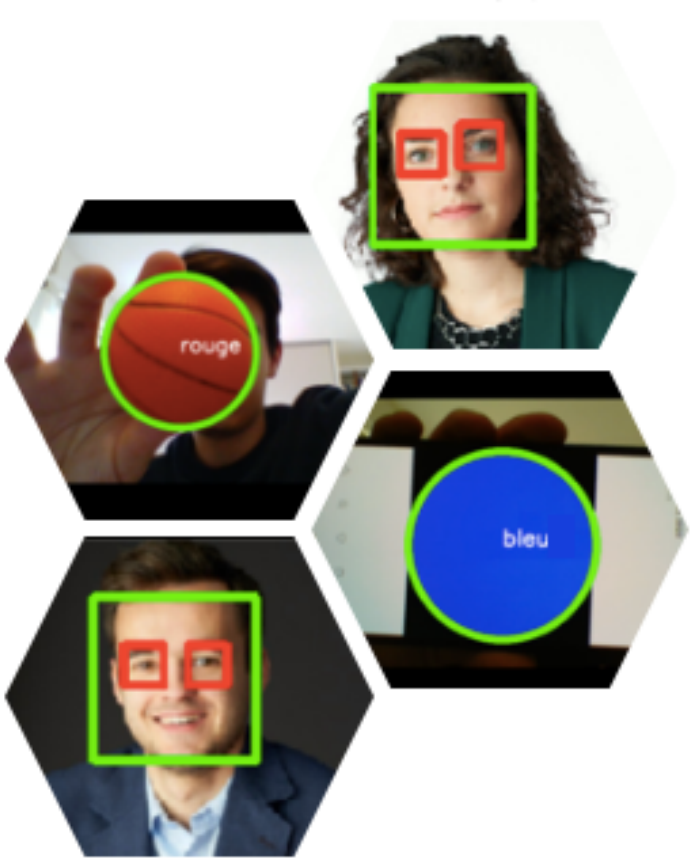
Guillaume BARTHE
Rushuang YANG
Clotilde VIÉ
Stefano ZANGIACOMI

Rapport technique



07/05/2020
Projet Convoyeur

SOLEC
Développement de Solutions Electroniques

Sommaire

Introduction

Dans ce rapport technique, nous détaillons la solution que nous apportons à un appel à projet lancé par l'entreprise SOLEC. Pour travailler sur ce projet, nous sommes une équipe de 4 personnes constituée de Guillaume BARTHE, Clotilde VIÉ, Rushuang YANG et Stefano ZANGIACOMI.

Au cours de ce projet, nous avons dû faire face à différentes contraintes, la principale étant le COVID-19. Cette pandémie mondiale nous aura contraint à travailler depuis chez nous. Par conséquent nous n'avions pas accès au matériel adapté pour réaliser le projet initial présenté. Néanmoins, souhaitant quand même être en mesure de présenter un projet fini et fonctionnel à SOLEC, nous avons modifier le projet initial afin de créer un projet adapté à la situation.

Au cours de ce rapport, nous vous présenterons dans un premier temps le projet initial ainsi que l'analyse technique que nous en avons fait. Nous consacrerons notre 2nd partie au projet actuel, à son implémentation technique, aux résultats que nous avons obtenu et à ses possibilités d'amélioration.

Nous remercions la société SOLEC d'avoir maintenu leur appel à projets et de nous avoir laissé la liberté de nous adapter de manière autonome.

I- Projet initial « convoyeur »

1. Description

L'objectif de ce projet initial était d'automatiser une chaine de tri. Plus précisément, il s'agissait d'automatiser le tri de pièces basé sur leur forme et leur couleur. Dans ce système, il y aurait eu un convoyeur entraîné par un moteur pas-à-pas, un interrupteur photoélectrique ainsi qu'un algorithme de détection de couleurs et de formes

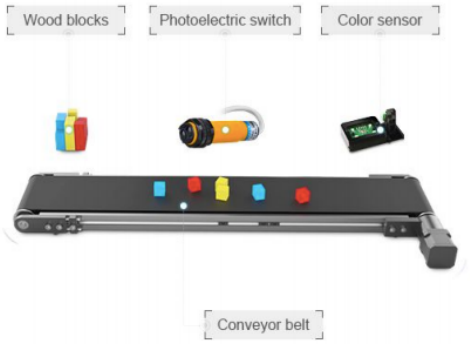


Figure 1 : schéma du projet « Convoyeur »

Ce projet se scinde en deux étapes :

- Dans un premier temps, une solution capable de détecter des cubes de 4 couleurs différentes et de les trier dans différents conteneurs.
- Dans un second temps, une solution capable de détecter une gamme plus large de couleur ainsi que des formes prédéterminées.

Le cahier des charges pour ce projet est donné ci-dessous :

- **Rapidité** : analyse de 10 pièces par minutes au minimum.
- **Détection** : différenciation des 4 couleurs suivantes : rouge, vert, jaune et bleu. La quantité de pièces triées et le temps moyen par pièce doivent pouvoir être affichés. Enfin, un nouveau capteur pourra être développé pour augmenter la quantité de couleurs détectables.
- **Fiabilité** : une erreur d'une pièce sur 1000 est tolérée sur la détection des couleurs de base.
- **Ergonomie** : une interface Humain-Machine, permettant de transmettre la couleur (ou forme) des pièces à trier, pourra être développée. Elle doit pouvoir être utilisée sans formation préalable.

2. Analyse technique

Après avoir étudié le principe du convoyeur, on détaille ci-dessous la solution technique développé par notre équipe.

