



UNIVERSITÉ PARIS CITÉ

PERCEPTION, ACQUISITION ET ANALYSE D'IMAGE

Neural Style Transfer

Barthélémy Metzlé

supervisé par
Jonathan Vacher

Table des Matières

1	Introduction	2
2	Présentation de L'article	3
2.1	Rappel sur les réseaux de neurones convolutif	3
2.2	Représentation du contenu	3
2.3	Représentation des textures	4
2.4	Transfert de Texture	5
3	Expérimentations	6
3.1	Présentation de l'algorithme	6
3.2	Expérimentations	7
3.2.1	Reproduction des expérimentations de l'article	7
4	Conclusion	10
A	Illustration du Réseau	12
B	Quelques Images Supplémentaires	13
C	Code	14

1 Introduction

Dans l'article intitulé *Image Style Transfer Using Neural Networks* Leon A. Gatys, Alexander S. Ecker et Matthias Bethge présente une méthode permettant de transferer le style d'une image sur une autre en utilisant un réseau de neurones convolutionnel. Le transfert de textures d'une image sur une autre de manière fidèle est un problème récurrent d'imagerie. Il existe plusieurs méthodes de transferts de textures mais ces dernières sont généralement orientées vers de le transfert des textures de hautes fréquences.

Le but de la méthode présentée dans cet article est donc de pouvoir extraire le contenu de l'image dont on veut modifier la texture et d'y apposer la texture d'une autre image. L'utilisation d'un réseau de neurone convolutionnel est particulièrement adaptée à la reconnaissance d'objet. Ainsi il est facile d'obtenir une représentation du contenu d'une image lorsque celle ci est passé dans un réseau convolutionnel profond. On peut par un procédé différent récupérer les informations sur textures des images en utilisant toujours une réseau convolutionnel profond. L'enjeu est donc de pouvoir mettre en œuvre un algorithme plus efficace qui ne se borne pas uniquement à l'analyse superficielle des caractéristiques visuelles, mais qui parvient également à intégrer et traiter des aspects plus complexes comme le style artistique d'une peinture.

2 Présentation de L'article

2.1 Rappel sur les réseaux de neurones convolutif

Pour rappel, dans un réseau de neurones convolutif (CNN) chaque couche peut être vu comme une banque de filtre qui se déplace sur l'image. Dans un réseau de neurones convolutionnels une feature map ou carte de caractéristique est le résultat de l'application d'un filtre ou d'un noyau de convolution à l'image d'entrée ou à une carte de caractéristiques précédente.

Cette carte de caractéristiques représente une version modifiée de l'image d'entrée, où certaines caractéristiques sont mises en évidence tandis que d'autres sont atténuerées. Chaque filtre dans la couche convolutionnelle est conçu pour réagir à un type spécifique de caractéristique visuelle, comme les bords, les textures ou les motifs. En passant sur l'image, le filtre produit une carte de caractéristiques qui capture la présence et l'intensité de ces caractéristiques à différentes positions spatiales.

À mesure que l'information progresse à travers les couches successives du réseau, les filtres deviennent capables de reconnaître des caractéristiques de plus en plus complexes. Les premières couches peuvent détecter des caractéristiques simples comme les lignes et les bords, tandis que les couches plus profondes peuvent identifier des éléments plus complexes tels que des formes ou des objets spécifiques. Ce processus est facilité par l'utilisation de fonctions d'activation non linéaires, qui permettent au réseau de modéliser des relations complexes et de hiérarchiser certaines caractéristiques par rapport à d'autres.

En plus des couches convolutionnelles, les réseaux CNN comprennent généralement des couches de pooling, qui réduisent la dimension spatiale de la carte de caractéristiques et contribuent à rendre le modèle plus robuste aux variations de position des caractéristiques dans l'image d'entrée. Enfin, les couches entièrement connectées en fin de réseau utilisent les caractéristiques extraites pour effectuer des tâches telles que la classification ou la reconnaissance d'objets. Ainsi, les CNN transforment progressivement l'image brute en une représentation abstraite et riche en informations, permettant de prendre des décisions précises basées sur le contenu visuel.

