

RAPPORT DE PROJET : Artistic Style Transfer

A partir de la publication de ..., ce projet consiste à implémenter de cette méthode de transfert de style d'une image à une autre par l'application d'un apprentissage profond.



Yans KHOJA

2020/2021 (1er semestre)

CNAM – UE RCP209

[Préambule](#)

[Introduction](#)

[Détails de la méthode “Image Style Transfert”](#)

[Algorithme utilisé](#)

[Couches utilisées](#)

[Les calculs du transfert de style](#)

[Conclusion](#)

[Cette méthode équivaut à 3 réseaux de neurones convolutifs de type VGG19 tronqué :](#)

[Remarques](#)

[Implémentation](#)

[Configuration et librairies utilisées](#)

[Chargement des images](#)

[Le modèle de transfert de style](#)

[Les caractéristiques de représentation](#)

[Traitement du transfert de style](#)

[Résultats](#)

[Rendu final](#)

[Impact des coefficients de perte sur le rendu finale](#)

[Impact nombre d'époque](#)

[Similitudes entre image style et image content](#)

[Conclusion](#)

[Annexe](#)

Préambule

Dans le cadre de l'unité d'enseignement RCP209 enseigné au CNAM, ce projet porte sur l'implémentation et l'analyse de la méthode "Image Style Transfert" proposée par Leon A. Gatys, Alexeander S. Ecker et Matthias Bethge. Cette méthode consiste à effectuer un transfert de style d'une image artistique vers une image représentant un contenu concret à l'aide d'un algorithme de deep learning.

Ce sujet me semblait intéressé car je suis issue d'une formation créatrice et je souhaite acquérir des compétences en interaction humain-machine. La vision est un des sens prédominant de l'être humain et aujourd'hui un des sujets les plus avancés dans l'intelligence artificielle notamment avec le domaine de recherche de la vision par ordinateur.

J'ai réalisé ce projet individuellement par contrainte d'organisation et de temps. Il est joint à ce rapport le code commenté dont je détaillerai son implémentation au cours de ce rapport. Je fournirai en annexe un READ.txt afin d'expliquer comment exécuter le code.

Introduction

Appréhender la sémantique du contenu d'une image est un mécanisme complexe qui nécessite d'identifier des éléments représentatifs dans l'image et d'en interpréter sa "significations". L'étude du deep learning sur le traitement d'image numérique peut permettre de trouver d'identifier des analogies dans le fonctionnement du cerveau dans ce traitement de l'information.

En tant qu'être humain, nous jouons facilement avec la reconnaissance d'éléments familiers dans la nature (paysage, nuage...) et les expressions graphiques (croquis, schémas, peinture, sculpture...). Nous essayons, en permanence, de composer des formes soit d'une façon logique par la géométrie soit de façon subjective en les rattachant à des ressentis. Notre "objectif" est de leur donner du sens. Nous analysons d'un point de vue réductionnisme (couleur, contraste, forme, texture ...) et d'un point de vue holistique (environnement, abstractions, souvenir...). Nous jonglons facilement entre les détails et le globale pour comprendre une image.

Comprendre comment fonctionne ce mécanisme pourrait potentiellement nous permettre de rationaliser les processus mis en place dans notre cerveau et les transmettre à la machine.

Dans un premier temps, je présenterai ma compréhension de cette publication. La deuxième partie consistera à exposer mon implémentation de cette méthode. Pour finir, je présenterai les divers résultats obtenus.

Détails de la méthode “Image Style Transfert”

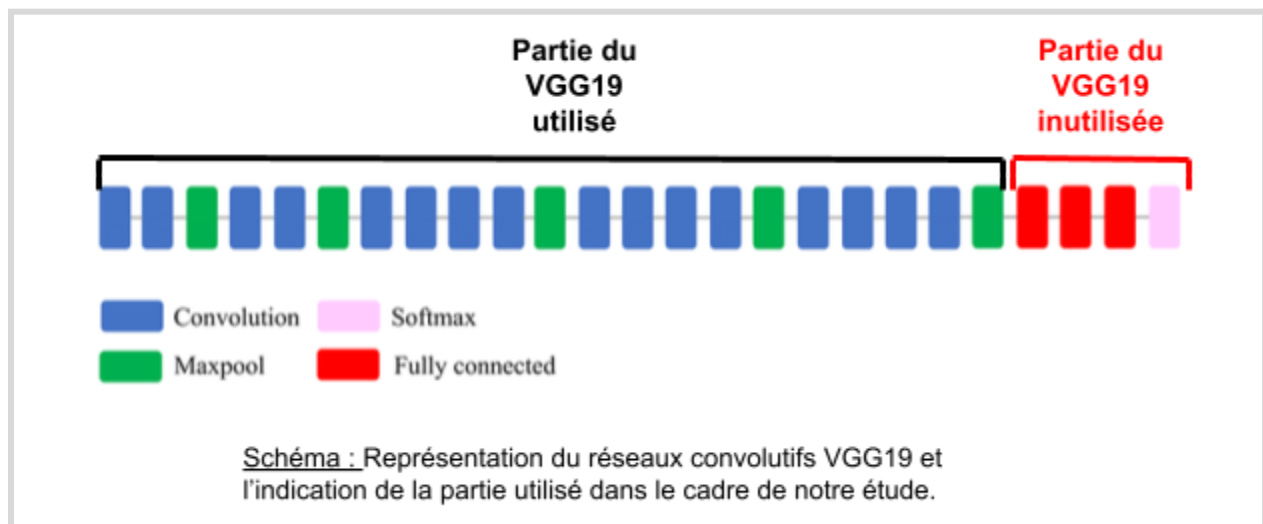
La méthode “Image Style Transfert” consiste à transférer le contenu stylistique d’une image vers le contenu d’une image réaliste grâce à l’utilisation d’un algorithme de deep learning pré-entraîné. Après récupération des poids des features maps de l’image “contenu” et de l’image “style” pour certaines couches (respectivement associé à l’image réaliste et l’image artistique), le transfert de style s’effectue par la rétro-propagation de l’erreur obtenu pour les images d’entrée sur l’image “transfert” (image obtenue).