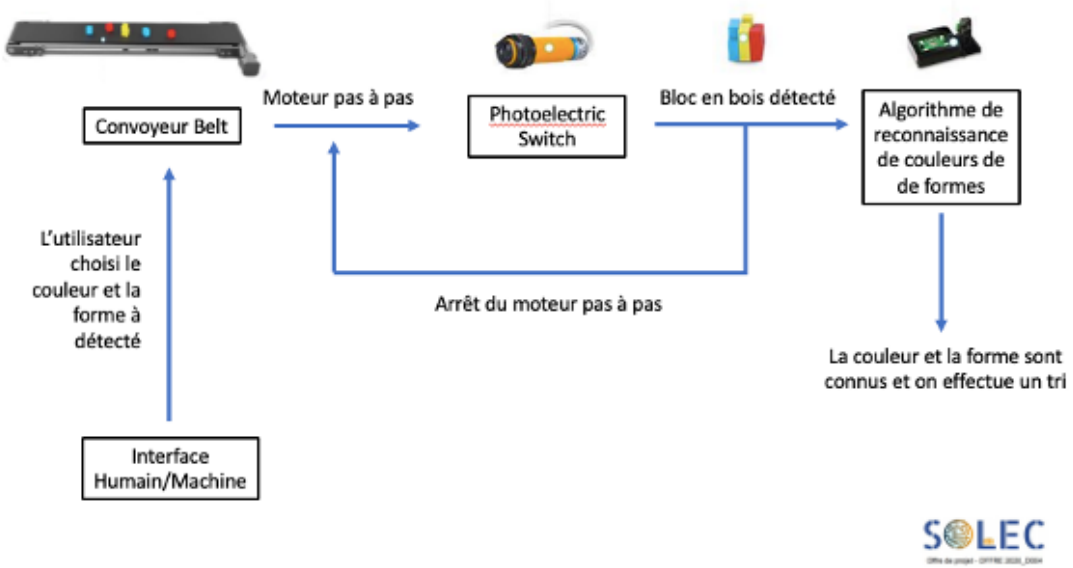


Figure 2 : solution technique : schéma-bloc du projet « Convoyeur »

Détails de la solution technique :

- **Interface humain/machine** : cette interface permettra à l'humain de rentrer la couleur et la forme qu'il souhaite détecter. Implémentation en Python avec la bibliothèque Tkinter.
- **Convoyeur Belt** : une fois la commande donnée par le biais de l'interface humain/machine, le convoyeur se met en marche grâce au moteur pas à pas. A la détection d'un objet (avec l'interrupteur photoélectrique), il s'arrête. Il repart lorsque la couleur et la forme de l'objet sont déterminés. Implémentation en Python et utilisation d'une carte Rasberry Pi pour l'électronique embarquée.
- **Interrupteur photoélectrique** : capteur permettant de détecter le passage d'un objet. Implémentation en Python et utilisation d'une carte Rasberry Pi pour l'électronique embarquée.
- **Reconnaissance de couleurs et de formes** : Algorithme de reconnaissance de couleurs et de formes. Implémentation en Python avec la bibliothèque OpenCv pour le traitement d'images.

3. Covid-19 : adaptation du projet

L'apparition du Covid-19 et le confinement qui a suivi nous a mis dans une situation qui ne nous permettait pas d'avoir accès au matériel nécessaire pour construire ce projet. En effet, nous n'étions plus en mesure d'avoir accès au convoyeur, à l'interrupteur photoélectrique, aux cartes Rasberry Pi, et surtout nous n'étions pas en mesure de tester nos algorithmes sur l'ensemble de ces objets. Cette dernière contrainte se révèle être très contraignante car sans tests concrets, il est très difficile de détecter des erreurs et d'être en mesure de proposer un projet fonctionnel.

Nous avons donc décidé de faire évoluer notre projet en gardant le maximum de caractéristiques du projet initial. L'algorithme de reconnaissance des couleurs et des formes nous a semblé intéressant, c'est la raison pour laquelle nous nous sommes basés sur ce dernier pour construire ce nouveau projet.
Nous nous sommes tournés vers une solution purement informatique : nous avons décidé de mettre en place un algorithme de détection de formes et de couleurs qui fonctionnerait sur un flux vidéo récupéré par la webcam de notre ordinateur.

Dans ce nouveau projet, vous retrouverez comme dans le projet initial, un algorithme de reconnaissance de couleur et forme ainsi qu'une interface humain/machine. Cependant, nous avons abandonné la partie « convoyeur » avec l'implémentation du moteur pas à pas. En contrepartie, nous avons souhaité aller plus loin dans nos algorithmes de détection : c'est ainsi que nous avons implémenté, en plus d'un premier algorithme de détection de forme et couleur, un algorithme de détection de visages et de yeux.

II- Nouveau projet « Big Brother »

1. Description

Comme expliqué ci-dessus, l'objectif de ce nouveau projet est de détecter la forme et la couleur d'objets se trouvant dans le champ de la webcam d'un ordinateur.

Nous nous sommes concentrés sur la détection d'une seule forme, les ronds, et ce pour des raisons de robustesse d'algorithme que nous détaillerons plus tard. L'algorithme qui a été développé détecte donc des ronds et est capable d'en donner la couleur. Nous avons également codé une interface graphique au travers de laquelle l'utilisateur peut lancer le programme et ajuster certains paramètres de l'algorithme.

Le principe de fonctionnement est le suivant : on place un objet rond devant la webcam de l'ordinateur ou alors on se place nous-même devant la webcam et on lance l'algorithme via l'interface graphique. En temps réel, l'algorithme détecte le rond et en donne la couleur ou bien détecte notre visage et nos yeux. Ci-dessous est présenté un schéma simplifié du principe de ce projet.

