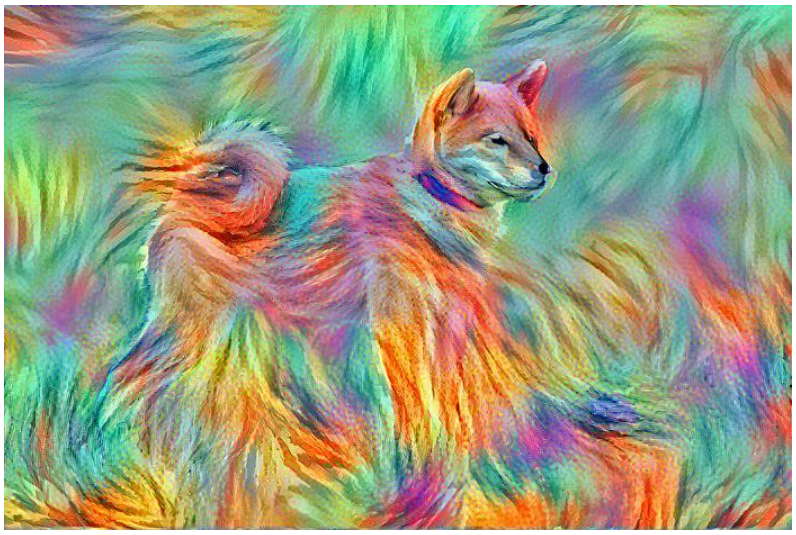
Projet Python

Expérimentations en traitement d'images

Du transfert de style à la détection de points clés du visage humain par webcam



Référent : Prosper Burq prosper.burq@polytechnique.edu

Pascal Jauffret (MS Data Science)

Maxence Brochard (MS Data Science)

Sommaire

1. Introduction

Le projet Python du deuxième semestre est pour nous l'occasion de réaliser un projet de grande ampleur tout en ayant une liberté importante dans le choix du sujet. Nous avons décidé de nous former aux modèles de « Deep Learning » sur lesquels nous sommes néophytes. En effet, nous souhaitons combler partiellement ce manque en travaillant sur un projet d'ampleur qui nous permette d'avoir une première expérience sur ce sujet.

Ayant tous les deux un fort attrait pour le traitement de l'image et le deep learning, nous avons décidé de nous orienter naturellement vers un projet qui combine ces deux aspects. Dès lors, après des recherches intenses et une phase non négligeable d'apprentissage (MOOC, cours écrits, papiers de recherche), notre choix s'est porté sur l'expérimentation sur des réseaux neuronaux convolutifs ou appelé plus communément CNN ou ConvNet.

Dans les sections suivantes, nous vous présenterons le projet initial que nous avons voulu mettre en œuvre, puis le développement et les limites de notre projet, allant de l'implémentation du transfert de style à l'utilisation d'un CNN pour la détection de points clés du visage par webcam

2. Historique du projet

Initialement, nous souhaitions faire une application capable de détecter les fausses peintures. L'idée aurait été de pouvoir trouver un jeu de données regroupant des peintures d'un artiste connu mélangeant originaux et faux. Nous aurions ensuite entraîné un modèle capable de détecter si l'œuvre était un faux. Malheureusement, nous n'avons pas réussi à trouver de « dataset » pertinent.

Lors de nos recherches nous avons découvert le « transfert de style », une application très intéressante des CNN permettant de construire des images réutilisant le style de grands artistes. Dès lors, nous avons décidé d'implémenter cette méthode, avec pour projet à terme de pouvoir utiliser le transfert de style en direct par la webcam.

Suite à l'implémentation, nous nous sommes rendu compte de la complexité temporelle de l'algorithme, et l'impossibilité de l'utiliser en direct par la webcam. Même en utilisant le GPU il était impossible de l'utiliser par la webcam en l'état.

Afin d'implémenter une nouvelle idée, nous avons effectué de nouvelles recherches et avons trouvé un jeu de données combinant visages et points clés associés à ces visages. Nous avons donc eu l'idée d'entraîner un réseau de neurones convolutifs sur ce jeu de données puis d'utiliser OpenCV et la webcam pour détecter les points du visage en direct.

3. Réseaux de neurones convolutifs par webcam

1. Rappels généraux sur les réseaux de neurones convolutifs

Dans cette section, nous allons effectuer quelques brefs rappels sur les réseaux de neurones convolutifs afin de replacer notre travail dans son contexte.

Les réseaux de neurones « convolutifs » ou CNN, sont un type d'architecture de réseaux de neurones hautement utilisés dans le traitement d'images. Ils ont notamment permis l'avancement de l'état de l'art dans le challenge de reconnaissance d'images « ImageNet » (le réseau AlexNet faisant passer en 2012 le pourcentage d'erreur sur ImageNet de 25% à 15%).

Ils sont caractérisés par leurs couches dites de « convolution ». Au sein de ces couches, on réalise une convolution sur l'ensemble de l'image. C'est-à-dire qu'au lieu de réaliser un simple produit scalaire sur l'ensemble des points de l'image (comme dans une couche entièrement connectée), on va réaliser un produit scalaire sur une petite zone de l'image (un « patch ») que l'on va ensuite déplacer sur l'ensemble de l'image. On obtient ainsi une matrice de tous les produits scalaires pour toutes les zones de l'image.

On génère ensuite une non-linéarité en utilisant une fonction dite d'activation, la plus classique pour les filtres de convolutions étant la ReLu (« Rectified Linear Unit ») qui fixe à zéro les valeurs négatives. On arrive ainsi à garder la cohérence spatiale locale, tout en ayant besoin de beaucoup moins de paramètres.

