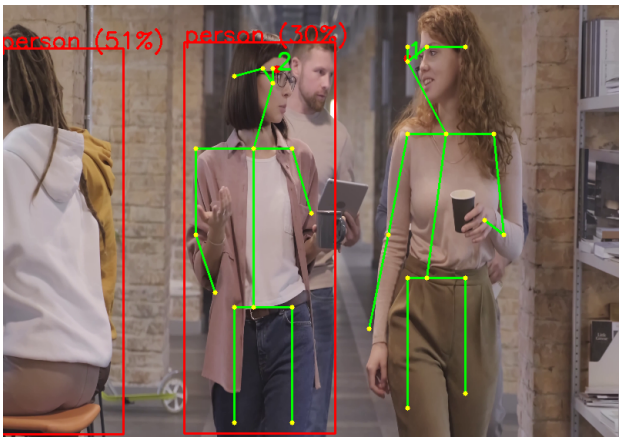
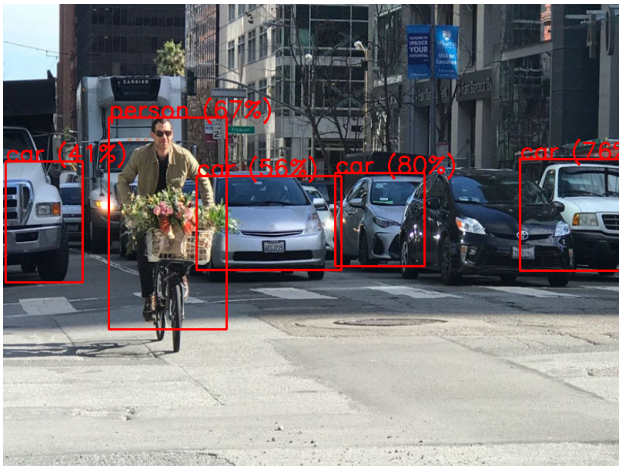
Image et Interaction - Intelligence Artificielle

Cyril Barrelet & Marc Hartley

December 2024





1 Introduction

L'utilisation d'intelligence artificielle est l'un des points les plus importants et utiles dans le traitement d'images et de signaux.

De la détection d'objets, la compréhension de scènes, ou encore l'analyse de vidéos, des algorithmes de plus en plus intelligents (mais complexes) sont mis en œuvre dans nos programmes de tous les jours. Aussi complexes que sont ces algorithmes, presque tout repose néanmoins sur un principe que vous avez déjà rencontré : les convolutions et les opérations morphologiques (et un peu d'huile de coude). En enchaînant de la bonne façon ces opérations, il est possible de faire des merveilles! Aujourd'hui on va utiliser les algorithmes de cascades de Haar et réseaux de neurones profonds pour interagir directement avec une caméra. Heureusement pour nous, le plus gros du travail est déjà calculé, optimisé, affiné par la librairie OpenCV!

1.1 Notre code de base

Pour commencer, nous allons travailler avec un projet Python pré-existant et partagé. Cela signifie que vous pouvez récupérer les fichier du projet sur GitHub. Pour télécharger ce projet :

- Ouvrez un terminal.

- Déplacez-vous (avec `cd`) dans votre dossier de codes (par exemple "IG3/FASE/Image/")
- Récupérez le projet en lançant la commande
  `git clone https://github.com/marchartley/PoseEstimation.git`
- Un dossier "PoseEstimation" a du être créé. Ce sera la "racine du projet".

Donc dans le "main.py", reprenons le code de base:

```python
import cv2
import numpy as np

def main():
    EPSILON = 30

    video_path = "chemin/vers/une/video.mp4"
    cap = cv2.VideoCapture(video_path)
    while cap.isOpened():
        ret, img = cap.read()
        if not ret:
            break
        img = cv2.resize(img, (800, 600))
        img = cv2.flip(img, 1)

        cv2.imshow("Resultat", img)

        key = cv2.waitKey(EPSILON) & 0xFF
        if key == ord("q") or key == 27:
            break

    cv2.destroyAllWindows()

# Bonne pratique : ajouter cet idiome
# dans nos scripts
if __name__ == "__main__":
    main()
```