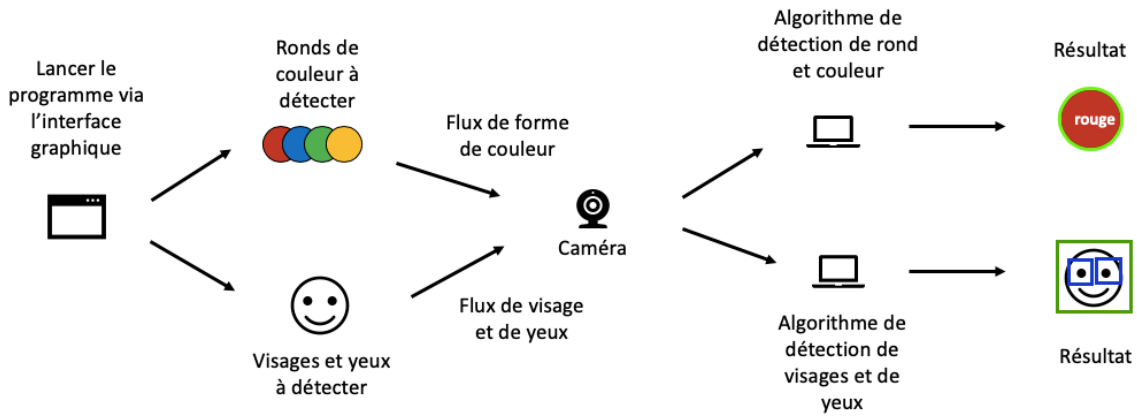


Figure 3 : schéma simplifié du projet « Big Brother »

2. Implémentation

a. Choix du langage Python

Nous avons décidé de réaliser notre système en langage Python en utilisant la bibliothèque OpenCv très utilisée en traitement d'images. (*Computer Vision)*

Nos algorithmes de détection peuvent se décomposer en 2 ou 3 grandes parties. La première consiste à obtenir le flux vidéo de la webcam de l'ordinateur. La deuxième, à détecter si un objet de la forme désirée se trouve dans l'image (rond ou visage). Et la troisième, une fois la position de l'objet détecté, à déterminer la couleur de cet objet (dans le cas de la détection de forme).

On commence par importer les bibliothèques suivantes qui seront utilisées tout au long du programme :

```
 8 import cv2
 9 import numpy as np
10 import tkinter as tkr
11 import urllib.request
12
```

Cv2 est la bibliothèque OpenCv servant au traitement d'image. Quant à la bibliothèque *numpy*, elle nous permettra d'effectuer des opérations mathématiques. *urlib.request* sera utile lors de la détection de visage et de yeux et *tkinter* permettra de coder l'interface graphique.

b. Flux vidéo

Afin d'obtenir le flux vidéo, il faut d'abord faire appel à l'objet webcam à l'aide de la ligne suivante :

```
11 cap = cv2.VideoCapture(0)
```

Afin d'avoir un flux vidéo en direct, il faut ensuite créer une boucle *while* infinie. Dans chaque itération de la boucle, on récupère l'image de la webcam à l'instant t.

```
15 while(1):
16
17     # Take each frame
18     _, frame = cap.read()  #C'est la variable frame qui contient l'image de la webcam
19
```

La variable *frame* contiendra ainsi l'image prise par la webcam. Néanmoins, bien que la boucle *while(1)* soit infinie, il est possible de l'arrêter en appuyant sur la touche Echap de l'ordinateur grâce aux lignes de codes suivantes :

```
90      k = cv2.waitKey(5) & 0xFF
91      if k == 27:
92          break
```

c. Détection de forme : le cercle

Une fois le flux vidéo récupéré, il est désormais possible d'y détecter une forme. Dans ce projet, nous nous sommes concentrés sur la détection de cercles dans l'image. Nous avons utilisé l'algorithme de Hough afin d'y parvenir. Afin d'utiliser cet algorithme, il faut tout d'abord transformer l'image en couleurs (BGR nativement sur OpenCv) en image en échelle de gris à l'aide de la fonction *cvtColor* de OpenCv. Ensuite, on applique un filtre moyenneur sur l'image afin d'augmenter le RSB (ratio signal sur bruit) et ainsi améliorer la détection des cercles.

```
20      frame_gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
21      frame_gray = cv2.medianBlur(frame_gray, 9)
```

Le paramètre *cv2.COLOR_BGR2GRAY* signifie que l'on souhaite passer de l'espace de couleurs BGR à l'espace en échelle de gris. On passe ainsi d'une image de profondeur 3 (BGR) à une image avec une seule valeur par pixel. Quant au filtre moyenneur *cv2.medianBlur*, le deuxième argument indique le diamètre en pixel du noyau du filtre. Cet argument doit absolument être un chiffre impair afin de pouvoir effectuer correctement la convolution sur l'image. Nous avons choisi un noyau de diamètre 9 pixels afin de réduire au maximum le bruit sur l'image mais sans trop détériorer celle-ci.

On peut ensuite appliquer la transformée de Hough afin de détecter les cercles sur l'image.

```
26      circles = cv2.HoughCircles(frame_gray,cv2.HOUGH_GRADIENT,1,100,
27                                 param1=400,param2=30,minRadius=100,maxRadius=250)
```

L'information sur les cercles détectés par l'algorithme de Hough est contenue dans la variable *circles*.

L'algorithme de Hough pour les cercles fonctionne de la manière suivante :

1. Une détection des bords présents dans l'image est obtenue par un filtre de Canny. La valeur du paramètre *param1* est la valeur du seuil (threshold) utilisé dans l'algorithme de Canny.

2. Une fois les pixels détectés comme « bord », l'algorithme va tracer des cercles compris entre *minRadius* et *maxRadius* de rayon (en pixels) pour chacun de ces pixels.
3. Pour chacun de ces cercles tracés, si le nombre de points passant par ce cercle est supérieur à *param2,* alors on considère que le cercle existe bel et bien.

La valeur optimale des paramètres dépend de nombreux facteurs comme la taille de l'image, la taille des cercles que l'on souhaite détecter ainsi que la résolution de l'image. Il est donc nécessaire d'effectuer des tests afin de trouver les paramètres optimaux.