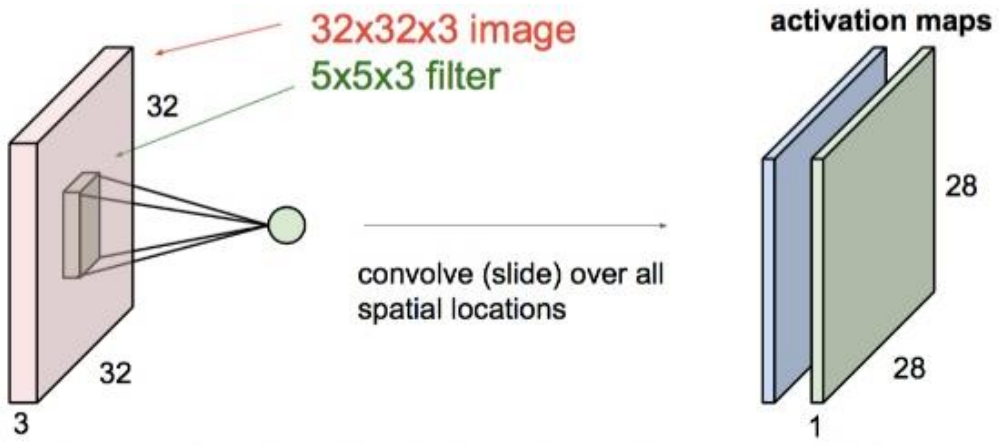


Figure 1 : Une couche de convolution[1]

La figure précédente nous montre un exemple de couche de convolution. On a une image en entrée sur 3 canaux (RGB en général), et l'on va faire varier un petit « patch » sur l'ensemble des localisations possibles de l'image pour pouvoir ainsi construire après ReLu une « carte d'activation » (ou « activation map » ou filtre) qui sera capable de détecter un motif spécifique.

Dans les architectures CNN, il est aussi très courant d'utiliser des couches dites de « poolings », elles permettent de réduire la dimensionnalité afin de limiter le nombre de paramètres tout en conservant le signal. La plus commune est celle dite de

[1] https://www.slideshare.net/nmhkahn/case-study-of-convolutional-neural-network-61556303

« max pooling », elle fonctionne un peu comme une couche de convolution, sauf qu'au lieu de calculer un produit scalaire sur chacune des zones, on conserve la valeur maximum.

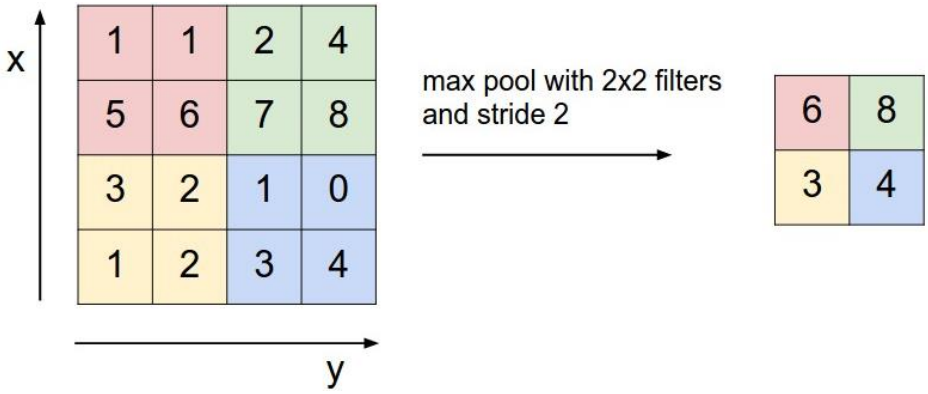


Figure 2 : Une couche de "max pooling"[2]

2. L'architecture VGG

VGGNet est le réseau neuronal convolutif qui a gagné le challenge « ImageNet » de 2014. Il possède une architecture caractéristique assez simple et régulière que nous avons utilisée comme base dans la suite de notre projet. L'architecture de VGG est simplement l'enchainement de plusieurs « block » contenant chacun plusieurs couches de convolutions avec des ReLu en activation suivi d'un « max pooling ». Le réseau se finit ensuite par des couches entièrement connectées classiques.

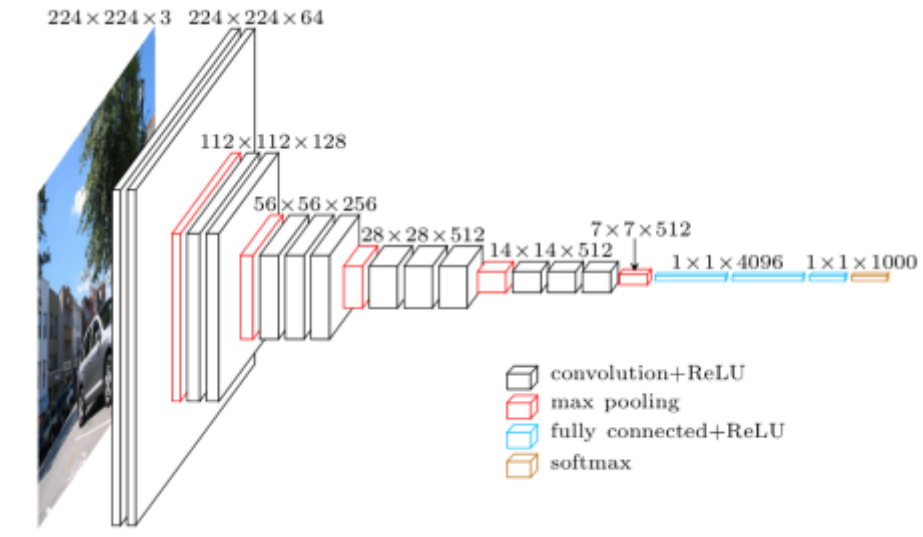


Figure 3 : Architecture de VGGNet[3]

[2] http://cs231n.github.io/convolutional-networks/
[3] https://www.cs.toronto.edu/~frossard/post/vgg16/

L'enchainement des « blocks » de VGG permet au réseau d'apprendre des « patterns » de plus en plus complexes. Si l'on construit les images qui maximisent l'activation de certains filtres de VGG, plus on prend des « blocks » loin dans l'architecture, et plus les images seront complexes. En effet, les motifs simples sont combinés pour pouvoir en construire des plus complexes.

