Laboratoire 1

—

Détection faciale avec OpenCV

—

Par Enzo Di Maria

Département TI - Session d'hiver
2023

Introduction

Les objectifs de ce laboratoire n°1 sont multiples :
— **Détection faciale**
— **Détection des mains**
— Et éventuellement, **reconnaissance faciale**

Le langage choisi est le Python, car il a l'avantage de proposer diverses manières de mener à bien ces objectifs.

Certaines approches sont issues du machine learning, d'autres emploient des pipelines de traitement d'image, d'autres encore des classifieurs.

L'objectif de ce rapport sera de présenter l'application, les approches choisies, d'en expliquer le fonctionnement et les limites.

Table des matières

1 Architecture de l'application

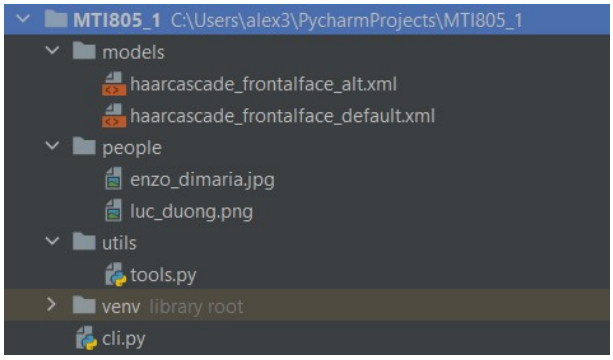

FIGURE 1 – Structure du projet

Le projet dispose de plusieurs sous-dossiers. Mon choix d'utiliser un environnement Python virtuel a généré le dossier *venv*, mais nous ne nous attarderons pas dessus. On compte aussi un dossier *models* renfermant deux modèles en cascade de Haar. Également, on note la présence d'un dossier *people* qui servira pour la reconnaissance faciale, ainsi si l'un des visages détectés est celui de Enzo DI MARIA ou de Luc DUONG, l'application l'affichera. Enfin on compte deux fichiers Python, l'un est dans un dossier *utils*, il contient toutes les fonctions de détection, de reconnaissance ainsi que la boucle principale, puis il y a *cli.py* qui est le point d'entrée du programme. C'est à partir de lui qu'on lance l'application.

Un telle séparation entre les rouages du code et le point d'entrée permet de rendre le programme plus ergonomique pour l'utilisateur.

2 Choix de conception

2.1 *cli.py*

Je passerai assez vite sur le point d'entrée du programme, il s'agit simplement de lancer une fonction **run_webcam(face_reco, face_detect)**. Toutefois, j'aimerais m'attarder sur un choix de conception. Il ne fait aucun sens d'utiliser dans le même temps la fonction à l'origine de la détection faciale et la fonction de reconnaissance faciale. En effet, cette dernière, détecte également les visages. C'est donc soit l'une soit l'autre. Ainsi j'ai utilisé la bibliothèque *argparse* afin de rendre robuste les arguments de la ligne de commande. Donc, quoi qu'il arrive la détection des mains est lancée, mais l'utilisateur devra choisir entre la détection simple des visages (avec l'option *-d*) ou la reconnaissance faciale (*-r*). Dans le cas contraire, des exceptions sont levées.

2.2 *tools.py*

Ce fichier contient des fonctions qui utilisent différentes bibliothèques de traitement d'images pour différentes tâches. Il utilise *opencv-python* (cv2), *MediaPipe* et *face_recognition*.

La fonction **detect_hands** utilise la bibliothèque *MediaPipe* pour détecter les mains dans le champ de la caméra et les dessiner à l'écran. Elle peut en détecter jusqu'à quatre.

La fonction **detect_faces** utilise un classifieur de Haar pour détecter les visages et les entourer d'un rectangle sur l'image originale.

La fonction **recognize_face** utilise la bibliothèque *face_recognition* pour détecter les visages dans une image, et comparer les images connues avec les encodages de visages dans l'image. Si l'un d'eux est reconnu, son nom est affiché. Dans notre base de données on compte deux images : *enzo_dimaria.jpg* et *luc_duong.png*. Notez que cette fonction travaille à partir d'une image réduite à 20% de sa taille d'origine afin d'améliorer les performances du programme.