2.2 Représentation du contenu

On a vu dans la partie précédente comment fonctionnait un réseau de neurones convolutif. Pour notre projet on utilise une architecture appelée VGG19 constituée de 16 couches convolutionnels, de 5 couches de pooling et de couches entièrement connectées. Dans notre cas nous n'utilisons pas les couches entièrement connectées qui servent normalement à discriminer le contenu des images. A chaque couche convolutionnel l le réseau définit N_l filtres qui permettent de calculer N_l cartes de caractéristiques de taille $M_l = h * L$ avec h la hauteur de la carte et L sa largeur. Ainsi la réponse d'un filtre à la couche l est stocker dans une matrice $F^l \in \mathbb{M}_{N_l \times M_l}(\mathbb{R})$

Ainsi si l'on veut visualiser les informations qui sont contenues dans le réseau,

il suffit de faire une descente de gradient sur un bruit blanc pour qu'il coincide au maximum avec les réponses des filtres de chaque couche.

Pour chaque couche on définit donc l'erreur quadratique par rapport au réponse des filtres de la couche l comme étant

$$\mathcal{L}_{\text{contenu}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2$$

La dérivée de la fonction de perte par rapport la réponse d'un filtre de la couche l est donnée par

$$\frac{\partial \mathcal{L}_{\text{contenu}}}{\partial F_{ij}^l} = \begin{cases} (F^l - P^l)_{ij} & \text{si } F_{ij}^l > 0, \\ 0 & \text{si } F_{ij}^l \leq 0. \end{cases}$$

Ici on suggère que \vec{x} est l'image originale que l'on souhaite approcher et \vec{p} est le bruit blanc que l'on modifie à chaque itération de l'algorithme. On obtient une image dont les cartes de caractéristiques à chaque couche se rapprochent au maximum de celles de l'image de départ.

On a rappelé au dessus que lorsqu'un réseau de neurones convolutif est entraîné pour la reconnaissance d'objets, il permet d'obtenir des informations sur les objets contenus dans l'image. Plus le réseau est profond, plus ces informations sont précises. On a ainsi que les cartes de caractéristiques d'une couche supérieur sont relativement invariantes par rapport à l'apparence globale de l'image. La réponse n'est alors pas contrainte par la valeur exacte des pixels de l'image, nous permettant ainsi d'obtenir une représentation fidèle du contenu de l'image.

2.3 Représentation des textures

Pour obtenir une représentation de la texture d'une image on essaie de trouver la corrélation entre les réponses aux filtres de chaque couche. On obtient ces caractéristiques en calculant la matrice de Gram de chaque couche $G^l \in \mathbb{M}_{N_l \times N_l}(\mathbb{R})$ définie telle que

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

Pour rappel pour une famille de vecteurs $V = \langle v_1, \dots, V_n \rangle \forall n \in \mathbb{N}_+^*$ la matrice de Gram G de taille $n \times n$ est la matrice donnée par $G_{ij} = v_i \cdot v_j$. Dans notre cas on calcule ce produit en considérant la famille de vecteur composée par les cartes de caractéristiques à chaque couche. La matrice de Gram est utilisée pour mesurer les corrélations entre les cartes de caractéristiques de chaque couche. En récupérant ces informations sur plusieurs couches on peut ainsi obtenir des informations sur la texture globale de l'image. Sans obtenir d'informations sur le contenu de l'image. Pour récupérer les informations contenues dans ces matrices il suffit de minimiser l'erreur quadratique, pour chaque couche, entre la matrice de Gram de l'image source, dont on veut récupérer la texture et celle

d'un bruit blanc en faisant une descente de gradient sur celui ci Pour chaque couche on definit :

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

Où G^l est la matrice de Gram de l'image originale et A^l est celle que l'on cherche à Optimiser.

