BRIEF 18 – Reconnaissance faciale

**📌 État de l’art des techniques de reconnaissance faciale**

La **reconnaissance faciale** est une branche de la vision par ordinateur et de l’intelligence artificielle qui consiste à identifier ou vérifier l’identité d’une personne à partir de son visage. Elle repose sur plusieurs étapes : détection du visage, extraction de caractéristiques (features), puis comparaison avec une base de données.

**🔹 Principales techniques utilisées**

1. **Méthodes traditionnelles (avant le deep learning)**
   * **Eigenfaces / Fisherfaces (PCA, LDA)** : basées sur la réduction de dimension et l’analyse statistique des traits du visage.
   * **LBPH (Local Binary Patterns Histograms)** : exploite des motifs locaux pour représenter les visages.  
     ➝ Avantages : rapides, peu coûteuses en ressources.  
     ➝ Limites : sensibles à la lumière, l’angle de vue et les expressions faciales.
2. **Méthodes modernes (deep learning)**
   * **CNN (Convolutional Neural Networks)** appliqués à la détection et l’encodage des visages.
   * **Systèmes pré-entraînés** :
     + **OpenCV** : bibliothèque de vision par ordinateur offrant la détection faciale via Haar Cascades et DNN.
     + **dlib** : implémente un détecteur de visages basé sur HOG + SVM et un modèle CNN plus précis.
     + **FaceNet (Google)** : encodeur basé sur deep learning pour représenter un visage comme un vecteur de 128 dimensions.
     + **DeepFace (Facebook)** : un des premiers modèles à atteindre des performances proches de l’humain.
     + **face\_recognition (Python)** : bibliothèque très populaire construite sur *dlib*, facile à utiliser pour la détection et la comparaison de visages.
3. **Solutions commerciales et cloud**
   * **Microsoft Azure Face API**, **Amazon Rekognition**, **Google Vision AI** : services clés en main avec API REST.  
     ➝ Avantages : très performants, intégration facile.  
     ➝ Limites : coût, dépendance au cloud, confidentialité des données.

**🔹 Outil choisi pour ce projet**

👉 Dans le cadre de ce projet, nous utiliserons **la bibliothèque face\_recognition en Python**, qui repose sur **dlib** et intègre :

* Détection de visages (HOG ou CNN)
* Encodage des visages en vecteurs numériques
* Comparaison et reconnaissance simple via distances entre encodages

✅ Avantages pour ce projet :

* Open source, gratuit et bien documenté
* Facile à installer et à utiliser (quelques lignes de code suffisent)
* Basé sur des modèles éprouvés (dlib + CNN)
* Compatible avec des projets Flask (application web locale)

**1. Quelles en sont les fonctionnalités ?**

La librairie **face\_recognition** permet :

* Détecter des visages dans une image (localisation par bounding box).
* Extraire des **empreintes faciales** (vecteurs d’encodage numériques).
* Comparer des visages entre eux pour savoir s’il s’agit de la même personne.
* Gérer une base de visages connus (ex. liste d’encodages avec leurs identités).
* Supporter images, vidéos, webcam en temps réel.
* Fonctionne **en local** (pas d’API externe → conforme aux contraintes RGPD du client).

**2. Quelles en sont les entrées/sorties ?**

* **Entrées possibles :**
  + Une image (JPEG, PNG, etc.)
  + Une frame vidéo (via OpenCV par ex.)
  + Une webcam (flux live)
* **Sorties :**
  + Coordonnées des visages détectés (x, y, largeur, hauteur).
  + Encodage du visage (vecteur numérique de 128 dimensions).
  + Résultat booléen / score de similarité (ex. "même personne" ou non).
  + Possibilité de renvoyer l’identité la plus proche dans la base connue.

**3. Cette technologie est-elle suffisante seule pour remplir le cahier des charges ?**

✅ Oui, **face\_recognition** suffit pour ton cahier des charges, car :

* Tout est en **Python**.
* Ça tourne en **local sur serveur** (pas besoin d’API cloud).
* Ça s’intègre dans **Docker** facilement.
* Tu peux exposer le modèle via une API (Flask ou FastAPI).
* Bonne précision pour un POC (Proof of Concept) → largement suffisant pour une première implémentation.