La figure suivante montre des exemples d'images maximisant l'activation de filtres issus du premier block (en haut), puis du cinquième (en bas).

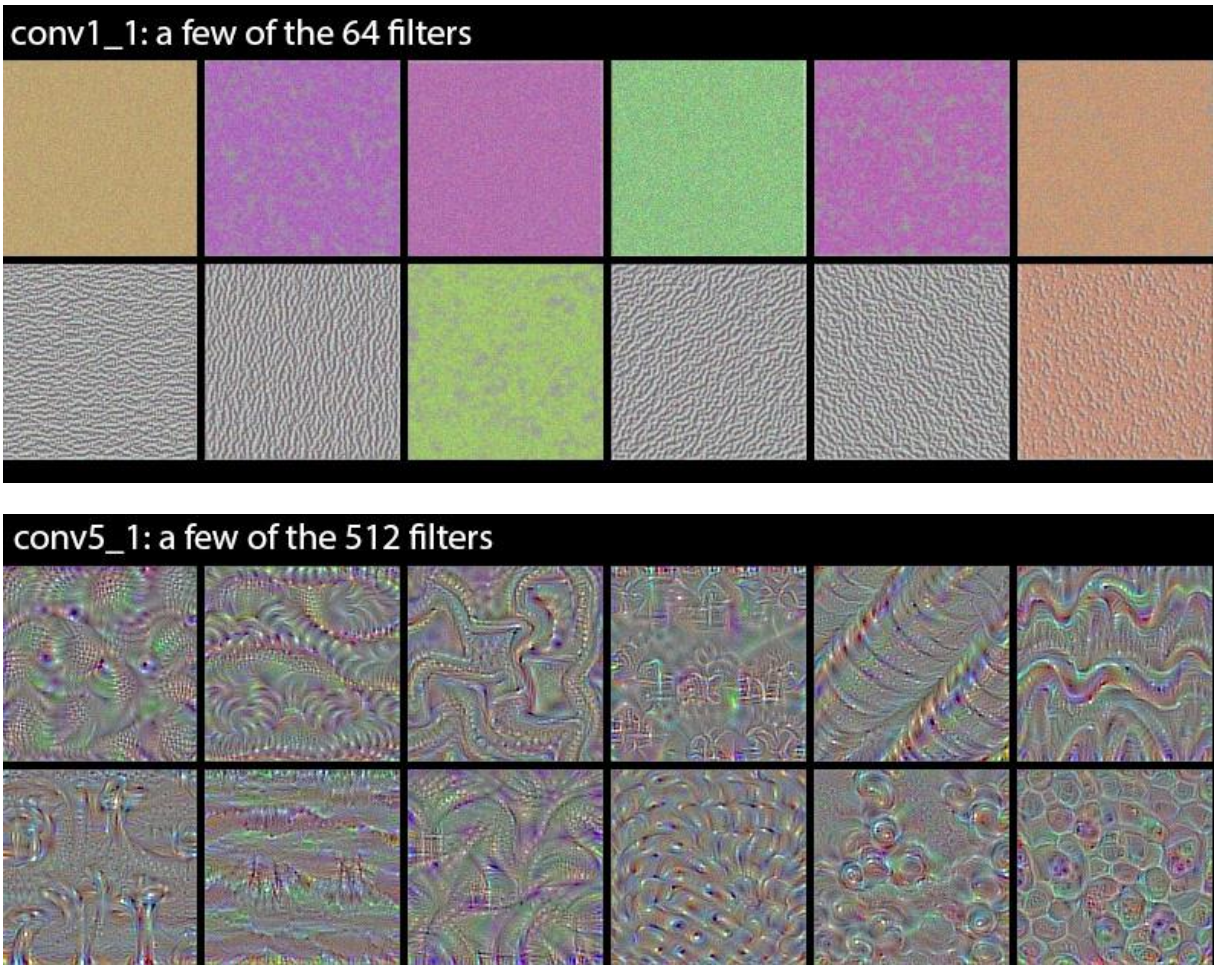


Figure 4 : Exemple d'images maximisant l'activation de filtres[4]

3. Un premier prototype à la webcam

Afin de nous familiariser avec l'utilisation d'un réseau de neurones sous la librairie Keras, l'une des premières étapes de notre projet a été de connecter un réseau pré-entraîné sur notre webcam afin de tester ce que l'on pouvait y détecter. L'accès à la webcam a été réalisée à l'aide de la librairie OpenCV.

[4] https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

Nous avons ainsi pu tester VGG et observer comment celui-ci pouvait être efficace dans la détection d'objets sur des scènes complexes. La figure suivante montre un exemple de détection de VGG en direct de la webcam : l'algorithme a su reconnaitre une bouteille d'eau avec une probabilité de 0.992.

Figure 5: Reconnaissance d'objet par la webcam

L'algorithme ayant été entraîné pour sur le challenge « ImageNet », il n'est pas forcément le plus performant pour détecter les objets dans une scène complexe. Il existe néanmoins des approches pour segmenter l'image afin de détecter plus précisément l'emplacement d'objets.

Le code utilisé pour la reconnaissance d'objets à partir de la webcam nous a été utile par la suite pour implémenter la reconnaissance de points clés du visage.

4. Transfert de style

Le transfert de style permet de combiner deux images afin d'appliquer le « style » artistique d'une image sur l'autre. Nous avons implémenté la méthode décrite dans l'article[5] de *Gatys et al. 2015.*

1. Principe de la méthode

L'idée du transfert de style est assez simple. En utilisant une descente de gradient, l'algorithme génère une image qui minimise une perte combinant l'image d'origine ainsi que le « style » extrait à partir de l'image de « style ». On part ainsi d'une image de « style », d'une image de « contenu » et d'un bruit blanc.