Ainsi la perte total sur la texture est défini par

$$\mathcal{L}_{texture}(\vec{p}, \vec{x}) = \sum_l \omega_l E_l$$

où ω_l est un poids donné à chaque couche. Ainsi la dérivée de E_l par rapport à chaque couche est donnée par

$$\frac{\partial \mathcal{L}_{texture}}{\partial F_{ij}^l} = \begin{cases} (F^l)^T (G^l - Al)_{ij} & \text{si } F_{ij}^l > 0, \\ 0 & \text{si } F_{ij}^l \leq 0. \end{cases}$$

2.4 Transfert de Texture

Pour le transfert et la synthèse de texture il suffit donc de résoudre ces problèmes d'optimisation en parallèle. En minimisant simultanément la fonction de perte sur le contenu d'une couche et la fonction de perte calculée sur la texture. Pour ça on initialise un bruit blanc et ce sont les valeurs des pixels de cette image que nous allont modifier. La fonction de perte total est donnée par

$$\mathcal{L}_{total}(\vec{p}, \vec{x}) = \alpha \mathcal{L}_{contenu}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{texture}(\vec{p}, \vec{x})$$

On se retrouve avec un problème d'optimisation unique. Pour modifier l'apparence de l'image à synthétiser on peut donc jouer sur les paramètres α et β , sur les différentes valeurs de w_l (dans notre cas on restera sur $\omega_l = \frac{1}{5}$, et sur la couche convolutionnel choisi pour la représentation du contenu. Ainsi pour un rendu intéressant et plus esthétique (même si ces critères restent arbitraire), il convient mieux d'utiliser les couches les plus élevé du réseau traitant la représentation du contenu. L'utilisation d'une couche plus basse aura pour effet de conserver une structure plus proche du contenu de l'image initial ,avec des détails , ce qui laissera moins de place à la texture que l'on cherche à synthétiser.

Enfin la méthode initialise la descente de gradient à partir d'un bruit blanc mais on peut tout à fait l'initialiser à partir d'une autre image notamment celles de style ou de textures.

3 Expérimentations

L'article datant de 2016 le matériel utilisé dans l'article est maintenant obsolète. Nous avons donc réaliser la totalité du code et des expériences sur Google Colab en utilisant les GPU Tesla V100 et Tesla A100 de Nvidia. L'implémentation est faite en python et j'ai utilisé les librairies Pytorch et PIL. Nous avons ensuite importé dans l'environnement d'exécution de Google Colab la librairie toolbox.py et chaque image, c'est à dire l'image de contenu et l'image de texture.

3.1 Présentation de l'algorithme

Pour reproduire les expériences décrites dans l'article, nous avons importé le modèle pré-entraîné VGG19, disponible via la bibliothèque Torchvision. Ce modèle se divise en deux composantes principales : une partie convolutionnelle et une partie discriminative, cette dernière étant principalement utilisée pour la reconnaissance d'objets. Dans notre cas nous avons uniquement besoin de la partie convolutionnelle du modèle. Pour ce faire, nous avons extrait les informations contenues dans les couches suivant chaque couche de pooling, conformément à la méthode décrite dans l'article. Cela nous permet d'accéder aux couches spécifiques nommées conv1_1, conv1_2, conv1_3, conv1_4, conv1_5, comme détaillé dans l'annexe A.

Algorithm 1: Optimisation de l'image de test

Input: Images : img_rand , img_tex , img_cont ; Nombre d'itérations : N_iter ; Taux d'apprentissage : lr ; Paramètres : α , β

Output: Image optimisée : img_test

- 1 Initialiser $img_test = img_rand$; Initialiser *optimizer* sur img_test lr ;
- 2 Initialiser $style_loss = 0$;
- 3 Définir les indices de couches de contenu et de style;
- 4 **for** $epoch = 0$ **to** $N_iter - 1$ **do**
- 5 Mettre à zéro les gradients dans l'optimiseur; Calculer les activations pour img_test , $style_image$ et $input_image$ avec le modèle VGG19 ;
- 6 Calculer $content_loss$ la perte sur le contenu; Réinitialiser $style_loss = 0$;
- 7 **for** chaque couche d'activation du réseau pour les textures **do**
- 8 Calculer les matrices de Gram pour img_test et $style_image$ à chaque couche; Ajouter à $style_loss$ la perte sur la matrice de Gram de chaque couche ;
- 9 Calculer $total_loss$ comme somme pondérée par α et β de $content_loss$ et $style_loss$;
- 10 Effectuer la backpropagation sur $total_loss$;
- 11 Faire une étape d'optimisation sur img_test ;