L'information des cercles détectés étant contenu dans la variable *circles,* il est possible d'y accéder pour connaitre le centre et le rayon des cercles afin de les afficher sur l'image.

```
29    if circles is not None:
30        circles = np.uint16(np.around(circles))
31        for i in circles[0,:]:
32            x = i[0] #coordonnée x du centre du cercle
33            y = i[1]#coordonnée y du centre du cercle
34            # i[2] : rayon du cercle
35            # (0,255,0) : Coordonnées BGR de la couleur du cercle que l'on veut afficher
36            # Dessine le contour du cercle
37            cv2.circle(frame,(x,y),i[2],(0,255,0),10) # 10 : épaisseur du trait
38            # Dessine le centre du cercle
39            cv2.circle(frame,(x,y),2,(0,0,255),3)   # 3 : épaisseur du trait
40
```

Si des cercles sont détectés dans l'image, on va alors récupérer les informations en commençant par convertir les données en format *uint16.* Ensuite on va parcourir les cercles contenus dans la variable *circles* et on les affiche avec le style souhaité.

d. Détection de couleur

Après avoir détecté la présence de cercles dans l'image ainsi que leurs positions, on peut ensuite déterminer la couleur de ces cercles. Pour y parvenir, on a décidé de se placer dans l'espace de couleur L*a*b, ce que l'on peut faire facilement à l'aide de la fonction suivante :

```
23    frame_LAB = cv2.cvtColor(frame,cv2.COLOR_BGR2Lab)
```

Cet espace de couleur à l'avantage de pouvoir permettre une comparaison fiable entre deux couleurs. En effet, l'écart entre 2 couleurs tel qu'il est perçu par l'œil humain peut s'exprimer par la formule suivante :

$$\Delta_{couleurs} = \sqrt{\Delta_L{}^2 + \Delta_a{}^2 + \Delta_b{}^2}$$

Pour chaque cercle détecté précédemment, on récupère les coordonnées L*a*b du pixel du centre et on calcul les écarts de couleur pour chaque couleur définie préalablement au début de l'algorithme dans la variable *colors.* On a enregistré 9 couleurs mais il est tout à fait possible d'en ajouter bien plus.

```
colors = [[53, 80, 67], [87, -86, 83],[32, 79, -107], [75, 24, 79], [97.14, -21.56, 94.48],
          [29.7821, 58.9401, -36.4980],[0,0,0], [100.0000, 0.0053, -0.0104],[95.9489, -4.1890, 12.0399]]
```

On enregistre chaque écart calculé dans une liste nommée *ecarts_liste* puis on détermine l'index du minimum de cette liste afin de connaitre la couleur du cercle.

```
            #Color detection cercle

            pixel_centre = frame_LAB[x,y]


            L = pixel_centre[0] * 100/255   #Convertion 8bits par Open CV donc il faut revenir au vrai L*a*b
            a = pixel_centre[1] - 128
            b = pixel_centre[2] - 128

            ecarts_liste = []

            for color in colors:
                ecart = np.sqrt((L-color[0])**2 + (a-color[1])**2 + (b-color[2])**2)
                ecarts_liste.append(ecart)
            index_color = ecarts_liste.index(min(ecarts_liste))


            if index_color == 0:
                color_name = 'rouge'
            elif index_color == 1:
                color_name = 'vert'
            elif index_color == 2:
                color_name= 'bleu'
            elif index_color == 3:
                color_name= 'orange'
            elif index_color == 4:
                color_name= 'jaune'
            elif index_color == 5:
                color_name= 'violet'
            elif index_color == 6:
                color_name= 'noir'
            elif index_color == 7:
                color_name= 'blanc'
            elif index_color == 8:
                color_name= 'beige'

```

Ensuite on affiche directement le nom de la couleur sur l'image.

```
            # font
            font = cv2.FONT_HERSHEY_SIMPLEX
            # fontScale
            fontScale = 1
            thickness = 2

            cv2.putText(frame, color_name, (x,y), font,
                   fontScale, (255, 255, 255), thickness, cv2.LINE_AA)

            # (255, 255, 255) couleur du texte à afficher

```

e. IBM Watson

La partie détection de formes et couleurs étant terminée, cette sous-partie est un léger aparté permettant d'introduire notre algorithme de reconnaissance faciale. Elle fait suite à un MOOC que nous avons suivis mais que nous n'avons utilisé que partiellement. Cette partie est néanmoins intéressante pour que l'on comprenne notre démarche et que d'autre part, l'on puisse réexpliquer rapidement les bases du *Machine Learning* qui fait partie intégrante de nos algorithmes.

Avant d'entamer notre MOOC, nous n'avions jusque-là aucune notion dans la détection de visages. Après plusieurs heures de cours et de TD sur machine nous avons appris à utiliser des algorithmes déjà implémentés sur le web, notamment celui implémenté par IBM, intitulé IBM Watson.

C'est une intelligence artificielle qui a été entraînée par apprentissage automatique (*Machine Learning*) et qui est utilisée dans plusieurs domaines dont la reconnaissance fait partie.

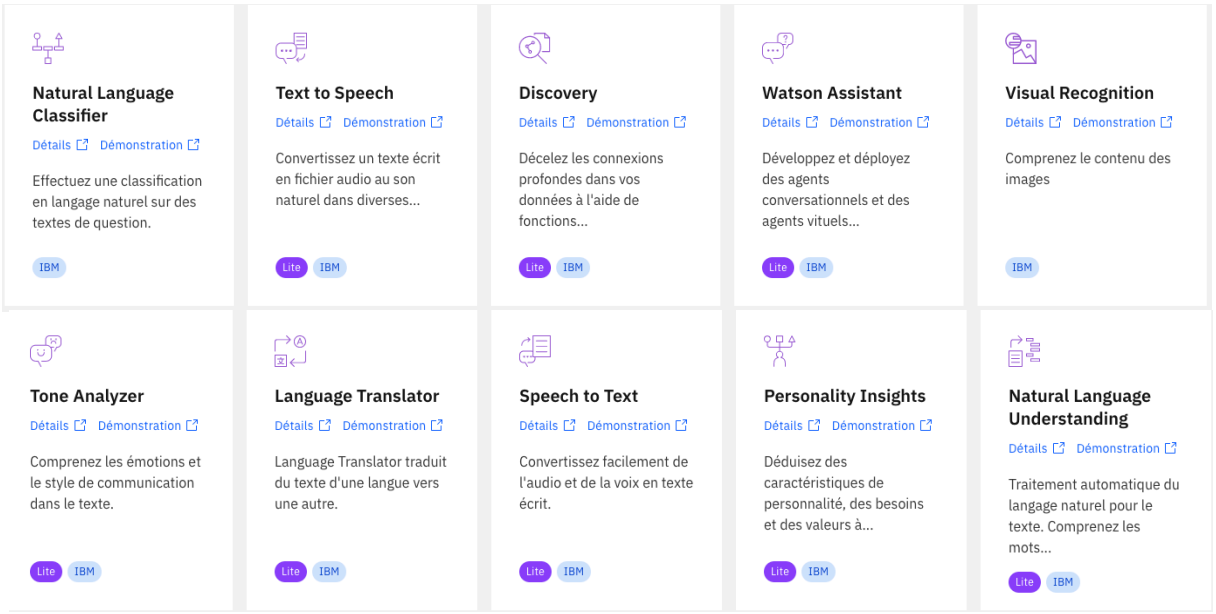


Figure 4 : exemple des services existant avec l'assistant Watson d'IBM.