Grâce à un CNN pré-entraîne (ici VGG), on peut récupérer les activations des filtres de convolution pour les utiliser comme représentation de l'image (vu que le réseau a été entraîné, ces filtres sont assez exhaustifs).

On obtient la perte associée au « contenu » en faisant simplement la somme des erreurs quadratiques entre les activations des filtres de l'image de « contenu » et ceux du bruit blanc.

La découverte fondamentale de cet article est que la matrice de Gram associée à une couche de convolution permet d'encapsuler l'information de « style » (de texture) d'une image.

On note F_{ik}^{L} le i-ème filtre de la couche de convolution L à la position k de l'image.

La matrice de Gram associée à la couche de convolution Lest définie par :

$$G_{ij}^{L} = \sum_{k} F_{ik}^{L} F_{jk}^{L}$$

La matrice de Gram représente les produits scalaires entre les différents filtres de la couche de convolution. Cela correspond à une sorte de « corrélation » entre les filtres. On construit ainsi la perte de « style » qui est simplement la somme des erreurs quadratiques entre les matrices de Gram des filtres de convolutions sélectionnés.

En partant d'un bruit blanc, et en utilisant une descente de gradient, on est capable de reconstruire une image maximisant la perte de style associée à une image de « style » ou bien à une image de « contenu ». On peut ainsi visualiser le style associé à différents étages du réseau de neurones (en utilisant les matrices de Gram, ou bien les filtres de convolutions plus ou moins haut dans le réseau). La figure suivante permet de visualiser ces reconstructions.

[5] http://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf

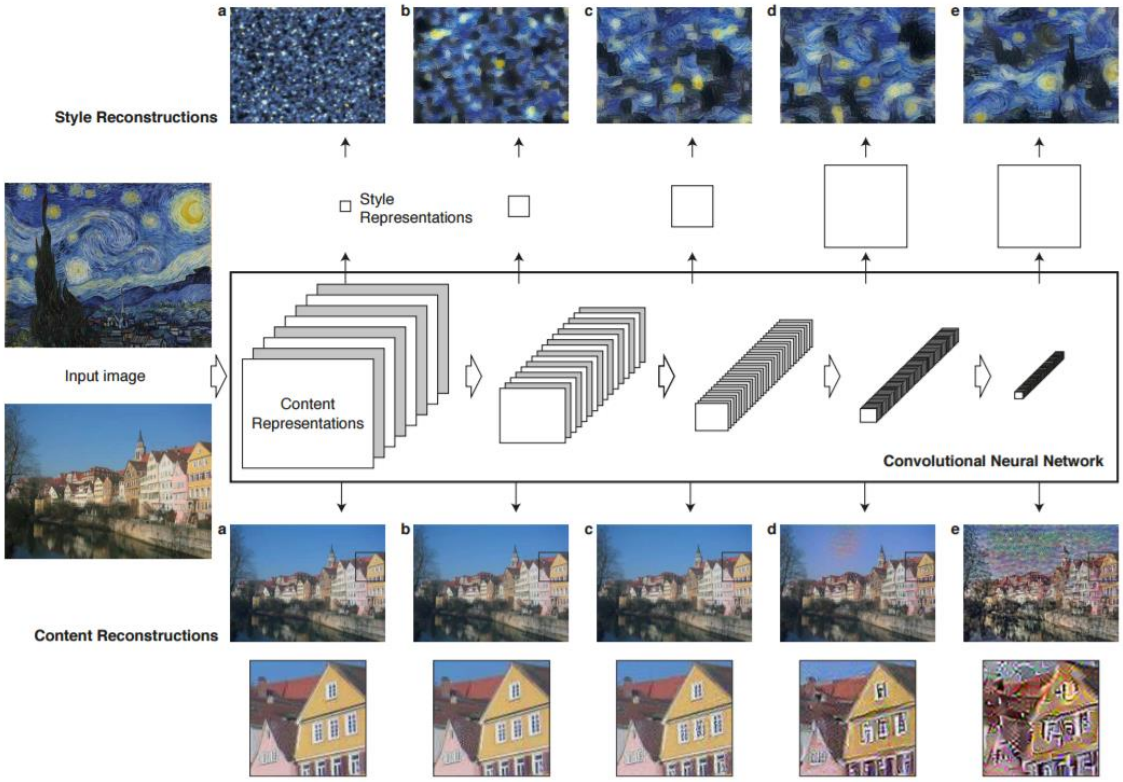


Figure 6 : Reconstruction à partir de la perte de "style" ou de "contenu" (illustration reprise de l'article de Gatys)

Ces visualisations ont permis aux chercheurs de sélectionner les couches qu'ils allaient utiliser pour la perte combinant le style et le contenu.

On construit ainsi une perte combinant la perte de style et la perte de contenu. On envoie une image de bruit blanc dans le réseau, et grâce à un algorithme de descente de gradient (ici L-BFGS), on est capable de mettre à jour l'image d'origine de façon à ce qu'elle minimise la perte.

La figure suivante résume la méthode du transfert de style. On n'utilise que la couche 4 pour le contenu, et les couches 1, 2, 3, 4, et 5 pour le style.