3.2 Expérimentations

Quelques constatations pratiques :

- Bien que les chercheurs de l'article recommandent l'utilisation de l'optimiseur BFGS, nous avons eu du mal à l'implémenter. C'est pourquoi nous avons opté pour l'optimiseur Adam, qui s'est révélé plus efficace dans ce contexte.
- L'algorithme fonctionne efficacement sur des images redimensionnées au format 512x512. Cependant, lorsqu'on dépasse cette résolution, le temps d'entraînement de l'algorithme augmente considérablement.

3.2.1 Reproduction des expérimentations de l'article

Pour pouvoir reproduire les expériences de l'article nous avons utilisé la même image que celle qui est présentée pour le contenu :

Pour les textures nous en avons choisi deux parmi les peintures qui sont présentées dans l'article.

Enfin pour que l'algorithme fonctionne correctement il faut que toutes les images soient de même taille, il est de plus très efficace pour des images de tailles 512 * 512.



FIGURE 1 – Neckarfront in Tübingen de Andreas Preafcke



FIGURE 2 – Le cri de Edvard Munch



FIGURE 3 – La nuit étoilée de Vincent Van Gogh

Ce qui nous donne comme résultat la figure 8 pour transfert de la texture du Cri. Après 100000 itérations (ce qui correspond à une durée de 1 heures 23 minutes et 29 secondes), cette image est obtenu avec un taux d'erreur de 0.25. Et la figure 9 pour un transfert de la texture du La nuit étoilée. Après 50000 itérations (ce qui correspond à une durée de 41 minutes et 30 secondes), cette image est obtenu avec un taux d'erreur de 0.79.

Comme précisé plus haut nous pouvons tout à fait initialiser l'image synthétisée non pas par un bruit blanc mais par l'image cible ou l'image de texture. Si on initialise l'image par la source de texture on obtient que l'image synthétisée ressemble beaucoup à cette dernière. Il semble même impossible de les différencier à l'oeil nu pour un nombre d'itération inférieur à 30000 ce qui représente 30 minutes d'entraînement environ. (Annexe B Figure 19,20,21). De plus pour une image qui initialisé à partir de l'image cible on obtient beaucoup plus facilement des résultat esthétique permettant ainsi de baisser le nombre d'itérations en voici exemples. (Annexe B, Figure 22).

Nous avons aussi décidé d'essayer avec d'autres images pour le contenu et d'autre source pour les textures dont voici des exemples. Ces exemples ont été réalisé avec un nombre d'itérations de 10000 pour $\alpha = 1$ et $\beta = 10^{-3}$



FIGURE 4 – Neckarfront
Tübingen redimensionnée



FIGURE 5 – Le cri redi-
mensionnée



FIGURE 6 – Image bruit
blanc initiale

FIGURE 7 – Images redimensionnées

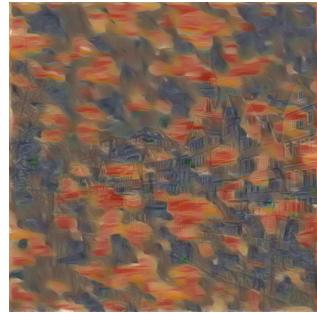


FIGURE 8 – $\alpha = 5$, $\beta = 10^4$,
 $N_{iter} = 100000$



FIGURE 9 – $\alpha = 5$, $\beta = 10^{-3}$,
 $N_{iter} = 50000$

Enfin nous pouvons aussi changer la resolution de l'image mais cela ralenti considérablement l'entraînement du réseau.