La fonction **run_webcam** prend en entrée deux paramètres, *face_reco* et *face_detect*, qui détermine si l'application doit utiliser la reconnaissance faciale ou la détection faciale. Il utilise la webcam pour capturer des images et utilise les fonctions mentionnées ci-dessus pour effectuer les tâches spécifiées. Un exception est levée si la capture d'images venait à échouer, par exemple si la webcam est débranchée. Il utilise également les fonctions **concurrent.futures** pour exécuter les tâches de traitement d'images de manière asynchrone. Ainsi si l'une est lente (c'est le cas de la reconnaissance faciale) alors ça n'impactera pas l'autre.

3 Le fonctionnement des approches choisies

3.1 MediaPipe

MediaPipe, un peu comme son nom l'indique utilise une technologie nommée *Pipeline ML* ou *pipeline d'apprentissage automatique*.

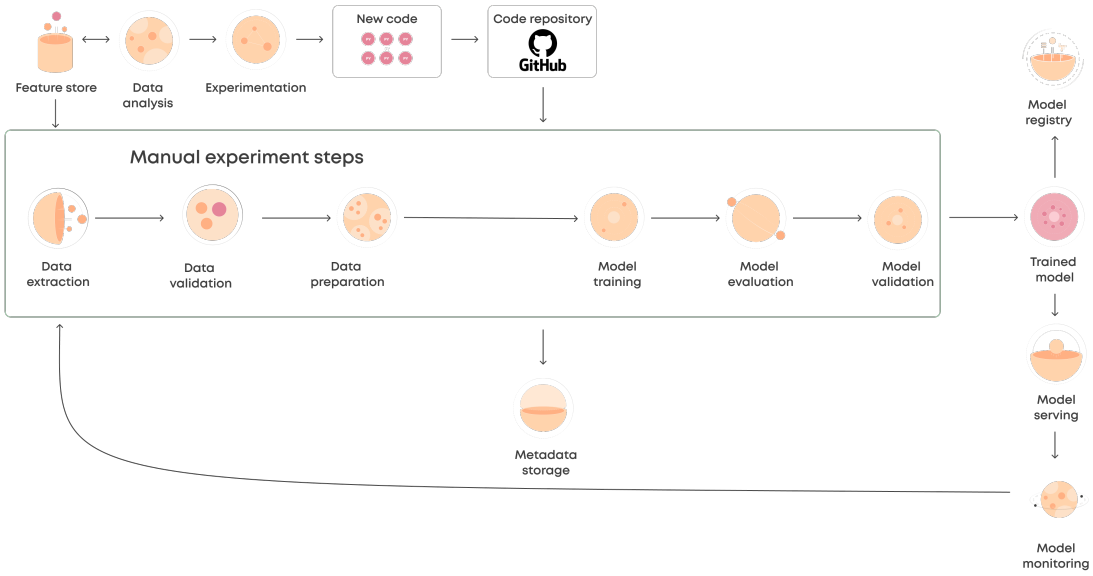

FIGURE 2 – Schéma d'une pipeline MLhttps://valohai.com/assets/img/automatic-pipeline.png

Il s'agit d'un ensemble d'étapes ou de tâches qui sont exécutées de manière séquentielle pour résoudre un problème donné en utilisant l'apprentissage automatique. La pipeline est conçue pour automatiser le processus de mise en place d'un système d'IA, en s'assurant que les données soient traitées de manière efficace et que les modèles sélectionnés et entraînés soient les plus appropriés pour résoudre le problème spécifique.

Dans notre cas particulier, la détection des mains se fait en réalité en deux parties. L'objectif dans un premier temps n'est pas de détecter la main dans son entièreté mais détecter la paume. C'est une opération bien plus simple. Elles sont analysées via des boîtes de délimitation carrées. La documentation raconte que la précision atteint les 95.7%.

Et c'est de ce modèle de paume que découle le modèle de la main, qui comprends les doigts. Ce dernier effectue via une régression une localisation précise de 21 coordonnées de nœud de main 3D à l'intérieur des régions de main détectées. Pour obtenir ces résultats, il a fallu plus de 30000 images annotées manuellement. Puis pour couvrir toutes les poses de main possibles et fournir une supervision supplémentaire sur la nature de la géométrie de la main, une image synthétique haute qualité en utilisant un modèle de main 3D animé a été généré.

Enfin, tout ce qui est rapport au calcul point clés 3D de la main a été confié à un réseau de neurones convolutifs (CNN). Toute cette suite de calculs forme une pipeline ML et permet d'aboutir à un résultat qui fonctionne.