⚠️ Limites :

* Moins performant que les solutions cloud (Microsoft Face API, AWS Rekognition, etc.).
* Pas toujours optimal pour une très grande base de visages (besoin d’optimisation si milliers d’utilisateurs).
* Dépend de la qualité de l’image et de la lumière.

**🔹 Architecture micro-services proposée**

**1. Frontend (UI)**

* **Technologie** : application web (HTML/JS, ou un mini frontend en Streamlit/React).
* **Rôle** :
  + Permettre à l’utilisateur de capturer son visage via la webcam.
  + Envoyer l’image (ou la vidéo) au backend via une requête API.
  + Afficher le résultat (authentification réussie ou non).

**2. Service API (Backend principal)**

* **Technologie** : Flask ou FastAPI (Dockerisé).
* **Rôle** :
  + Recevoir les images envoyées par le frontend.
  + Appeler le micro-service de reconnaissance faciale.
  + Comparer le visage reçu avec la base de visages connus.
  + Retourner un résultat (OK / KO / score de similarité).

**3. Service de Reconnaissance Faciale**

* **Technologie** : Python + face\_recognition.
* **Rôle** :
  + Détecter et encoder les visages.
  + Comparer les encodages.
  + Fournir une réponse brute (vecteur encodage ou booléen "match/no match").
* **Entrée** : image (base64, fichier, ou URL).
* **Sortie** : résultat de comparaison (identité reconnue ou non).

**4. Service Base de Données**

* **Technologie** : PostgreSQL, SQLite ou MongoDB (selon simplicité).
* **Rôle** :
  + Stocker les **visages connus** (nom + encodage).
  + Stocker les logs d’authentification.
* **Entrée** : requêtes SQL (ajout / recherche d’encodages).
* **Sortie** : encodages faciaux pour comparaison.

**5. Orchestration (Docker Compose)**

* Chaque service tourne dans son **conteneur Docker** :
  + frontend-service
  + api-service
  + face-recognition-service
  + database-service
* docker-compose.yml lie les services ensemble (réseau interne).

**🔹 Schéma simplifié (sur papier)**

[ Utilisateur ]

|

v

[ Frontend ] ---> capture webcam + envoi image

|

v

[ API Service (FastAPI/Flask) ]

|

v

[ Micro-service Face Recognition ] ---> compare encodage

|

v

[ Base de données ] ---> stock encodages connus

┌─────────────────────┐

│ Frontend │ (Streamlit)

│ - Page login │

│ - Capture webcam │

└─────────┬───────────┘

│ POST /login

│ user\_id + password

▼

┌─────────────────────┐

│ API │ (Flask)

│ - Vérifie login │

│ - Gère sessions │

│ - Redirige vers │

│ 2FA faciale │

└─────────┬───────────┘

│ POST /verify\_face

│ file image + user\_id

▼

┌─────────────────────┐

│ Recognition │ (Flask/FastAPI)

│ - /register │ Enregistre encodage visage

│ - /verify │ Compare encodages

│ - Stockage fichiers│ (ou DB)

└─────────┬───────────┘

│ Retour JSON

▼

┌─────────────────────┐

│ Base de données │ (optionnel)

│ - Encodages visage │

│ - Info utilisateur │

|  |
| --- |
|  |

Dans ton architecture :

* **frontend (Streamlit)** → c’est l’interface utilisateur.
* **api (Flask/FastAPI)** → c’est la couche métier : elle parle à la base de données **et** au service de reconnaissance faciale.
* **recognition** → c’est le service bas-niveau qui fait juste la détection/encodage des visages.

👉 Donc, le **frontend** doit uniquement dépendre de **api** (et non de recognition), car :

* Le frontend **n’appelle jamais recognition directement**.
* Toutes les requêtes passent par l’API (api\_service), qui elle se charge de contacter recognition\_service.

**2️⃣ Qu’est-ce qu’un fichier .npy ?**

.npy est **le format de fichier natif de NumPy** pour stocker des tableaux (numpy arrays) de manière rapide et efficace.

* Dans ton cas, un encodage facial est **un vecteur numérique de 128 valeurs flottantes** représentant les caractéristiques uniques du visage.
* Le fichier .npy contient ces 128 nombres, sous forme de tableau NumPy.
* Avantages :
  + Lecture et écriture très rapides (np.save et np.load).
  + Préserve le type de données exact (float32 ou float64).
  + Compact et portable.