FIGURE 10 – La grande vague de Kanagawa de Katsuhika Hokusai



FIGURE 11 – Palais Impérial To-kyo e Barthélémy Metzlé



FIGURE 12 – Palais Impérial To-kyo avec La grande vague de Kanagawa

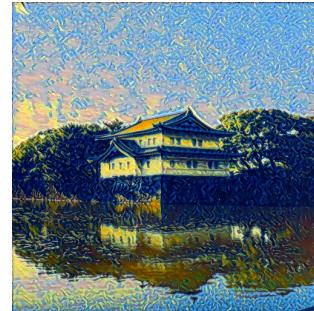


FIGURE 13 – Palais Impérial To-kyo avec Nuit Etoilée

4 Conclusion

En résumé, cet algorithme représente un excellent exemple de l'utilisation d'un réseau de neurones convolutifs (CNN). Toutefois, il soulève des interrogations sur les questions des droits d'auteur et la propriété intellectuelle. En mettant de côté ces considérations, d'un point de vue technique, il introduit une méthode nouvelle pour le transfert de textures. Actuellement, il est difficile d'imaginer des applications pratiques, car l'entraînement pour des images de résolution 512x512 reste très long, rendant son utilisation en temps réel inenvisageable. Il est néanmoins important de reconnaître les avancées techniques réalisées : alors que le temps d'entraînement était, à la parution de l'article d'environ une heure, il ne nécessite désormais que quelques dizaines de minutes. Des différences persistent entre les images obtenues dans l'article et celle obtenue expérimentalement, cela pourrait s'expliquer par les langages de programmation utilisés : l'article original a été rédigé pour une implémentation en LUA, tandis que cette version est en Python. Malgré ces limites, l'algorithme demeure fascinant et pourrait trouver des applications, notamment dans la création artistique.



FIGURE 14 – Time Square



FIGURE 16 – Time Square et Gravure 512*512 pixels



FIGURE 15 – La cascade de Dorrillouse de William Bartlett



FIGURE 17 – Time Square et Gravure 1024*1024 pixels

et le marketing personnalisé, où la personnalisation visuelle joue un rôle crucial.

A Illustration du Réseau

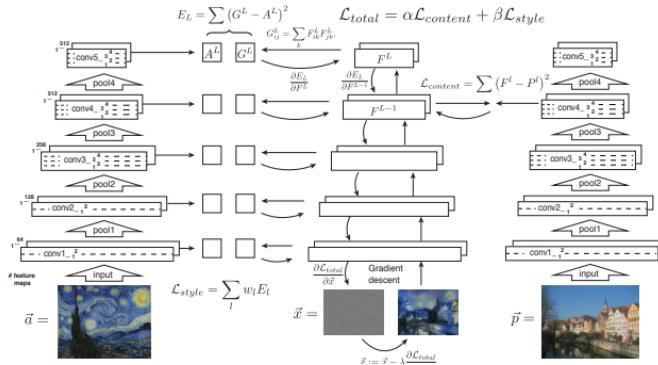


FIGURE 18 – Illustration des réseaux convolutifs extrait de Gatys Et Al. "Image Style Transfer Using Convolutional Neural Networks". 2016

B Quelques Images Supplémentaires



FIGURE 19 – Le cri après 30000 itérations ayant comme image originale l'image source de texture