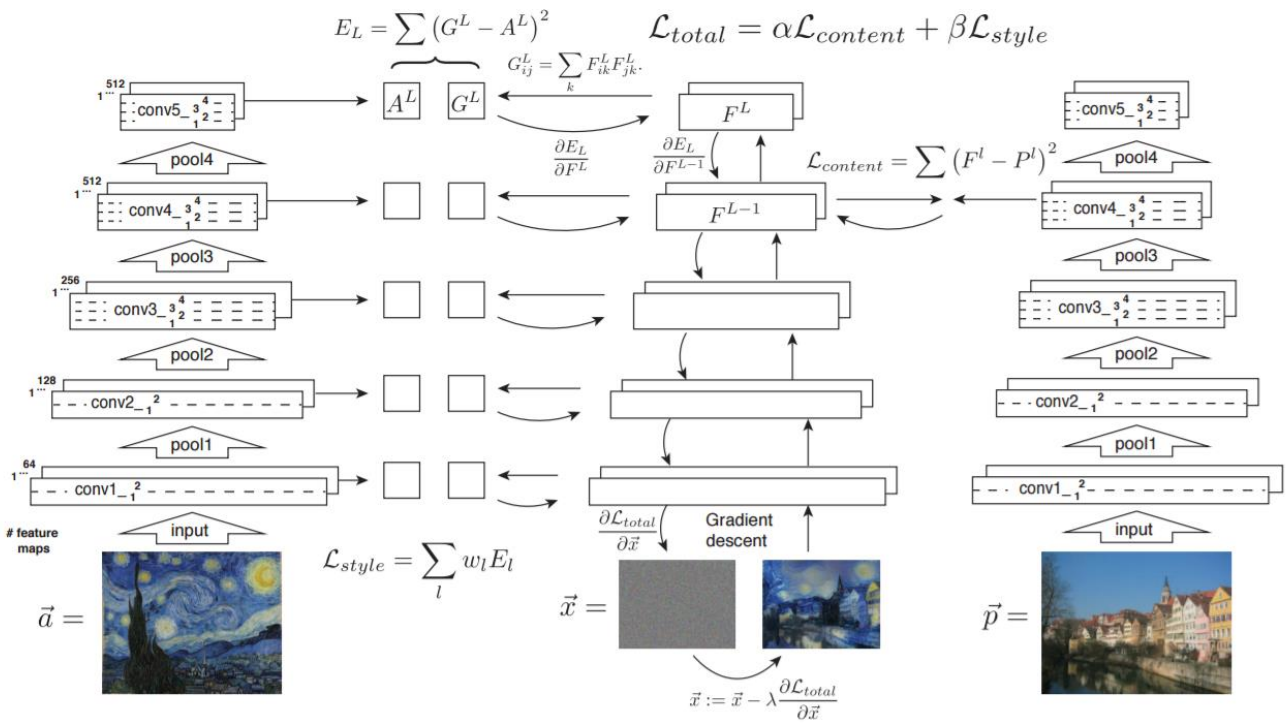


Figure 7 : Procédure du transfert de style, (illustration reprise de l'article de Gatys)

2. Implémentation

L'implémentation de l'algorithme n'aura pas été triviale. Nous avons dû nous familiariser avec l'API de Keras et de TensorFlow, et bien comprendre le fonctionnement des tenseurs.

Nous avons essayé d'implémenter nous-même la descente de gradient, soit par gradient stochastique (sans grand succès) ou soit en utilisant l'accélération de Nesterov mais l'algorithme ne parvenait pas à converger vers d'aussi bons résultats que L-BFGS (possible erreur d'implémentation). La figure suivante nous présente la différence de convergence pour une image entre L-BFGS et l'accélération de Nesterov.

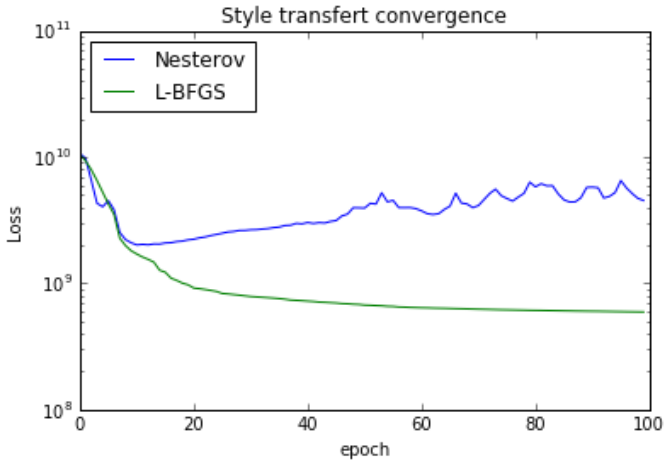


Figure 8 : Différence de convergence entre les deux algorithmes de descente de gradient

Voici deux exemples d'images obtenus soit par Nesterov soit par L-BFGS. On peut par ailleurs voir que la méthode de Nesterov n'a pas réussi à converger vers une valeur optimale.

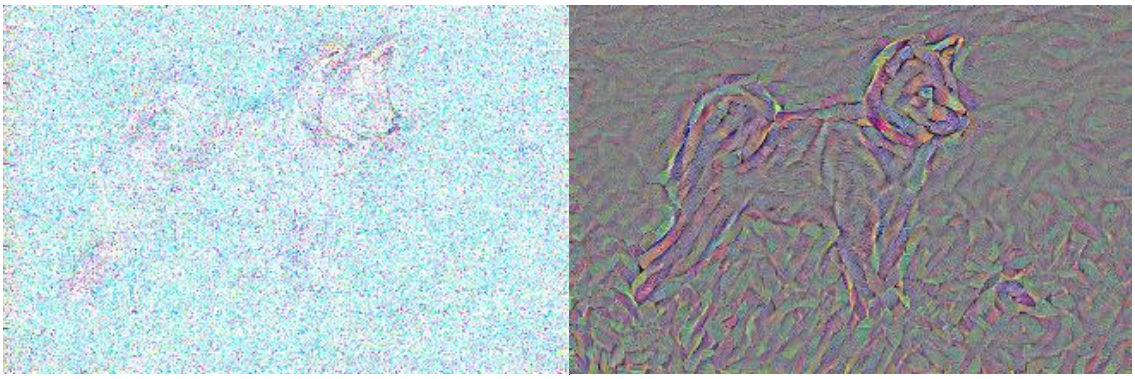


Figure 9: Images obtenus par accélération de Nesterov (à gauche) et L-BFGS (à droite)

Pour utiliser L-BFGS, nous nous sommes basés sur l'implémentation fournie par la librairie Scikit. Nous avons réalisé une légère optimisation permettant de ne calculer les gradients qu'une seule fois par itération (et non pas deux fois) en stockant les informations dans un dictionnaire.

