# Vision par ordinateur - Détection de visage et Machine Learning

Paigneau Hugo

***Abstract*— La détection des visages peut être considérée comme un cas spécifique de détection de classe d'objet. Dans la détection de classe d'objet, la tâche consiste à trouver les emplacements et les tailles de tous les objets d'une image qui appartiennent à une classe donnée.**

**Une approche fiable de la détection des visages basée sur l'algorithme génétique et la technique "eigen-face". Une autre approche est aussi celle vue en cours : la méthode Viola et Jones. Elle fait partie des toutes premières méthodes capables de détecter efficacement et en temps réel des objets dans une image.**

## I. INTRODUCTION

Ce projet fait partie de l'UV SY32 (Analyse et synthèse d'images). L'objectif est de mettre en pratique les méthodes vues en cours et en TD en créant un detecteur de visage from scratch. Nous avons à notre disposition une base de données de 1000 images labellisées contenant chacune un.des image.s. Nous utiliserons le language Python ainsi que les librairies de machine learning très connues pandas, numpy, Pillow, scikit-learn, scikit-image et OpenCV.

## II. GÉNÉRATION DU DATASET

Comme chaque problème à une solution, toute situation de machine learning nécessite un dataset. Cette étape est primordiale puisque notre modèle va être entraîné depuis ces données dans un premier temps. Le but est donc de générer un dataset qui sera utile à notre problème, qui correspondra le mieux aux problèmes posés. J'ai opté pour sélectionner des données exclusivement rectangulaires puisqu'un visage humain est par nature allongé. La taille retenue est de $(60, 45)$px, une taille qui m'a permis de manipuler les données assez efficacement en consommant du temps de calcul et de la RAM dans une moindre mesure.

### A. Positifs

Nous allons dans un premier temps s'attarder sur la génération d'images positives. Cette partie n'est pas particulièrement complexe puisque nous disposons d'un fichier ***label_train.txt*** qui nous donne pour chaque image, la position du ou des visages présents sur ces dernières. On obtient in fine 1284 visages.

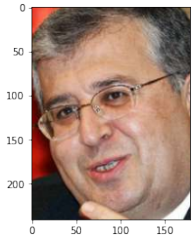


**Fig. 1:** Image positive généré à partir du dataset

La partie intéressante vient alors. En effectuant de la data augmentation, on peut empêcher notre modèle d'apprendre des schémas non pertinents, essentiellement en augmentant la performance globale. Deux options s'offrent à nous. La première consiste à effectuer toutes les transformations nécessaires au préalable, essentiellement en augmentant la taille de votre ensemble de données. L'autre option consiste à effectuer ces transformations sur un batch, juste avant de l'envoyer à notre modèle d'apprentissage machine. L'utilisation la première en vue de notre dataset assez réduit pour un tel sujet est une bonne idée. Cette première option est connue sous le nom de ***offline augmentation***. Elle est préférable pour les ensembles de données relativement petits, puisqu'on va augmenter la taille de l'ensemble de données d'un facteur égal au nombre de transformations effectuées. J'ai ainsi essayé diverses transformations :

- retourner toutes les images (facteur 2 d'augmentation)
- rotation de toutes les images d'un angle de 10 degré (facteur 2 à 4 d'augmentation).
- mise à l'échelle (facteur arbitraire selon les choix cette méthode semble bonne au première abord, mais nous verrons plus tard que ce n'est pas toujours le cas).

Il fallait en fait que les augmentations aient un sens pour notre application cible. Notre but est de détecter des visages sur une image. Ces visages ne seront jamais à l'envers. Il faut aussi pouvoir gérer la profondeur : c'était le but primaire de la mise à l'échelle, cependant, les résultats n'ont pas été fructueux; il a été plus judicieux d'effectuer un algorithme de ***Sliding Window*** pour effectuer cette mise à l'échelle. En bilan, nous utilisons un total de 2 568 exemples positifs pour notre meilleur modèle.

**Fig. 2:** Extrait d'exemples positifs du dataset

### B. *Négatifs*

Il s'agit de la partie la plus complexe de la génération d'un dataset. Nous avons besoin d'un ensemble d'exemples négatifs de taille similaire qui ne contiennent pas de visage. Une façon d'y parvenir est de prendre n'importe quelle image d'entrée et d'en extraire des ***patchs*** à différentes échelles. Nous disposons d'outils intégrés dans les librairies utilisées, mais elles rendent difficile le fait d'exclure les visages. Mettre à profit un algorithme de ***Sliding Window*** nous permet de remplir cette condition.