FIGURE 20 – Image Source

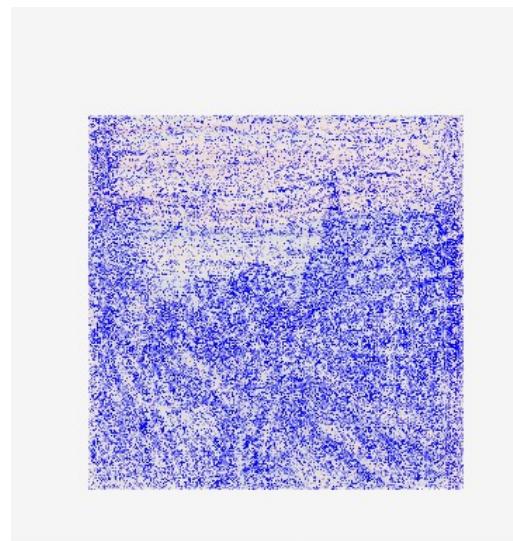


FIGURE 21 – Différence entre les valeurs des pixels des deux images précédentes

C Code

```
[81] import torch as tch
     import torchvision as tchvs
     from PIL import Image

     import numpy as np
     import matplotlib.pyplot as plt
     import imageio as imio
     from toolbox import disp

[73] device = 'cuda'

[102] img_2 = tch.tensor(imio.v2.imread('photo.jpg'))
     img = Image.open('photo.jpg')
     style = Image.open('style.jpg')
     #test = Image.open('image_test.png')
     img_height, img_width, img_depth = img_2.shape
     disp(img_2,(1,1),3)
```

FIGURE 22 –

```
▶
def gram_matrix(tensor):
    depth, height, width = tensor.shape
    return tensor.view(depth, height*width).mm(tensor.view(depth, height*width).t())

class VGG(tch.nn.Module):
    def __init__(self):
        super(VGG, self).__init__()

    self.chosen_features = [ '0', '5', '10', '19', '28']
    self.model = tchvs.models.vgg19(pretrained=True).features[:29]

    def forward(self, x):
        features = []

        for layer_num, layer in enumerate(self.model):
            x = layer(x)
            if str(layer_num) in self.chosen_features:
                features.append(x)
        return features

model = VGG().to(device).eval()
```

FIGURE 23 –

```

103] img_size = 512
    #on créer une fonction qui permet de faire en sorte que les images soient de même taille
    transform = tchvs.transforms.Compose([
        tchvs.transforms.Resize((img_size, img_size)),
        tchvs.transforms.ToTensor(),
        tchvs.transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    input_image = transform(img)
    style_image = transform(style)

    input_image_vs = transform(img)
    style_image_vs = transform(style)

    input_image_vs = tch.transpose(input_image_vs,0,2)
    style_image_vs = tch.transpose(style_image_vs,0,2)
    disp([input_image_vs,style_image_vs],[1,2],3)

    input_image = input_image.unsqueeze(0).to(device)
    style_image = style_image.unsqueeze(0).to(device)

    #img_test = tch.randn(input_image.shape, requires_grad = True, device = device)
    img_test = input_image.clone().requires_grad_(True)
    #img_test = style_image.clone().requires_grad_(True)

    #image_test = transform(test).unsqueeze(0).to(device)
    #img_test = image_test.clone().requires_grad_(True)

```

FIGURE 24 –

```

N_iter = 50000
lr = 1e-3
#content loss
alpha = 1
#style loss
beta = 1e-3

content_layer_index = 3
style_layer_indices = [0, 1, 2, 3, 4]

optimizer = tch.optim.Adam([img_test], lr=lr)
#loss = tch.nn.MSELoss(reduction = 'mean')
style_loss = 0

for epoch in range(N_iter):
    optimizer.zero_grad()
    test = model(img_test)
    style = model(style_image)
    content = model(input_image)

    content_loss = (1/2)*tch.mean((test[content_layer_index] - content[content_layer_index]) ** 2)
    style_loss = 0

```

FIGURE 25 –

```
for layer in style_layer_indices:  
    G = gram_matrix(test[layer])  
    A = gram_matrix(style[layer])  
    style_loss += (1/2)*tch.mean((G-A)**2)  
  
total_loss = alpha*content_loss + beta*style_loss  
total_loss.backward()  
optimizer.step()  
if epoch % 100 == 0:  
    print("epoch : ",epoch)  
    print(total_loss)  
  
if epoch% 10000 == 0:  
    tchvs.utils.save_image(img_test,"image_test.png")  
  
tchvs.utils.save_image(img_test, "image_test_final.png")
```

FIGURE 26 –