Par ailleurs, nous avons mis en place l'utilisation du GPU (grâce à Cuda et CuDNN) pour réaliser le transfert de style, et nous avons pu diviser le temps de calcul par 6 : nous réalisons un transfert de style en une dizaine de minutes là ou il nous fallait auparavant plus d'une heure.

La figure suivante présente un exemple de transfert de style avec plusieurs pondérations entre perte de style et perte de contenu :

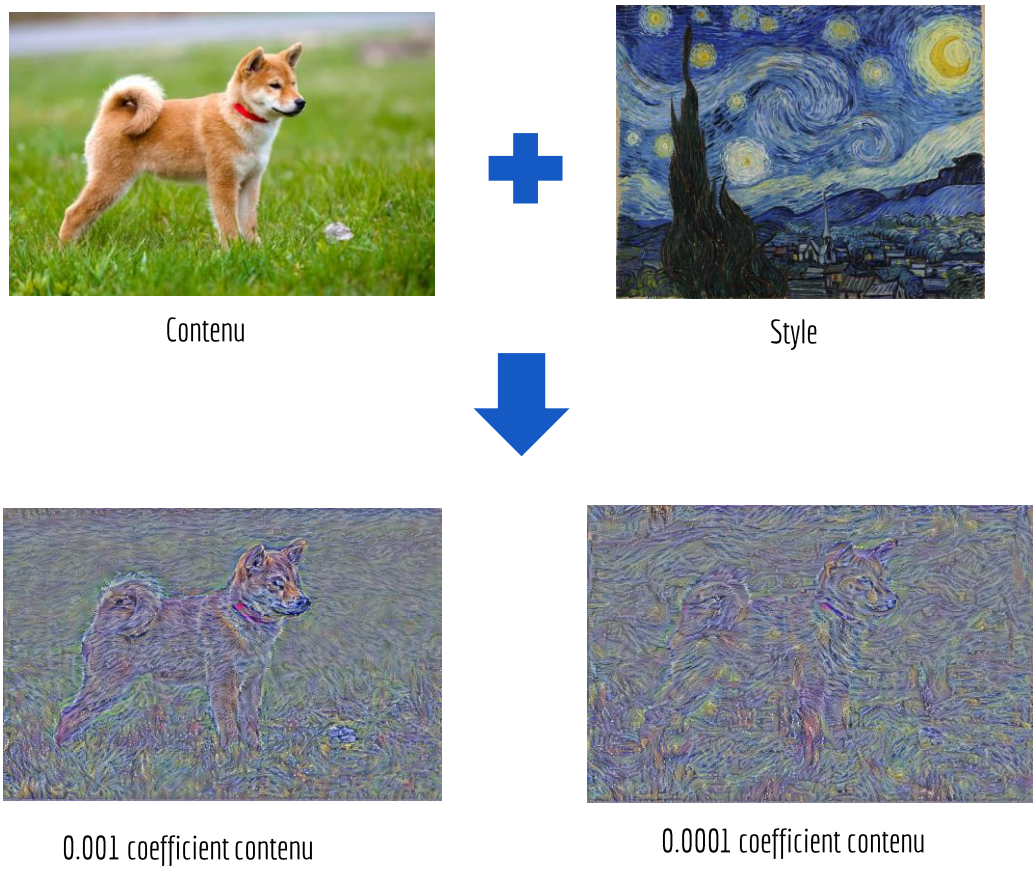


Figure 10 : Exemples de transfert de style

3. Limites

Bien qu'accéléré par GPU, l'algorithme de transfert de style reste consommateur de ressources : générer une image prend une dizaine de minutes. En effet, l'algorithme construit une nouvelle image de façon ad-hoc pour chaque style et contenu. Il est possible d'améliorer la vitesse de convergence en ne partant pas d'un bruit blanc mais directement de l'image de contenu, mais cette approche est beaucoup plus déterministe.

A l'origine nous voulions tenter de connecter notre transfert de style à la webcam, mais face à ce temps de calcul, cela s'est avéré être impossible. Il existe des papiers de recherche proposant des approches permettant de réaliser un transfert de style beaucoup plus rapide (en construisant notamment au préalable un réseau convolutif dédié à un « style » en particulier). Ces algorithmes nécessitent néanmoins une connaissance beaucoup plus approfondie des outils informatiques (implémenter nous-même certaines couches par exemple). Nous avons pris le parti de ne pas nous aventurer sur ce terrain par manque de temps.

Le tableau suivant montre un ordre de grandeur des gains de temps, sachant que le temps d'exécution pour L-BFGS dépend très fortement du paramétrage. Nous sommes justement passés au GPU car il était beaucoup trop long d'exécuter un transfert de style sur une centaine d'itération avec L-BFGS par CPU.

Algorithme sur une itération	GPU	CPU
Nesterov	10.3 secondes	40.4 secondes
L-BFGS	12.7 secondes	4.11 minutes

Tableau 1 : Comparaison des temps d'exécution entre CPU et GPU

5. Détection des points clés du visage en direct depuis la webcam

Ayant déjà un système prêt à utiliser la webcam (la détection d'objets de la partie 1), nous sommes partis à la recherche d'un jeu de données nous permettant de proposer une nouvelle idée à implémenter.

1. Les données

Nous avons ainsi trouvé un jeu de données Kaggle[6] proposant d'associer des visages avec des points clés. Le jeu de données se compose d'images en 96x96 noir et blanc associé à des points caractéristiques (yeux, nez, bouche, sourcil).