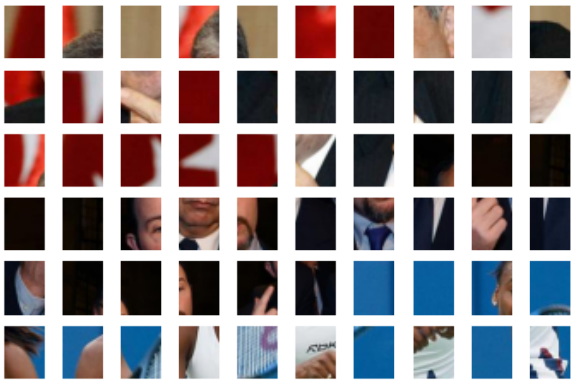

**Fig. 3:** Extrait d'exemples négatifs du dataset

En conclusion, notre dataset comporte ainsi près de 80 000 images de même taille, le tout avec un facteur 30 entre positifs et négatifs : le but étant de coller au plus à la réalité, et équilibrer les classes n'est pas une bonne idée pour la reconnaissance visuelle d'objets.

## III. Vecteur descripteur : Histogramme des Gradients Orientés

L'Histogramme des gradients orientés est une procédure simple d'extraction de caractéristiques qui a été développée dans le contexte de l'identification des piétons dans les images. Les HoG features sont donc utilisés dans ce projet comme vecteur descripteur. La transformation en vecteur HoG est effectué avant chaque prédiction. Même si ce type de vecteur descripteur apporte une solide description de l'image, son utilisation est particulièrement coûteuse et dépendante du nombre de pixels par cellule.

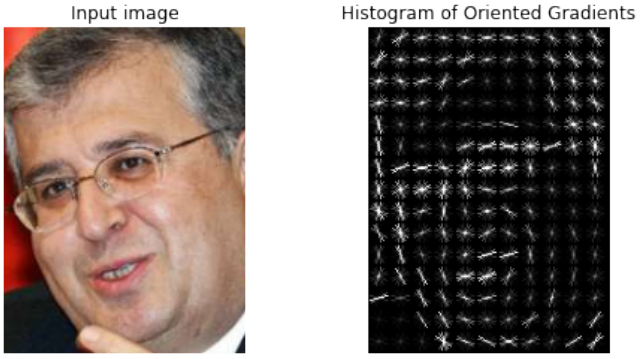


**Fig. 4:** Exemple visuel de transformation d'un visage en HoG feature

## IV. Choix du modèle

Bien que la solution ***AdaBoost Viola et Jones*** ait été envisagée au début du projet, c'est un ***Support Vector Machine*** qui a été retenu réputé plus performant pour la classification d'images. La méthode Viola et Jones a été révolutionnaire dans milieu du machine learning, mais l'implémentation d'image intégrale et des calculs des caractéristiques Haar étaient assez chronophages sans l'aide de la librairies OpenCV qui facilite grandement le travail (mais il ne s'agissait pas d'appliquer un modèle "tout fait").

Le choix s'est tourné vers un SVM à noyau rbf. Après quelques réglages de paramètres, on trouve un coût optimal de 1 (on pourrait avoir tendance à le faire tendre vers de grande valeur puisqu'on va detecter très peu de positifs sur un grand échantillon de test, mais cela à tendance à surentraîner le modèle). On peut voir les résultat avec l'ensemble d'apprentissage via ***Grid Searching*** et ***validation croisée*** sur le tableau ci-dessous:

| Linear SVC | | | SVC "rbf" | | |
|---|---|---|---|---|---|
| C = 1 | C = 10 | C = 100 | C = 1 | C = 10 | C = 100 |
| 0.97855 | 0.97758 | 0.97774 | 0.99984 | 0.99855 | 0.99642 |

**TABLE I:** Cross Validation for training dataset

Le modèle SVM final repose sur une notion ***AdaBoost*** tout de même : la récupération des faux positifs calculés par un premier modèle pour être transformés en exemples négatifs. On apprend alors un nouveau modèle avec toutes ces données. Cette solution permet d'améliorer considérablement la performance du modèle.

On obtient des résultat assez satisfaisant pour notre dataset entier d'entraînement :

| SVM primaire | | |
|---|---|---|
| Rappel | Précision | F-1 Score |
| 0.79600 | 0.74455 | 0.76942 |

**TABLE II:** Score pour un classifieur SVM primaire

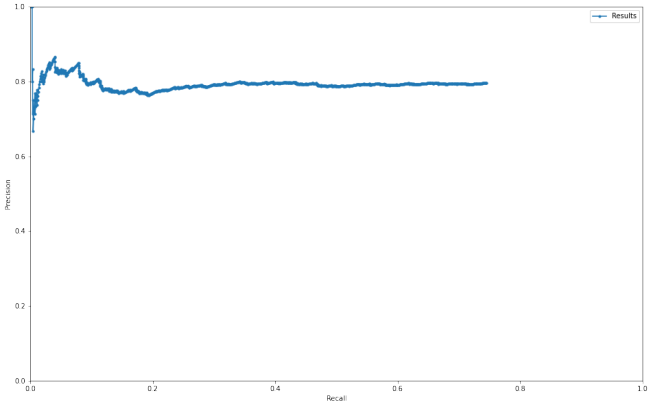


**Fig. 5:** Courbe précision/rappel du classifieur sans False Positive boosting

## V. Pipeline ML

Cette partie s'intéressera plus précisément au pipeline ML (processus global) utilisé pour le projet.