Avec l'aide de Watson, nous avons appris à entrainer un algorithme à reconnaitre des images. En effet, l'efficacité d'une intelligence artificielle basée sur du Machine Learning dépend principalement de l'entrainement qu'elle a reçu.

Mais avant de vous expliquer comment nous avons procédé pour entraîner notre algorithme, voici quelques précisions sur son principe de fonctionnement.

Aujourd'hui, n'importe quel smartphone est capable de détecter un visage sur une photo. On a pourtant longtemps été incapable de le faire. La principale difficulté réside dans la façon dont les pixels qui constituent l'image sont perçus par l'ordinateur. En effet cette dernière est extrêmement différente de la façon dont un humain traite une image. C'est la raison pour laquelle il était impossible pour un humain de définir des conditions permettant à l'ordinateur de décrire ce qu'il observait sur l'image. L'idée du Machine Learning et de l'algorithme de reconnaissance faciale est donc de présenter à la machine une collection d'images afin qu'il détecte lui-même les caractéristiques nécessaires à l'identification d'un visage et qu'il puisse l'appliquer sur les prochaines images qui lui seront fournies.

Pour entrainer notre algorithme, la première chose à faire a été de lui donner un jeu de données. Le jeu de données dans notre cas était constitué de plusieurs milliers d'images de visage rangées dans des classes. L'idée étant de faire comprendre à l'ordinateur que sur ces milliers de photos, il existe un unique point commun : la présence d'un visage.

La seconde étape de l'apprentissage est ce que l'on appelle l'apprentissage *négatif*. Elle consiste à définir une nouvelle classe d'images ne contenant pas de visages, permettant à l'ordinateur de détecter la présence ou l'absence d'un visage sur une future image.
La nécessité de cette étape est facilement illustrable : imaginez que vous entrainiez votre algorithme à repérer des fruits sur une image. Si vous ne lui montrez que des photos de fruits, la première fois qu'il rencontrera une image contenant des légumes il aura tendance à vous dire qu'il détecte des fruits.

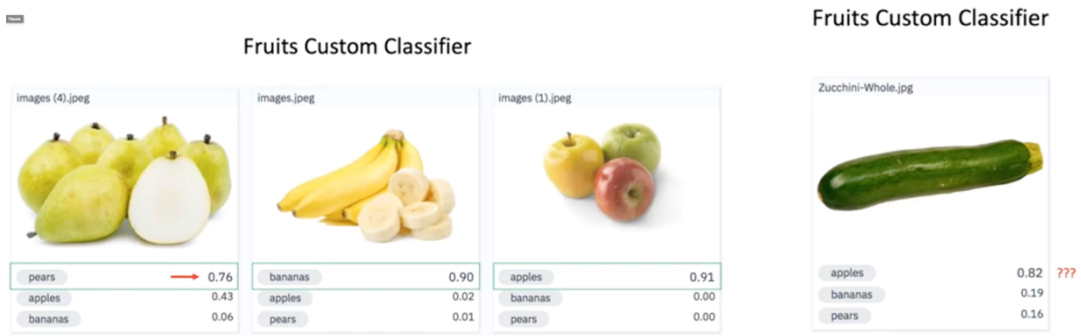


Figure 5 : à gauche, résultat efficace pour une classification de fruits avec Watson et à droite, résultat aberrant si l'on omet l'étape d'apprentissage négatif

Une fois la phase d'apprentissage réalisée, il nous a fallu choisir le programme adapté pour l'exécution de l'algorithme. Comme expliqué précédemment, notre objectif était de réaliser la reconnaissance faciale au travers du flux vidéo de la webcam d'un ordinateur. Nous avons dans un premier temps entrainé notre algorithme directement sur le site d'IBM Watson. Puis nous avons implémenté un programme Python permettant de

récupérer le flux vidéo de la caméra et d'appliquer notre programme pré-entraîné sur ce flux vidéo.

Pour ce faire, nous avons préalablement dû récupérer notre clé API sur le site d'IBM et ainsi permettre à notre code de fonctionner. Cette clé permet à l'ordinateur de se connecter à notre compte IBM et d'exécuter les manipulations demandées en se servant des données présentes sur le site IBM.

Cependant l'algorithme utilisé par Watson pour la détection faciale n'étant pas disponible localement mais uniquement au travers du site internet d'IBM, il est impossible pour le logiciel de traiter des images introuvables sur internet. Ce dernier ne peut donc pas traiter le flux d'images de notre webcam. C'est pourquoi nous avons décidé d'abandonner l'utilisation d'IBM Watson et de poursuivre le MOOC pour apprendre à faire de la détection d'images locales.

f. Détection de visage et de yeux

Nous avons utilisé pour cela des *cascades de Haar* (*Haarcascade* en anglais). C'est l'une des méthodes de détection d'objet les plus connues, créée en 2001. L'idée est simple : il s'agit de parcourir l'image avec une fenêtre glissante pour déterminer si un visage y est présent. Le terme de cascade fait référence à l'application successive de classificateurs simples. Le premier permet de rejeter rapidement l'image si l'objet cherché ne s'y trouve pas. En revanche, s'il est possible qu'il s'y trouve on utilise alors le deuxième classificateur plus précis et ainsi de suite. L'idée est de traiter les images avec le moins de calculs possibles.