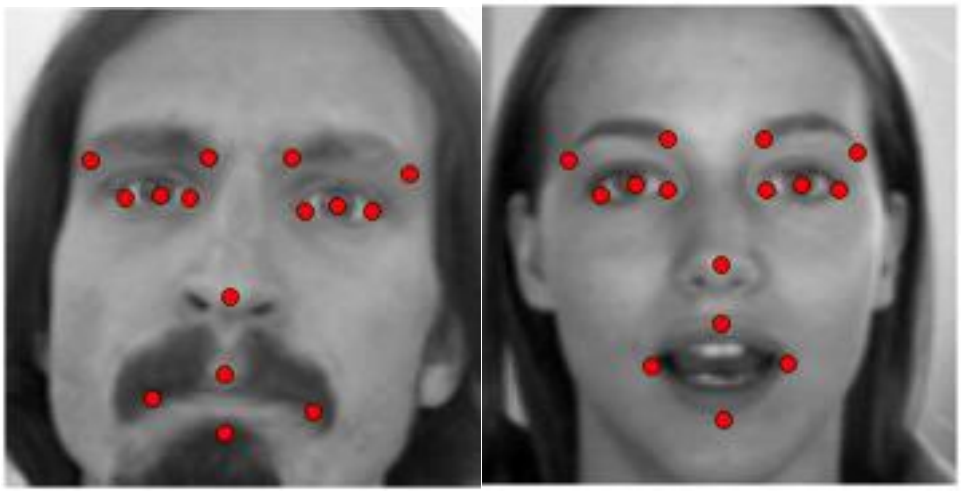


Figure 11 : Exemples de visages et points clés associés

Les données contiennent 7049 individus, mais elles sont complètes (i.e. contiennent tous les points) pour seulement 2140 individus. Nous avons décidé de supprimer les 4909 autres individus en estimant qu'avec 2140 lignes nous aurions suffisamment de données pour entraîner un algorithme relativement robuste.

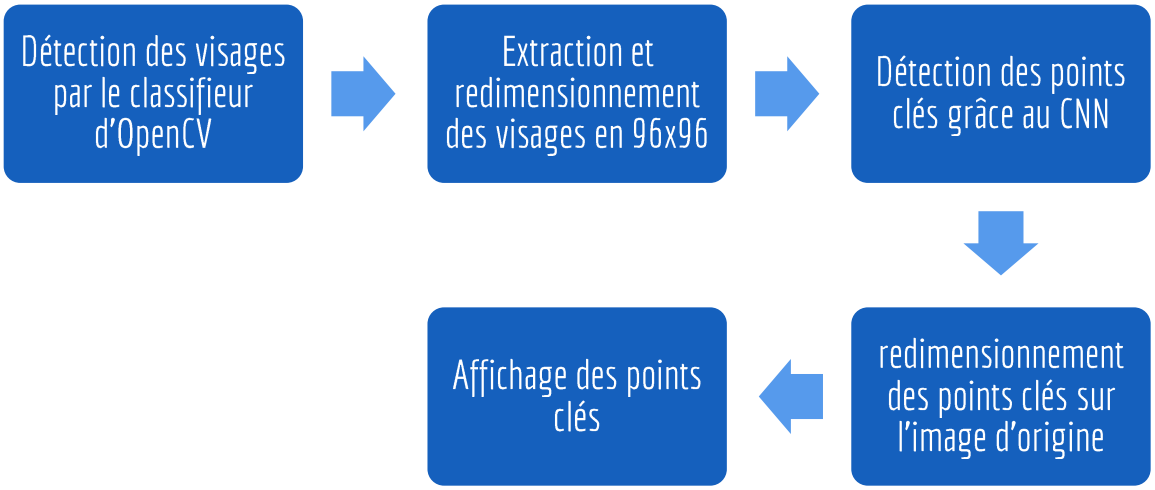
2. L'architecture du système

Nous avons eu l'idée de combiner plusieurs classifieurs afin de proposer un système assez efficace pour détecter des points clés du visage tout en utilisant nos données.

Nous avons ainsi combiné un détecteur de visage intégré à OpenCV (utilisant un classifieur basé sur des caractéristiques de Haar[7]) à un réseau convolutif que nous avons entraîné sur nos données.

Le schéma ci-dessous présente les étapes réalisées à chaque « frame » de la webcam :

6 https://www.kaggle.com/c/facial-keypoints-detection
7 Caractéristiques utilisées en vision par ordinateur pour la détection d'objet dans des images numériques.



On commence par détecter les visages grâce au classifieur basé sur les caractéristiques de Haar, on extrait ces visages, et on les redimensionne pour les données en entrée du CNN. En sortie du CNN, on récupère les points clés au format 96x96 que l'on redimensionne pour être à la taille de l'image d'origine. On obtient ainsi un classifieur efficace en direct et relativement précis (comme on va le voir dans le chapitre suivant).

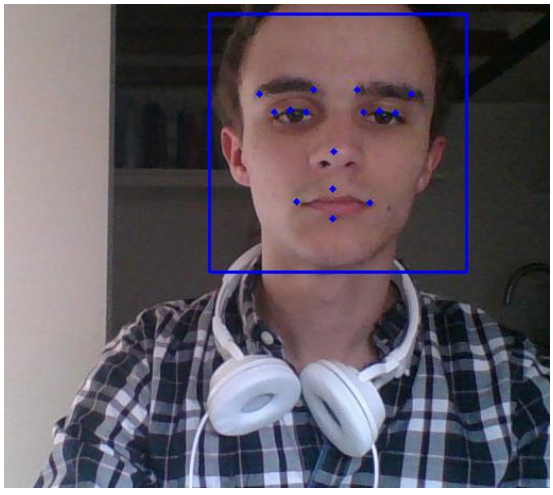


Figure 12: Exemple de détection des points clés et du visage

3. L'entraînement du réseau

En premier lieu, nous avons séparé nos données en ensemble d'entraînement et de test (à une proportion 85/15). Le temps d'apprentissage étant long, nous voulions obtenir un algorithme avec un maximum de données tout en ayant un score pertinent de validation (d'où la proportion élevée dans l'ensemble d'entraînement). Néanmoins, nous disposions encore de 321 visages pour tester notre algorithme.