On a vu la dernière fois qu'en modifiant une simple ligne, on peut lire soit une vidéo sur notre ordinateur, soit utiliser directement la webcam :

```python
# Lire une vidéo :
cap = cv2.VideoCapture(video_path)
# Lire la webcam :
cap = cv2.VideoCapture(0)
```

1.2 Ajouter des fonctions externes

On va maintenant ajouter des modèles d'apprentissage profond pré-entraînés, que vous pouvez télécharger ici (`https://drive.google.com/drive/folders/1z2JPdyjzKaJz0SWsFOwZkL8-60u12jn_?usp=sharing`). Décompressez le dossier à la racine du projet. L'arborescence de votre projet doit ressembler à cela :

- main.py
- FPSCounter.py
- Parsers.py
- SkeletonTracker.py
- models/
  - face/
  - haarcascades/
  - pose/
  - yolo/
  - ...

Vérifiez une seconde fois si la structure de votre projet est bonne.

Si vous fouillez le dossier, vous retrouverez plusieurs fichiers, chacun comprenant des classes qui seront utiles durant ce TP.

Vous avez remarqué ces `import` en haut du code source à chaque fois? C'est un moyen de dire qu'on va utiliser des fonctions qui sont écrites dans d'autres fichiers. On a déjà beaucoup utilisé les fonctions de `cv2` (OpenCV) et `numpy` jusque là. Ce sont des "packages" installées quelque part sur notre ordinateur.

On va inclure les fonctions du présentes dans le projet dans notre programme principal en ajoutant au début de notre main.py :

```
from FPSCounter import FPSCounter
import Parsers
from Parsers import parseYoloResults, getSkeleton
```

Lancez votre programme avec `python3 main.py`. Pas d'erreur? Parfait! Sinon, appelez votre enseignant.

2 Afficher un FPS

Le "FPS" (Frame Par Seconde) est un point intéressant à garder à l'œil quand on utilise du Deep Learning (réseaux de neurones profonds). En effet, on arrive vite à un point où notre application n'est plus du tout interactif si on cherche trop de précision dans nos calculs. Le vrai travail d'un ingénieur, c'est de trouver un équilibre entre temps de calculs et précision des résultats.

On va alors ajouter un compteur de FPS sur notre application. En dehors de notre boucle principale (ce qu'on appelle la "partie d'initialisation"), on va initialiser notre compteur :

```
fps = FPSCounter()
```

Maintenant, dans la boucle principale, on lui dit de se mettre à jour à chaque frame.

```
fps.update()
```

Et enfin, on va demander d'afficher son résultat sur l'image qui va être affichée à l'écran :

```
img = fps.display(img)
```

Et voilà, aussi simplement que ça, on a un compteur de FPS qui doit s'afficher en haut à gauche de votre écran. Le code n'est pas si compliqué que ça, vous pouvez aller l'observer dans le fichier "FPSCounter.py".

Questions :

- Quel est votre FPS ?
- Quel est *approximativement* le FPS à conserver pour avoir une application fluide ?
- Remplacez `EPSILON = 100`, puis `EPSILON = 1`. Qu'est-ce qu'on va garder pour la suite ?

3 Initiation au Deep Learning

Parfait, on vient de voir qu'on peut cacher des morceaux de codes assez long dans des fichiers externes, comme ça notre code principal (main.py) reste clair et lisible!

3.1 Détecter la posture d'une personne

Parmi les cas d'application du traitement d'images/vidéos, l'analyse de posture est un point souvent étudié. Dans ce cas, on cherche à extraire le "squelette" d'une personne (pas au rayon-X, c'est pas encore possible). Vous voulez savoir si une personne lève la main? Étudier comment une personne marche (recherche de maladie), ou le mouvement d'un athlète (recherche de performances) ? Est-ce qu'une personne âgée tombe (recherche de détresses vitales) ? Dans tous ces cas, on cherche à trouver le squelette des gens.

L'état de l'art en terme d'estimation de squelettes, c'est l'algorithme OpenPose. Dans le dossier téléchargé, vous avez un chemin "models/pose/body_25" avec les fichiers "pose_deploy.protoxt" (qui décrit comment doivent s'enchaîner les convolutions) et "pose_iter_584000.caffemodel" (qui précise les coefficients pour chaque matrice de convolution). Vérifiez que ces fichiers existent.

Voyons ensemble comment on peut le mettre en place.