3.2 Les classifieurs de Haar

La bibliothèque *opencv-python* fournit un grand nombre de features, parmi elles il y a la fonction **cv2.CascadeClassifier(file)** qui permet de charger un classifieur en cascade sous la forme d'un fichier *.xml*. A partir de là, il suffit de convertir l'image en nuances de gris et le classifieur détecte les visages présents dans le champ de la caméra.

Mais comment cela fonctionne-t-il ?

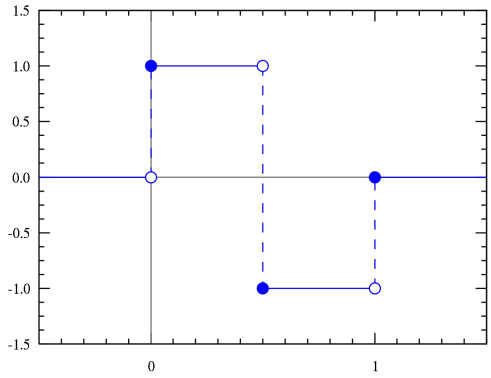

FIGURE 3 – Fonction de Haar
`https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/Haar_wavelet.svg/1200px-Haar_wavelet.svg.png`

Les cascades de Haar sont un type de classifieur utilisé pour détecter des objets dans des images. Ils sont dit "en série" et utilisent des fonctions de base appelées "fonctions de Haar" pour détecter des caractéristiques particulières dans une image, comme les contours d'un visage. Chacune des étapes dans la cascade utilise une fonction de Haar pour éliminer rapidement les régions de l'image qui ne contiennent pas l'objet recherché, jusqu'à ce qu'il ne reste plus qu'une petite région qui contient probablement l'objet.

Un tel modèle se construit en plusieurs étapes :

1. La première est la constitution d'une base de données conséquente.
2. Puis vient la génération des fonctions de Haar. Pour chaque étape de la cascade, on en génère un grand nombre qui vont être utilisées pour détecter les caractéristiques de l'objet. Ces fonctions sont générées en utilisant des régions de l'image qui contiennent l'objet et en comparant ces régions à des régions qui ne contiennent pas l'objet.
3. Ensuite, débute l'entraînement des classifieurs. Pour chaque étape de la cascade, on entraîne un classifieur pour sélectionner les fonctions de Haar les plus efficaces pour détecter l'objet.
4. Vient après, l'évaluation des classifieurs. On les évalue en utilisant un ensemble de données de test qui n'a pas été utilisé pour l'entraînement. Ceux qui ne détectent pas efficacement l'objet sont éliminés et remplacés par de nouveaux.
5. Puis on optimise la cascade en ajustant le nombre d'étapes, les seuils de décision et les poids de chaque classifieurs pour améliorer la précision de la détection de l'objet.
6. Enfin, on enregistre le modèle pour des utilisations ultérieures.

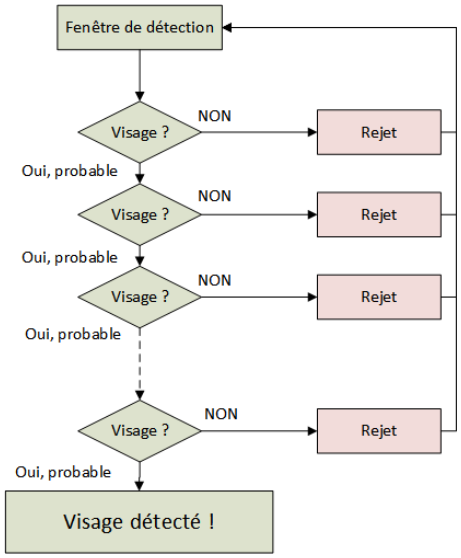

FIGURE 4 – L'étape 4 illustré
`https://www.coxprod.org/domotique/wp-content/uploads/2020/04/cascade_classifieur1.png`

Cette méthode dispose d'un certain désavantage, il est long à apprendre. Plusieurs jours, voire plusieurs semaines sont parfois nécessaires pour construire un modèle correct.

3.3 face_recognition

Dans la documentation officielle sur GitHub, il est évoqué que ce module est basé sur *dlib*, le deep learning et permet une reconnaissance précise à 99.36% d'après le benchmark de *Labeled Faces in the Wild*. On ne trouve cependant pas plus de précision quant à son fonctionnement.

On peut cependant regarder comment *dlib* fonctionne afin d'avoir une idée des rouages de *face_recognition*. A ce titre, on note que *dlib* offre deux façons de reconnaître les visages.