L'entraînement du réseau s'est ensuite fait en plusieurs étapes. Nous sommes repartis de l'architecture de VGG avec un seul « block » de convolution, puis nous avons ensuite essayer d'en rajouter de plus en plus.

L'idée est de partir d'une architecture la plus simple possible, d'observer les résultats obtenus, puis de complexifier le modèle. Lors du passage à trois « block » de convolution, nous avons été confronté à du « surapprentissage » et nous avons pu corriger cela en ajoutant des couches de « dropout ». Le dropout permet de limiter le surapprentissage en n'autorisant qu'une certaine fraction des poids à être modifiée lors d'une passe de rétropropagation du gradient.

Nous nous sommes rapidement rendu compte qu'il était possible d'obtenir un score très faible en indiquant systématiquement les yeux et le nez de façon centrée sur le visage (pour obtenir des points clés fixes, peu importe le visage). Afin de disposer d'un algorithme pertinent, il était donc nécessaire de viser des niveaux de précision relativement élevé (moins de 2 pixel d'erreur quadratique moyenne par exemple) afin d'éviter d'avoir simplement des points clés fixe à la moyenne des yeux nez etc. Le rajout de « block » de convolution permet d'atteindre ces niveaux de précisions.

La figure suivante permet de comparer les différents résultats que nous avons obtenus pour nos architectures. On peut voir comment le dropout permet de limiter le surapprentissage.

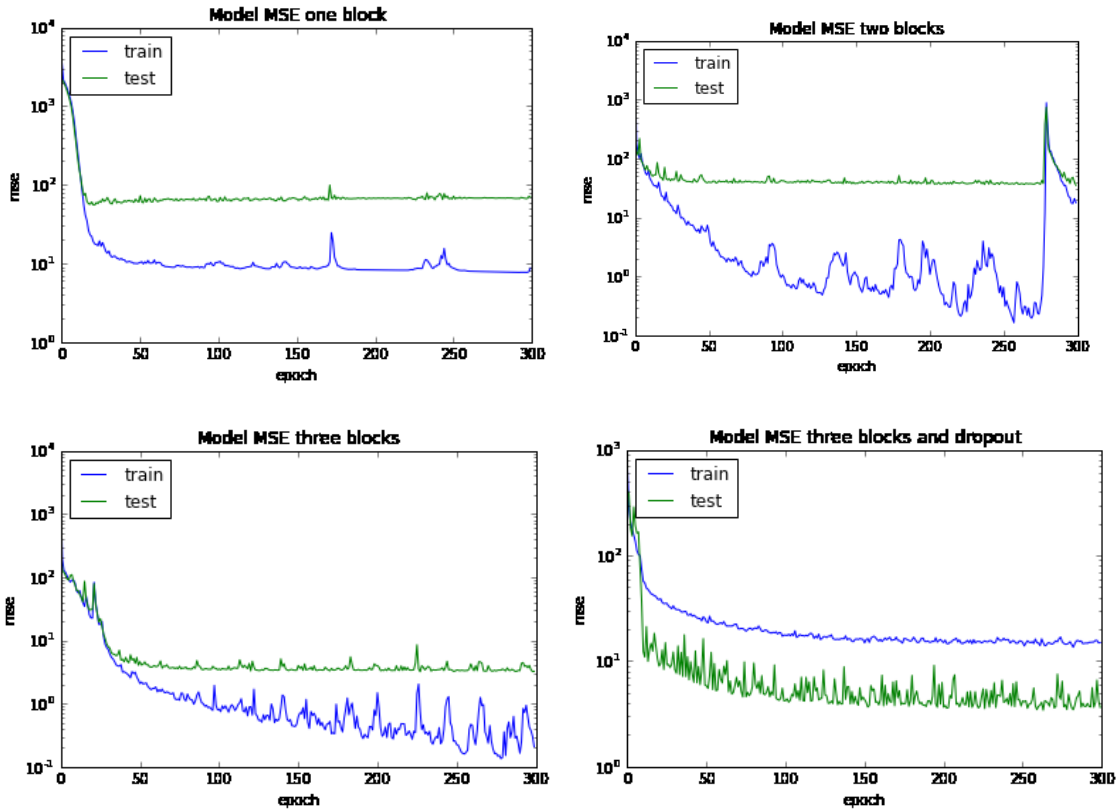


Figure 13 : Convergence des différentes architectures de CNN

Même sur GPU, l'entraînement sur un nombre d'itérations suffisant pour atteindre le minimum d'erreur est relativement long (plusieurs heures). Il faut donc porter une attention particulière quant à la validité du code avant de le lancer.

6. Conclusion

En conclusion, ce projet aura été riche en apprentissage. Nous avons eu l'occasion de manipuler des réseaux de neurones et quelque unes de leurs applications. Le sujet est vaste et nous n'en avons aperçu qu'une mince partie.

Concernant notre algorithme, nous aurions pu aller plus loin et tenter d'améliorer les scores de prédiction. La difficulté réside dans la capacité de trouver un compromis entre précision de l'algorithme et taille du modèle : avec trop de poids, la vitesse d'affichage diminue et l'ordinateur risque d'être à court de mémoire pour l'entraînement.

Il existe de nombreux prolongements à notre projet. Nous pourrions ainsi :

- Construire une application de filtres de visages comme sur « SnapChat »
- Tenter d'implémenter de nouvelles méthodes de transfert de styles
- Entraîner un algorithme avec les points clés du visage pour de la détection d'identité
- Relier le « style transfer » et la détection des points de visage pour construire une image de visage avec différents styles sur différentes parties du visage