De la même manière qu'avec Watson, nous avons décidé d'utiliser les Haarcascade présents sur Github plutôt que d'essayer de les coder intégralement (ce qui n'aurait pas été à notre portée).

La première étape du code est d'importer des bibliothèques utiles. Ainsi, on importe comme pour la détection des formes et des couleurs la bibliothèque OpenCv avec *cv2* pour le traitement d'image et *urlib.request* afin de pouvoir importer les cascades utiles. On importe ensuite les *Haarcascades* utilisés pour la détection des visages/yeux.

```
import cv2              #Importation de la bibliothèque OpenCv
import urllib.request

facecascade_url='https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_frontalface_default.xml'
face_name= "haarcascade_frontalface_default.xml"
urllib.request.urlretrieve('https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_frontalface_default.xml', 'haarcascade_frontalface_default.xml')
facedetector=cv2.CascadeClassifier(face_name)

haarcascade_url='https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_eye.xml'  #on va chercher sur github les codes cascade qui permettent de de
eye_name='haarcascade_eye.xml'
urllib.request.urlretrieve(haarcascade_url,eye_name)
eyedetector=cv2.CascadeClassifier(eye_name)

```

Comme pour l'algorithme précédent, on crée une boucle *while(1)* infinie pour récupérer le flux vidéo de la webcam de l'ordinateur et on détecte sur ce flux s'il y a des visages.

```
while(1):

    _, frame = cap.read()  #C'est la variable frame qui contient l'image de la webcam
    faces_list = facedetector.detectMultiScale(frame)  ## Detect faces froum our image
```

De même, on peut arrêter l'algorithme à tout moment en cliquant sur la touche Echap.

```
    k = cv2.waitKey(5) & 0xFF  # permet d'arreter le programme d'une simple pressions sur escape
    if k == 27:
        break
```

Nous avons constaté que les résultats étaient satisfaisants lorsque que l'on se mettait face à la caméra mais dès lors que l'on se déplaçait légèrement du cas idéal, l'algorithme détectait de nombreux faux visages. C'est pour régler ce problème que nous avons décidé implémenter une détection supplémentaire : la détection des yeux.

Le principe de détection des yeux est identique à celui présenté précédemment, il se base simplement sur un *Haarcascade* différent. Nous avons donc implémenté cette détection des yeux puis ajouté au programme la règle suivante : si dans un visage détecté, aucun œil n'est détecté alors l'affichage du visage n'est pas voulu. Cela a permis d'améliorer grandement l'efficacité du programme.

```
    if faces_list is not None:
        for face in faces_list:  #On parcourt tout les visages trouvés
          (x,y,w,h)=face
          rect=frame[y:y+h,x:x+w]

          eyes_list=eyedetector.detectMultiScale(rect)
          compteur_yeux=0
          for eye in eyes_list:
            (ex,ey,ew,eh)=eye
            if compteur_yeux<2:
                cv2.rectangle(rect,(ex,ey),(ex+ew,ey+eh),(255,0,0),3)
                cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,0),4)      # on affiche le visage que s'il contient des yeux
            else:
                compteur_yeux=0
            compteur_yeux=compteur_yeux+1

    cv2.imshow('frame',frame)

    k = cv2.waitKey(5) & 0xFF  # permet d'arreter le programme d'une simple pressions sur escape
    if k == 27:
        break

cv2.destroyAllWindows()
cap.release()
```

Finalement, on affiche l'image simplement avec la commande :

```
    cv2.imshow('frame',frame)
```

g. Interface graphique

L'interface graphique doit permettre à l'utilisateur de lancer les deux programmes ci-dessus, tout en lui permettant de calibrer certains paramètres nécessaires au bon

fonctionnement de chacun des algorithmes. Afin de coder cette interface, on s'est basé sur un MOOC d'initiation à Tkinter.

L'interface finale se présente sous la forme suivante :

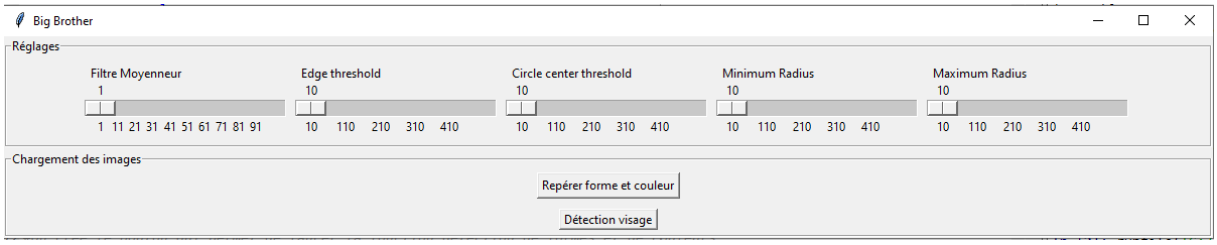


Figure 6 : interface graphique

La fenêtre ci-dessus s'ouvre au lancement du code Python. Elle est constituée de deux cadres :

- Un cadre de réglages des paramètres qui permettent le bon fonctionnement de l'algorithme de détection des cercles et des couleurs.
- Un cadre contenant deux boutons qui lancent séparément les deux programmes.

L'interface présente ainsi une structure simple mais efficace. L'utilisation est alors ergonomique : il suffit de choisir les paramètres adaptés, et un simple clic de souris lance alors la détection.

Après l'importation des bibliothèques utiles (II.a), le code se présente en 3 parties. On définit tout d'abord la fenêtre de travail et on lui attribue une dimension (ici 1000*200), ainsi qu'un titre (« Big Brother »). On définit ensuite deux sous-fenêtres, pour le cadre de réglage et de lancement.

```
13
14 #Définition de l'interface
15 fenetre = tkr.Tk() #On crée la fenêtre qui constitue l'interface
16 fenetre.geometry("1200x200") #Définition des dimensions
17 fenetre.title("Big Brother") #titre de la fenêtre
18
19 P=tkr.PanedWindow(fenetre, orient=tkr.VERTICAL) #Sous fenêtres pour le cadre de réglage et de lancement
20 P.pack(side=tkr.TOP, expand=tkr.Y, fill=tkr.BOTH)
21
```

On crée ensuite le premier cadre des réglages, et on le nomme « réglages ». Nous lui ajoutons 4 curseurs qui serviront à ajuster les paramètres de l'algorithme de détection de forme et couleur.