Brief explication du processus utilisé :

- Génération d'exemples positifs (via Data Augmentation)
- Génération d'exemples négatifs (Via algorithme de Sliding Window)
- Concaténation des exemples, mélange puis convertion en HoG feature.
- Construction de modèles à partir du dataset créé
- Choix par validation croisée du modèle
- Mise à l'échelle d'une image test en plusieurs taille
- Application de l'algorithme de Sliding Window sur chacune des images mise à l'échelle
- Ré-itérer pour toutes les images
- Tri par score de détection et suppression des non-maxima
- Calcul des métriques sur le set d'image testé
- Récupération des Faux Positifs et entraînement d'un nouveau modèle boosté
- modèle prêt à servir pour n'importe quelle image

### A. Sliding Window

Parlons un peu de l'algorithme central de notre détécteur. Pour rappel, on applique une boîte rectangulaire de largeur $w$ et de hauteur $h$ fixes qui "glisse" sur une image. Pour approfondir cette notion et détecter les visages qui sont à différentes profondeurs, on doit jouer sur la mise à l'échelle. Il est plus facile de mettre à l'échelle l'image que la boîte. C'est pourquoi, avec une complexité en $n^2$, on applique notre algorithme à différentes tailles et on peut ainsi détecter des visages de taille différentes. Sur ce projet, le choix s'est porté vers une réduction systématique de l'image : on applique notre algorithme pour l'image de base et ainsi détecter les plus petit visages/les visages en profondeur, puis on réduit la taille de l'image jusqu'à un seuil qui permet de détecter un visage sur un portrait. Pour améliorer les résultats et notamment le ***rappel***, la sélection des pas de décalage sur les axes $(x, y)$ ainsi que le nombre de divisions à effectuer sont primordiales. Il est évident que plus on réduit le pas et plus on fait de divisions, plus on détectera de visages et a foritori plus on augmentera le rappel. Cependant, cet avantage à un coût : le temps de calcul. En effet, on ne peut pas se permettre d'effectuer des pas de 1 pixel puisque les images étant en moyenne de taille $(400, 400)$, on aurait 160000 x $nbDivision$ prédictions à effectuer par image. En choisissant un pas de 4 pixels, on arrive à prédire les visages sur une image en 10 secondes de moyenne, ce qui est plutôt correct, sachant que notre but étant d'en prédire 500.

| 10px et 10 divs | | | 4px et 15divs | | |
|---|---|---|---|---|---|
| Rappel | Précision | F1-score | Rappel | Précision | F1-score |
| 0.62501 | 0.74455 | 0.67956 | 0.79600 | 0.74455 | 0.76942 |

**TABLE III:** Score pour un classifieur SVM primaire

### B. Seuil de surface

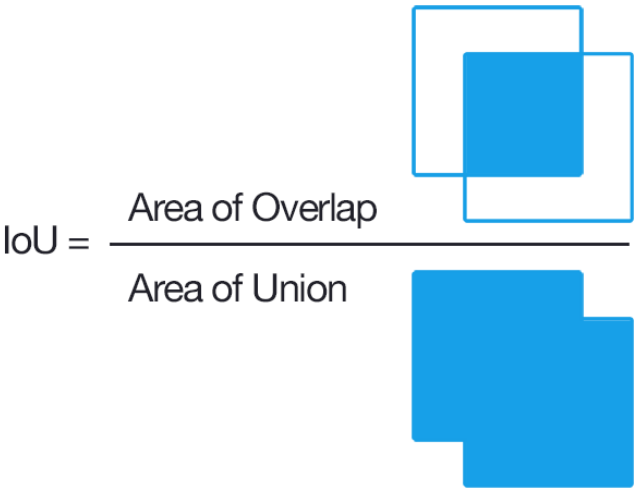


**Fig. 6:** Intersection sur l'union

Pour comparer nos détections, on utilise un outil connu sous le nom anglais "Intersection Over Union". Comme on le voit sur la figure 6, on peut comparer l'aire de deux boîtes englobantes. Le seuil de rigueur est de 0.5. Pour une détection, si l'$IoU$ entre deux boîtes est supérieur à ce seuil, on dit que les boîtes signalent un même et unique visage : on peut supprimer ainsi la boîte ayant le score le plus faible; c'est la ***suppression des non-maxima***.

