para acelerar el procesamiento de imágenes y el entrenamiento de modelos, para probar varios modelos tuve que entrenar varios de ellos, así que incluso con una buena gráfica, se tarda bastante al contemplar la totalidad de los entrenamientos.
- Habría sido muy útil que Mediapipe ofrezca algún parámetro que indique si hubo oclusión (puntos de la mesh imaginaria que son bloqueados por otro objeto), aunque la confianza puede ser útil, no es muy exacta para decidir si hay oclusión o no. Probé a usar modelos como Dlib que son más sensibles a oclusión, pero lo son cuando se tapan puntos clave usados como base para la generación de landmarks.

  Si bien Dlib es más sensible a oclusión, MediaPipe ofrece más landmarks y se pueden hacer mejores aplicaciones con ella con facilidad, con lo que decidí usar MediaPipe.

7. Diario de Reuniones del Grupo

Al ser una única persona no vi necesario hacer un cronograma detallado, pero hablando de cronología, en primer lugar, probé la generación de embeddings y su comparación con DeepFace, lo cual fue más sencillo de lo esperado gracias a las facilidades que ofrece la librería. Posteriormente hice una interfaz gráfica sencilla para probar tkinter. Tras ello, implementé una versión simple del register y el login.

Entrené los modelos de detección de objetos, implementé por separado un código para estimar la posición de la cabeza y fui mejorando gradualmente el register y el login, añadiendo funcionalidades o restricciones a la hora de hacer las capturas. Finalmente traté de mejorar la interfaz mediante un mejor uso de colores, botones y demás.

8. Créditos Materiales No Originales del Grupo

- **Modelos Pre-entrenados:**
    - FaceNet (DeepFace).
    - YOLO para detección de objetos.
- **Recursos externos:**
    - **Datasets**
        - **Glasses**
        - **Masks**
        - **Spoofing**
    - **Imágenes y Música**
        - El video inicial es de mi propiedad (solo el video, el audio no. Hecho con three.js para el trabajo final de la asignatura Informática Gráfica)
        - Música del video inicial
        - Música de fondo de la app
        - Sonido de sacar foto
        - Gif del loader
        - Los botones fueron hechos con la IA de Canva
        - Background Foto
- **Modelos humanos**
    - Agradecer a mi familia por registrarse como usuarios (varias veces) e iniciar sesión durante la fase de desarrollo.

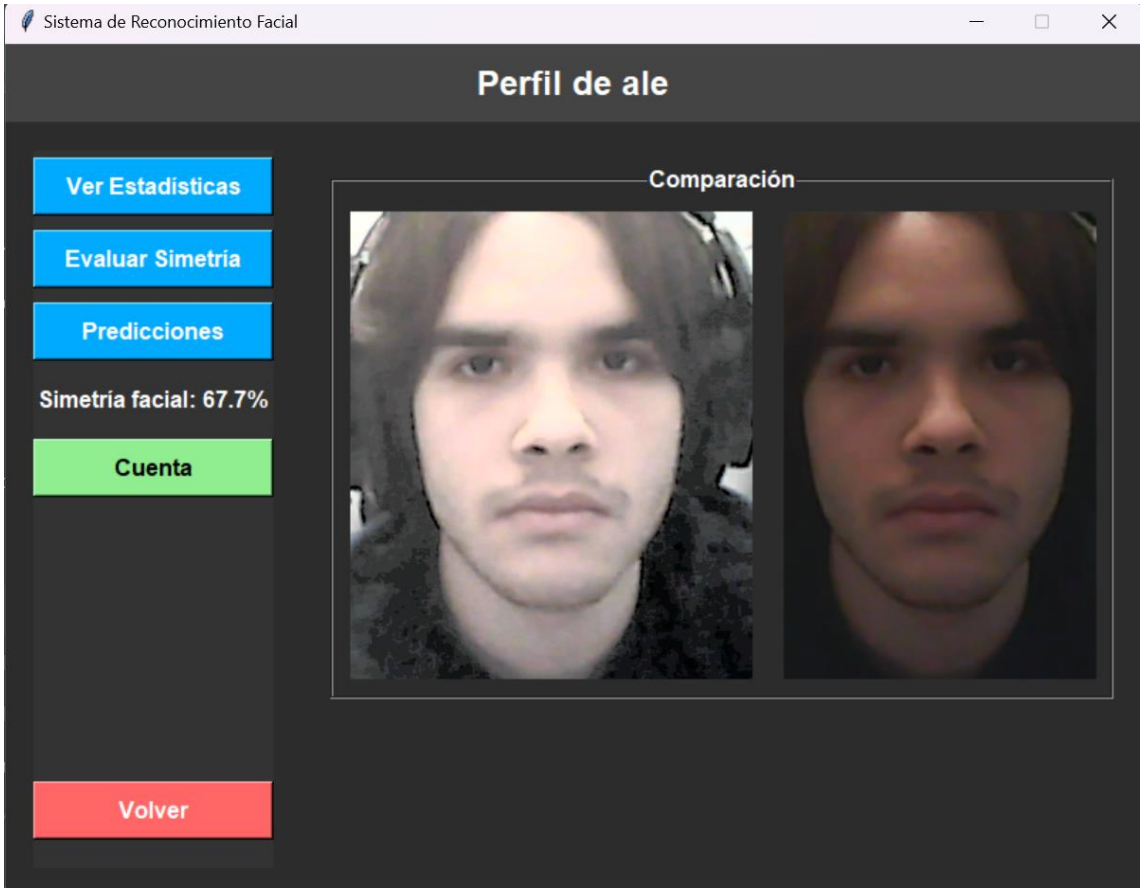
9. Enlace al código

Enlace al cuaderno

10. Imágenes de Entrenamiento

Utilicé roboflow, así que se puede acceder desde roboflow universe a los datasets que acabé usando. El de gafas y máscaras fue tras un merge de 2 datasets, y el de spoofing fue sacado directamente de roboflow Universe, solo eliminé algunas imágenes que no necesitaba. A ambos les apliqué preprocesamiento (reescalado e ignorar imágenes con clase nula) y data augmentation (saturación, luminicidad, y blur.)

11. Imagen "Carátula" del Trabajo



12. Vídeo Resumen de Venta del Trabajo

Se encuentra en el README del github

13. Material Adicional de Interés

Tanto las capturas como la explicación del enviroment para replicar el proyecto se encuentran en el README. Además de la muestra de los resultados del entrenamiento de YOLO para ambos modelos.