Premièrement, initialisons le réseau de neurones avec OpenCV dans la partie d'initialisation de notre code :

```
fichier_prototxt = ???
fichier_coffemodel = ???
pose_network = cv2.dnn.readNetFromCaffe(fichier_prototxt, fichier_coffemodel)
```

Maintenant, OpenCV sait exactement quelles sont les opérations qui sont demandées par cet algorithme (qu'on appelle maintenant un "modèle"), et on a (presque) fini notre travail !

Généralement, les modèles demandent à traiter les images dans un format spécial : la valeur des pixels est comprise entre 0 et 1, et on les stocke dans des "conteneurs" (ou "blobs" ou "tensors" ou "batch"). Mais tout est déjà prévu ! Dans notre boucle principale, on va modifier notre image avant de la donner au réseau :

```
taille_reseau = (256, 256)
# On "normalise" les images entre 0 et 1
inpBlob = cv2.dnn.blobFromImage(img, 1.0 / 255, taille_reseau, (127.5, 127.5, 127.5), crop=False)
```

Maintenant, dans la boucle principale, il n'y a plus qu'à le donner au réseau, et lui demander de retourner un résultat :

```
pose_network.setInput(inpBlob)
resultat = pose_network.forward()
print(resultat.shape)
```

Questions :

- Quelle est la structure ("shape") de la sortie ?

Ok, j'ai menti, le résultat n'est pas très compréhensible... On va le passer dans un "parseur", une fonction qui permet de mieux comprendre le résultat.

```
articulations = getSkeleton(resultat)
print(articulations)
```

Maintenant, si une personne est détectée, cela devrait afficher une liste de coordonnées, qui correspond aux articulations de la personne en vidéo.

Au lieu d'afficher cela dans le terminal, on va afficher à l'écran ces articulations sous forme de petit disques à l'écran :

```
couleur = (0, 255, 255)
for iArticulation, articulation in enumerate(articulations):
    if not np.isnan(articulation).any():
        position = np.array([articulation[1] * img.shape[1], articulation[0] * img.shape[0]])
        nom = Parsers.POSE_PARTS[iArticulation]
        cv2.circle(img, position.astype(int), 3,
                                couleur, -1)
        cv2.putText(img, f"{nom}" position.astype(int), cv2.FONT_HERSHEY_SIMPLEX, 1, couleur, 2)
```

Vous pouvez aussi ajouter un peu de code pour afficher les liaisons entre ces articulations :

```
for pair in Parsers.POSE_PAIRS:
    artA, artB = articulations[pair[0]], articulations[pair[1]]
    if not np.isnan(artA).any() and not np.isnan(artB).any():
        posA = np.array([artA[1] * img.shape[1], artA[0] * img.shape[0]])
        posB = np.array([artB[1] * img.shape[1], artB[0] * img.shape[0]])
        cv2.line(output, posA.astype(int), posB.astype(int), couleur, 2)
```

Questions :

- Est-ce que le réseau détecte des personnes sur les différentes vidéos à disposition ?
- Vous détecte-t-il sur votre webcam ?
- Que ce passe-t-il quand deux personnes sont présentes sur la vidéo ?
- Quel est le FPS ? Faites varier la variable "taille_reseau". Quelle est son influence sur le FPS et sur le squelette? (En informatique, on aime utiliser des dimensions en puissance de 2 [64, 128, 256, 512, 1024, ...])

Exercices :

- Créez un fichier "utilitaire_Pose.py".
- Dans ce fichier, créez une fonction "initialiser_reseau()" qui prend en argument les fichiers ".prototxt" et ".coffemodel", et qui retourne l'objet "network".
- De même, créez la fonction "traiter_image()" qui prend en argument le "network" ainsi qu'une image et qui retourne la liste des articulations détectées.
- Enfin, créez la fonction "afficher_resultats()" qui prend en argument une liste de coordonnées des articulations détectées et une image, et qui ajoute des points aux articulations et dessine le squelette. La fonction retourne cette image modifiée.
- Dans le code principal (main.py), importez votre nouveau fichier `import utilitaire_Pose` et réutilisez vos fonctions

- `reseau = utilitaire_Pose.initialiser_reseau(...)`
- `resultat_detection = utilitaire_Pose.traiter_image(...)`
- `frame = utilitaire_Pose.afficher_resultats(...)`