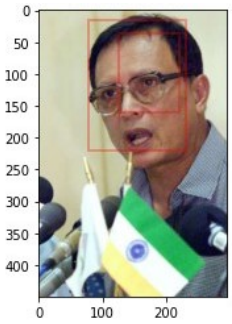


**Fig. 7:** Détection intra-boîte

On peut adapter cet algorithme pour résoudre un problème plus complexe : lors de détections, il arrive qu'on se retrouve dans la situation de la figure 7. Pour parer ce problème, un choix serait de dire : on supprime la boîte englobante la plus petite. Cependant, cela pose problème dans la propreté de détection. Les prédictions qui ont été réalisées pour ce projet font état d'une autre solution : on supprime une boîte si et seulement si la relation entre les boîtes est de 2.5 ou plus (cas où le classifieur détecte une portion du visage) et que cette boîte est couverte à plus de $90\%$ par la plus grande boîte. Ainsi, on s'assure de ne pas supprimer un Vrai positif à l'insu d'une boîte englobante moins précise.

Cette technique nous permet d'améliorer considérablement notre détecteur, puisqu'en l'utilisant avec les paramètres

sliding window optimaux ainsi qu'un classifieur boosté, on obtient ces résultats encourageants :

| Modèle final | | |
|---|---|---|
| Rappel | Précision | F-1 Score |
| 0.86542 | 0.96540 | 0.92728 |

**TABLE IV:** Score final pour l'ensemble train

## VI. Parallélisation pour l'application déployée

On a vu dans la partie précédente que l'analyse d'une image peut prendre un certain temps. Notre application se doit d'être polyvalente notamment dans un sens temporel, c'est pourquoi la parallélisation des détections permet de réduire considérablement le temps que prend notre algorithme. Le but étant d'utiliser la notion de pooling en python afin de réaliser, en parallèle l'algorithme de Sliding Window pour chaque image. A titre informatif, l'extraction et la prédiction des 500 images de test prennent 3h.

## VII. Conclusions

Nous avons développé de A à Z un classifieur de visage. Ce projet aura été l'occasion de découvrir le sujet passionnant qu'est la vision par ordinateur et plus précisément la reconnaissance de visage qui est au coeur des préoccupations depuis plusieurs années maintenant. L'utilisation d'un SVM appliqué à l'algorithme de Sliding Window permet d'arriver à des résultats plus que corrects. Ce projet rassemble aussi tous les outils nécessaires au machine learning et à la vision par ordinateur : validation croisée, algorithme Sliding Window, Data Augmentation, etc... Le problème majeur reste le temps d'exécution qu'il serait intéressant de perfectionner (peut être en utilisant un réseau de neuronne). Je suis globalement content de ma réalisation puisqu'elle m'a beaucoup appris, et le projet a surtout été très formateur : j'avais déjà utilisé la plupart des algorithmes et notions vues en cours pour des projets personnels, mais c'est la première fois que je construis from scratch un algorithme de Sliding Window en utilisant ma propre fonction de suppression des non-maxima.

## VIII. Structure du code source

Le code source se structure en plusieurs parties : on retrouve le dossier modules qui stocke toutes les fonctions utiles à la vision pour ordinateur (telles que le cropping d'image, les algorithmes de Sliding Window et de Scaling ou encore les fonctions de statistiques).

Le fichier ***train.py*** contient le code d'exécution permettant l'entraînement du modèle. Il suffit de lancer le script pour générer le modèle. (Attention, le temps de génération est long, puisqu'on extrait les vecteurs descripteur des 1000 images)

Le fichier ***test.py*** permet de tester une image / des images qui doivent être placées dans un dossier nommé test, sous format jpg. Pour l'execution, il faut avoir placé les images dans le dossier test, puis lancer le script.

On retrouve enfin, le classifieur final utilisé pour la prédiction ainsi que le fichier texte des prédictions sur les 500 images de test en format sav manipulable via la librairie pickle.

## IX. Annexes

### A. Réseaux de neuronnes

Comme réflexion finale, il est possible d'ajouter que les features HOG et d'autres méthodes d'extraction de caractéristiques procédurales pour les images ne sont plus des techniques de pointe. Au lieu de cela, de nombreux pipelines modernes de détection d'objets utilisent des variantes de réseaux neuronaux profonds : une façon de penser aux réseaux neuronaux est qu'ils sont un estimateur qui détermine les stratégies optimales d'extraction de caractéristiques à partir des données, plutôt que de se fier à l'intuition de l'utilisateur.

### B. Liste des librairies utilisées

Numpy, Pandas, OpenCV, skimage, sklearn, matplotlib, pickle, multiprocessing, itertools, glob, pathlib