Lorsqu'on glisse un curseur et qu'on le règle à une certaine valeur, cette dernière est affectée à une variable (*val1* par exemple), qui est ensuite utilisée dans le programme de détection.

```
#Cadre de réglages
Reglage = tkr.LabelFrame(P, text="Réglages", padx=5, pady=5)
PReglage = tkr.PanedWindow(Reglage, orient=tkr.HORIZONTAL)


#Réglage filtre : curseur qui attribue la valeur donnée par l'utilisateur à la variable val1
frameSeuil=tkr.Frame(PReglage)
val1=tkr.IntVar()
tkr.Scale(frameSeuil, variable=val1,from_=1, to=99,orient=tkr.HORIZONTAL,tickinterval=10, length=200, label='Filtre Moyenneur').pack()
PReglage.add(frameSeuil)

```

On crée finalement le deuxième cadre qui contient les commandes permettant de lancer les programmes. Ils sont sous forme de boutons : en appuyant dessus, on peut soit lancer l'algorithme de formes ou l'algorithme pour les visages. Comme expliqué précédemment, pour quitter l'un des deux algorithmes, il suffit d'appuyer sur le bouton Echap.

```
#Cadre de chargement de l'image live de la webcam
Chargement = tkr.LabelFrame(P, text="Chargement des images") #attribue titre au cadre

#On crée le bouton qui permet de lancer la fonction détection de formes et de couleurs
boutonFile = tkr.Button(Chargement,text="Repérer forme et couleur", command=WebcamCouleur)
boutonFile.pack(pady=5)

#On crée le bouton qui permet de lancer la fonction détection de visages
boutonFile = tkr.Button(Chargement,text="Détection visage", command=DetectionVisage)
boutonFile.pack(pady=5)

P.add(Chargement)

fenetre.mainloop()
```

3. Résultats

a. Algorithme de détection de forme et couleur

Ci-dessous, voici des exemples de ce que l'on a pu obtenir avec l'algorithme implémenté.

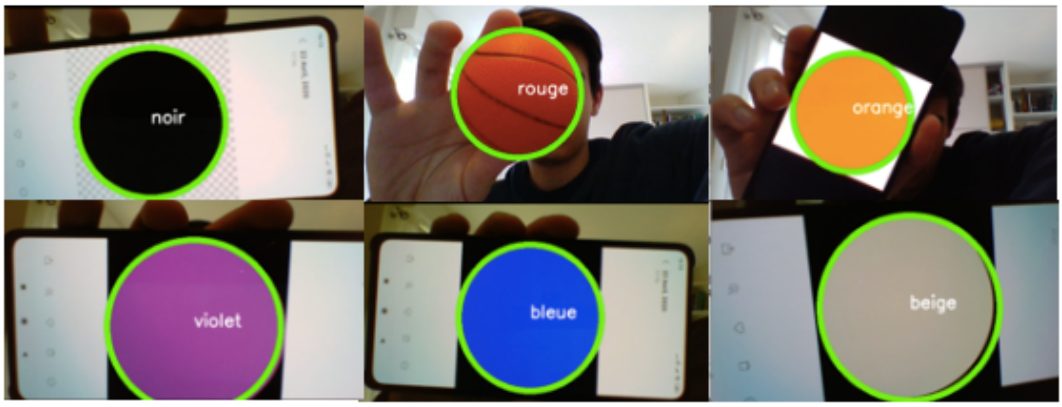


Figure 7 : résultats obtenus pour de la détection de rond de couleur

Globalement, les résultats de l'algorithme sont très satisfaisants. Il y a tout de même quelques points d'amélioration que nous évoquerons en fin de rapport.

b. Algorithme de détection de visage et yeux

Ci-dessous, voici les premiers résultats obtenus sans prise en compte des yeux.

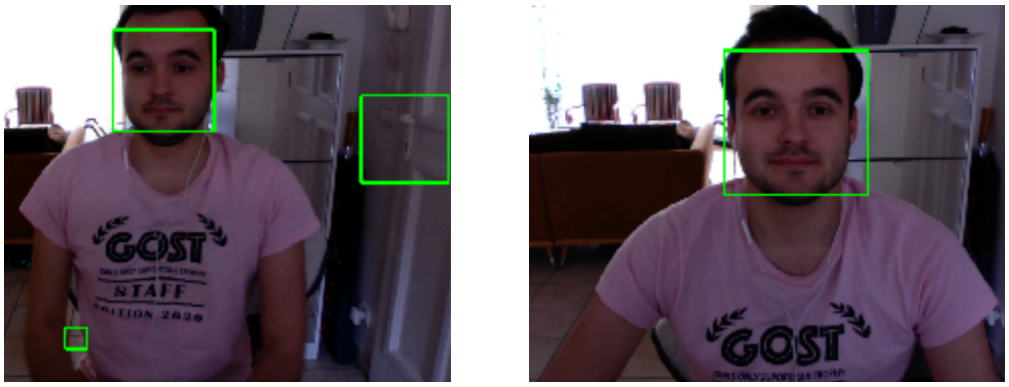


Figure 8 : à gauche, résultat aberrant, détection de multiples visages et à droite, résultat satisfaisant, détection d'un seul visage

Maintenant, voici les résultats, bien plus satisfaisants, obtenus lorsqu'on tient compte des yeux dans le visage. L'algorithme fonctionne sur un flux image de la webcam mais il fonctionne aussi très bien sur une image préexistante.