La première utilise l'algorithme *Histogram of Oriented Gradients* (HoG) et *Linear Support Vector Machine* (SVM). En inspectant le code, j'ai pu déduire qu'il s'agissait de la méthode par défaut qui est utilisé par *face_recognition*, car il est également possible d'opter pour une approche via un *CNN* (la deuxième façon de fonctionner de *dlib*). Il s'agit d'une approche rapide mais moins robuste que la seconde. HoG utilise des gradients orientés pour identifier les caractéristiques clés d'un visage, comme les contours et les textures, tandis que SVM utilise une technique de classification pour déterminer si ces caractéristiques correspondent à un visage réel ou non.

La seconde utilise *Max-Margin Object Detection* (MMOD), une technique utilisant des réseaux de neurones convolutifs (CNN) pour reconnaître les visages dans une image, et en identifier les caractéristiques clés comme les contours, les textures et les formes. Ces techniques sont robustes, même dans des conditions difficiles ou sous des angles particuliers, et est accéléré par GPU pour améliorer les performances. Le principe de MMOD est de maximiser la distance (au sens mathématique) entre les visages détectés et les autres objets dans l'image.

Mais puisque j'ai décidé de n'utiliser que mon CPU, cette dernière méthode n'est pas utilisé.

4 Analyse des résultats et conclusion

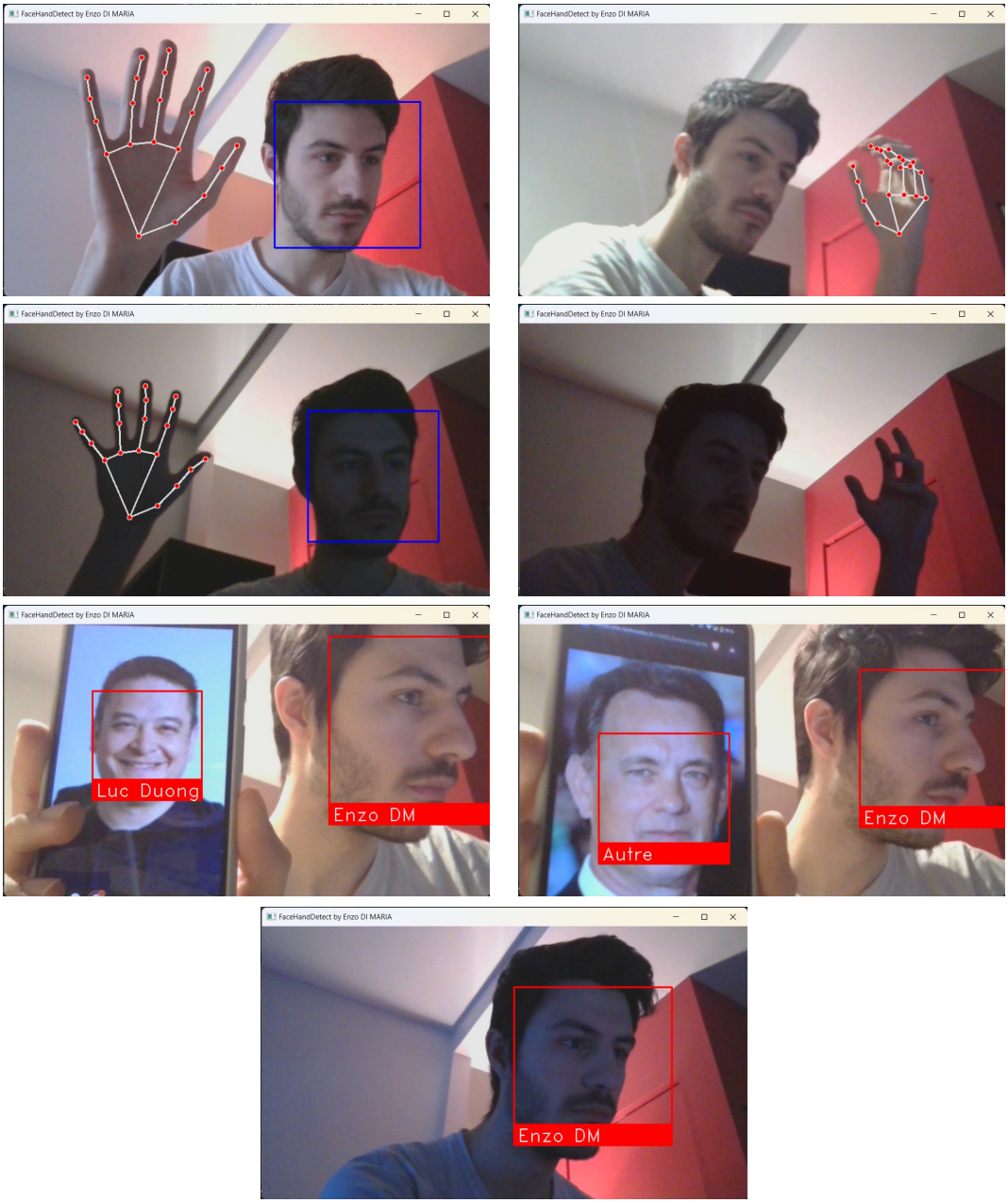

FIGURE 5 – Ensemble d'exécution du programme sous différentes conditions

Dans ces expériences il y a eu trois variables : la luminosité, l'angle/inclinaison du visage et la tenue de la main.

On note qu'en pleine luminosité, presque tout fonctionne très bien. Les mains peu importe leur tenue sont détectées, la reconnaissance de visage est précise peu importe l'angle de ma tête, seule la détection faciale peine lorsque j'incline mon visage.

En faible luminosité, c'est moins simple. Une main parfaitement exposée sera toujours détectée, mais si sa tenue change, le programme aura du mal à être aussi précis. De la même façon, un visage assez droit sera bien détecté par le modèle en cascade, mais dès qu'il est incliné, le programme aura du mal à le détecter. En revanche, la reconnaissance fonctionne assez bien.

Notons aussi que certains choix de conception peuvent expliquer ces résultats.

— L'image de la webcam a été redimensionné en 800×450.
— Les images modèles pour la reconnaissance sont dimensionnées en 300×300 et en 200×200.