3.2 Détection d'objets dans la scène

Vous n'avez peut-être pas beaucoup d'objets autour de vous, alors téléchargez quelques vidéos sur le Moodle. Vous adapterez le lecteur vidéo pour correspondre aux fichiers vidéo que vous souhaiterez utiliser :

```
video_path = ???
```

L'état de l'art en terme de détection d'objets, c'est l'algorithme YOLO (You Only Look Once). La version actuelle est le YOLO v11, mais on va se contenter de la v4 (parce qu'on est des dinosaures, bien sûr). Dans le dossier téléchargé, vous avez un chemin "models/yolo/" avec les fichiers "yolov4-tiny.cfg" (qui décrit comment doivent s'enchaîner les convolutions), "yolov4-tiny.weights" (qui précise les coefficients pour chaque matrice de convolution) et "coco.names" (qui est un fichier texte qui précise tous les objets que notre réseau doit pouvoir reconnaître). Vérifiez que ces fichiers existent.

Voyons ensemble comment on peut le mettre en place.

(!! Vous allez voir que c'est quasiment identique à l'exercice précédent !!)

Initialisons le réseau de neurones avec OpenCV dans la partie d'initialisation de notre code :

```
fichier_cfg = ???
fichier_weights = ???
fichier_coco = ???

yolo_classes = None
with open(fichier_coco, 'rt') as f:
    yolo_classes = f.read().rstrip('\n').split('\n')

yolo_network = cv2.dnn.readNetFromDarknet(fichier_cfg, fichier_weights)
if yolo_network.empty():
    print("Erreur de chargement de YOLO")
else:
    print("YOLO est chargé")
```

Maintenant, dans la boucle principale, il n'y a plus qu'à le donner au réseau, et lui demander de retourner un résultat :

```
taille_reseau = (256, 256)
# On "normalise" les images entre 0 et 1
inpBlob = cv2.dnn.blobFromImage(img, 1.0 / 255, taille_reseau, (127.5, 127.5, 127.5), crop=False)
```

```
yolo_network.setInput(inpBlob)
resultat = yolo_network.forward()
print(resultat)
```

Utilisez la fonction pour lire un peu plus clairement le résultat :

```
objets_detectes = parseYoloResults(image, resultats)
print(objets_detectes)
```

Maintenant, si un objet est detecté, cela devrait afficher une liste de dictionnaires avec, pour chacun, un "id" (l'indice de l'objet detecté dans la liste "coco.name"), un "confidence" (la confiance du réseau) et une "box" (les coordonnées de la boite qui englobe l'objet).

Au lieu d'afficher cela dans le terminal, on va l'afficher à l'écran :

```
for detection in objets_detectes:
    box = detection["box"]
    cv2.rectangle(img, [box[0], box[1]], [box[0] + box[2], box[1] + box[3]], (0, 0, 255), 2)
    cv2.putText(img,
            f"{yolo_classes[detection['id']]} ({int(detection['confidence'] * 100)}%)",
            (box[0], box[1]), cv2.FONT_HERSHEY_SIMPLEX, 1,
            (0, 0, 255), 2)
```

Questions :

- Est-ce que le réseau détecte des objets sur les différentes vidéos à disposition ?
- Y a-t-il des objets détectés sur la webcam ?
- Quel est le FPS ?

Exercices :

- Créez un fichier "utilitaire_YOLO.py".
- Dans ce fichier, créez une fonction "initialiser_reseau()" qui prend en argument les fichiers ".cfg" et ".weights", et qui retourne l'objet "network".
- De même, créez la fonction "traiter_image()" qui prend en argument le "network" ainsi qu'une image et qui retourne la liste des objets détectés.
- Enfin, créez la fonction "afficher_resultats()" qui prend en argument une liste d'objets détectés et une image, et qui ajoute un rectangle autour de chaque objets, avec le texte qui précise le nom de l'objet. La fonction retourne cette image modifiée.

- Dans le code principal (main.py), importez votre nouveau fichier `import utilitaire_YOLO` et réutilisez vos fonctions
  - `reseau = utilitaire_YOLO.initialiser_reseau(...)`
  - `resultat_detection = utilitaire_YOLO.traiter_image(...)`
  - `frame = utilitaire_YOLO.afficher_resultats(...)`

Bravo, vous pouvez maintenant connaître quels sont les objets présents dans une vidéo !