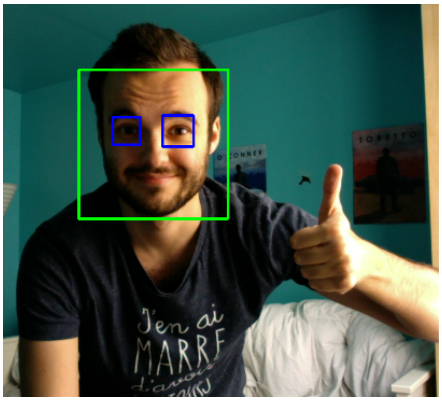


Figure 9 : résultat satisfaisant avec le flux de la webcam

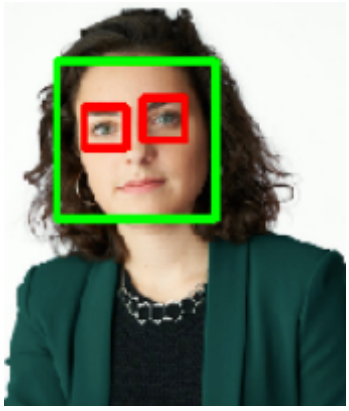


Figure 10 : résultat satisfaisant sur une image déjà existante

Néanmoins, on a aussi observé que l'algorithme fonctionnait parfois moins bien si la luminosité était mauvaise, si les yeux n'étaient pas distinctement reconnaissables ou si la personne était de profil.

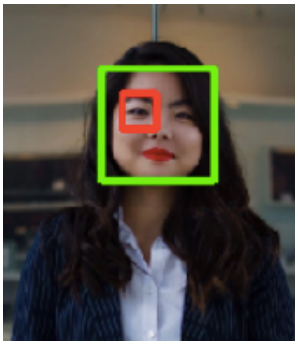
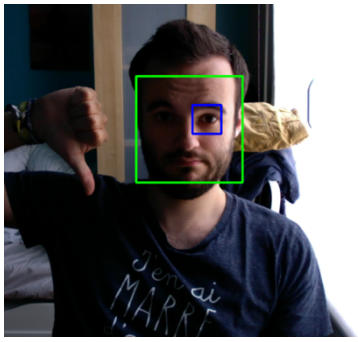
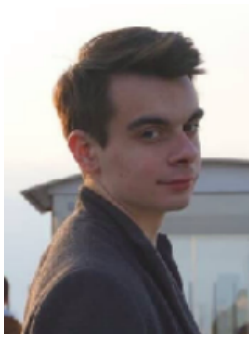




Figure 11 : résultats moins satisfaisants lors d'un mauvais éclairage, d'une mauvaise distinction des yeux et d'une personne de profil.

4. Pistes d'amélioration

Globalement, le projet fonctionne bien mais il y a bien sûr des pistes d'amélioration à explorer.

Premièrement, il serait intéressant d'implémenter un algorithme permettant la détection de différentes formes. Ainsi on pourrait proposer sur l'interface différentes formes à détecter telles que des triangles ou carrés et l'utilisateur choisirait alors lesquelles il souhaite détecter. Dans la continuité du développement de cet algorithme, on pourrait aussi proposer un choix de couleur plus vaste.

Au cours du développement de notre projet, nous avons souhaité intégrer l'image vidéo sur l'interface graphique mais cela n'a malheureusement pas fonctionné car cela s'est avéré assez complexe. De fait, ce serait l'un des axes sur lequel il serait possible de travailler.

En ce qui concerne la détection de couleur, il arrive que dans certains cas elle donne le mauvais résultat. Cela est notamment dû à la conversion BGR vers L*a*b. En effet, afin d'obtenir une conversion correcte il est nécessaire de connaître l'illuminant de la scène. Par conséquent, l'implémentation d'un algorithme de balance des blancs augmenterait la robustesse du programme.

Enfin, étant donné que l'algorithme de détection de visage et yeux a du mal à détecter correctement un visage dès que la personne est de profil, qu'il y a des problèmes de luminosité ou que l'on distingue mal les yeux, il serait intéressant de travailler là-dessus afin de rendre cet algorithme plus robuste.

Conclusion

En répondant à cet appel à projet, nous avons su évoluer avec les contraintes actuelles en proposant un projet adapté à la situation et qui réponde autant que possible au cahier des charges initiales

Nous avons pu mettre en avant nos capacités à coder des algorithmes de détection efficaces et relativement robustes ainsi qu'une interface graphique permettant à n'importe quel utilisateur de les utiliser simplement. Il ne fait pas de doute que nous pourrions dans des circonstances « normales » adapter relativement notre travail au projet initial.

Bibliographie commentée

1. edX CV0101EN: *Computer Vision Fundamentals with Watson and OpenCV* accessible au lien suivant https://courses.edx.org/courses/coursev1:IBM+CV0101EN+1T2020/course/

MOOC très bien fait sur la reconnaissance d'image. C'est un format de courtes vidéos qui s'enchaînent et de mini "labs" pour s'entrainer à coder, c'est accessible à des personnes ayant des bases de Python mais il n'y a pas besoin d'en connaitre beaucoup.
Il y a toute une première partie sur Watson IBM, l'algorithme d'intelligence artificielle créée par IBM, puis une seconde sur la reconnaissance d'image plus générale.

2. *An Introduction to Tkinter* accessible au lien suivant : http://effbot.org/tkinterbook/?fbclid=IwAR3XPkqoC9w7KsGDtGAUWSTIXOq4TBFQ91DYeMIxvaXY7aQqzElL6JbGv3A

MOOC très complet sur comment créer une interface graphique avec tkinter. Il y a beaucoup de fonctionnalités expliquées dessus (peut-être un peu trop). Bien adapté aux débutants (il y a un onglet introduction pas à pas qui permet de mettre en œuvre une première interface facile et efficace). Des exemples de code sont prêts à être copiés et ils sont bien commentés.
Bémol : écrit en anglais (ceux qui n'ont pas l'habitude pourraient avoir du mal avec les mots techniques).

3. *OpenCV Python Tutorial* accessible au lien suivant : https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html?fbclid=IwAR0DW8X5sNvkWoJBuU0UFpB7HONAqVp2pmee0A8BK6552L8zARNmXqGbhtE

Regroupement de tutoriels permettant d'apprendre les bases du codage sur Python avec OpenCV pour le traitement d'image.

4. Cours de colorimétrie d'Hervé Sauer

Ces cours nous auront été utile pour traiter correctement la partie colorimétrique de notre algorithme.