— Dans la fonction **recognize_face(frame)**, on travaille à partir d'une image réduite à 20% de la taille d'origine.

Ces ajustements, en plus du multithreading (mais qui lui n'impacte pas les résultats) ont été fait afin d'améliorer les performances d'une application qui dans son usage de reconnaissance rame beaucoup. On ne travaille qu'à un rythme de 1 image par seconde. C'est peu. Cela s'explique par le fait que l'usage de *cuda* et d'une accélération GPU sont fortement conseillés par la documentation officielle de *face_recognition*. Mais pour des raisons de portabilité, cela n'a pas été implémenté.

En conclusion, cette expérience a montré combien l'usage d'un GPU semble nécessaire pour une reconnaissance optimale. Elle a aussi montré l'importance des réseaux de neurones, qui peuvent être bien plus efficace que les autres méthodes de calcul. Si je devais citer un de mes précédents travaux sur la classification d'images via deep learning, je dirais également combien constituer une base de données conséquente et cohérente est important pour l'efficacité du modèle. Car une base de données de qualité permet de s'adapter à différents paramètres, différentes conditions qui dans la vie réelle s'imposent à nous. Dans notre exemple, avec les visages, on peut citer l'inclinaison, l'angle, l'éloignement. Car rarement, dans la vie, une caméra ne pourra détecter des visages bien droit et bien de face. Il faut pouvoir s'adapter à l'imprévisible.

5 Bibliographie

— https://google.github.io/mediapipe/solutions/hands.html, **MediaPipe**
— https://github.com/ageitgey/face_recognition, **face_recognition**
— https://github.com/opencv/opencv-python, **opencv-python**
— https://docs.python.org/3/library/concurrent.futures.html, **concurrent-features**
— https://docs.python.org/3/library/argparse.html, **argparse**
— https://cours.etsmtl.ca/log725/private/contact.html, **Photo de Luc Duong**
— https://valohai.com/machine-learning-pipeline/, **Pipeline ML**
— https://fr.wikipedia.org/wiki/Ondelette_de_Haar, **Fonctions de Haar**
— https://fr.wikipedia.org/wiki/CaractÃŀristiques_pseudo-Haar, **Caractéristiques pseudo-Haar**
— https://www.coxprod.org/domotique/reconnaissance-faciale-tp2-detection-faciale-avec-cascade-de-ha **Les cascades de Haar**
— https://www.width.ai/post/facial-detection-and-recognition-with-dlib, **dlib**
— https://github.com/EnzoN7/Image-classification, **Mon projet de classification d'images** (évoqué en conclusion)