Travail maison : Fouillez sur internet quelles sont les capacités des versions récentes de YOLO. C'est pas bien compliqué à installer, il y a beaucoup de ressources, et les capacités sont dingues !

3.3 Créer des classes pour nos détections

Prenons tout de suite des bonnes habitudes, il faut penser Programmation Orienté Objet (POO). On va alors modifier nos fichiers "utilitaire_YOLO.py" et "utilitaire_Pose.py" pour y inclure des classes (et donc utiliser des "objets").

Commençons par "utilitaire_YOLO.py" : Créez une classe "ObjectDetected" qui va contenir toutes les informations de chaque objet :

```
class ObjectDetected:
    def __init__(self, idClasse, label, box, confidence):
        self.idClasse = idClasse
        self.label = label
        self.box = box
        self.confidence = confidence

    def afficherTerminal(self):
        print(f"{self.label} detecté à {int(self.confidence * 100)}% aux coordonnées {self.box}")
```

Et on peut maintenant modifier les fonctions créées précédemment pour utiliser des objets ObjectDetected : "utilitaire_YOLO.initialiser_reseau()", "utilitaire_YOLO.traiter_image()" et "utilitaire_YOLO.afficher_resultats()"

De même, on va modifier "utilitaire_Pose.py" pour créer une classe "Person" qui va contenir le squelette d'une personne :

```
class Person:
    def __init__(self, skeleton):
        self.skeleton = skeleton
```

Et identiquement, on va modifier les fonctions créées précédemment : "utilitaire_Pose.initialiser_reseau()", "utilitaire_Pose.traiter_image()" et "utilitaire_Pose.afficher_resultats()"

Normalement, le code principal doit pouvoir être exécuté sans modifier "main.py" !

4 Utiliser un wrapper

Un wrapper (ou "surcouche") est un morceau de code qui, comme on l'a fait jusque là, doit simplifier l'utilisation de méthodes complexes. Il se trouve que nous avons développé un wrapper qui est probablement plus complet et plus optimisé que le votre, donc nous vous proposons de l'utiliser à la place.

Pour ce faire, importez le fichier "SkeletonTracker.py" en ajoutant en début de votre code "main.py" :

```
from SkeletonTracker import SkeletonTracker
```

Le wrapper est contenu dans la classe "SkeletonTracking", donc on va tout de suite créer un objet dans la partie d'initialisation :

```
tracking = SkeletonTracker(use_yolo=True, use_body=True, max_bodies = 1)
```

Maintenant on va faire nos traitements dans la boucle principale :

```
tracking.update(img)
```

Affichez le squelette des personnes et objets présents dans la scène :

```
for person in tracking.persons:
    img = person.displaySkeleton(img)
```

Il est possible de connaître la position de articulation d'une personne avec la fonction :

```
if len(tracking.persons) > 0:
    if person.articulationVisible("Nose"):
        position = (person.getArticulation("Nose") * np.array([600, 800]))
        x = int(position[1])
        y = int(position[0])
        cv2.circle(img, (x, y), 3, couleur, -1)
```

(La liste des articulations qu'il est possible de détecter est stockée dans Parsers.POSE_PARTS) ou la position d'un objet avec

```python
for objet_detecte in tracking.objects_detected:
    position = objet_detecte.center()
    label = objet_detecte.label
    cv2.circle(img, position.astype(int), 3, couleur, -1)
    cv2.putText(img, f"{label}", position.astype(int), cv2.FONT_HERSHEY_SIMPLEX, 1, couleur, 2)
```

Exercices :

- Affichez en bas à droite de l'écran "X personnes présentes" avec "X" le nombre de personnes détectées.
- Affichez un cercle rouge centré entre vos deux poignets ("RWrist" et "LWrist") et dont le diamètre est la distance entre vos mains.
- Créez une classe "CompteRebours" en s'inspirant du compteur de FPS. Commencez avec 10.0 secondes. Mesurez le temps écoulé entre deux appels à la fonction "update()" et soustrayez cette valeur au compte à rebours. La fonction "display()" doit afficher le temps restant en bas à droite de votre écran. Ajoutez une fonction "pause()", "reprendre()" et "reset()".
- Si la personne se penche à droite, mettez le compte à rebours en pause. Si les deux mains de la personne sont proches, remettez le compte à rebours à 10.0 secondes.