Algorithme utilisé

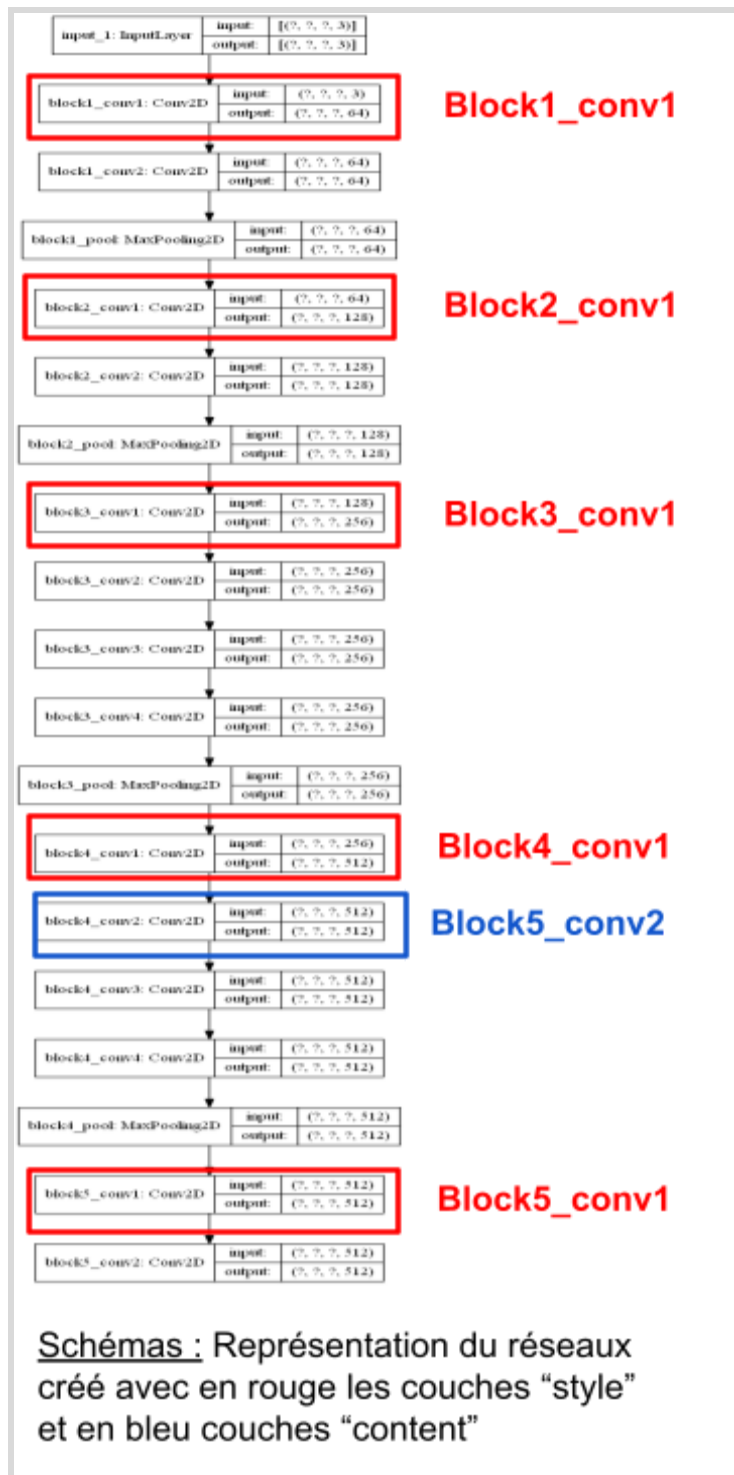
Le modèle d’apprentissage proposé est le réseau de neurones convolutifs VGG19. Uniquement une partie de l’architecture de ce modèle est utilisée, c’est-à-dire les 16 couches de convolution et les premières 5 couches de max pooling. Il sera tronqué par les 3 couches full connectées et la dernière couche softmax.

Ce qui s'apparentera à l'image ci-dessous.

Cet algorithme a été entraîné sur la base de données ‘imagenet’. Nous procéderons donc à un transfert d’apprentissage pour ce projet.



Couches utilisées



Comme mentionné précédemment, nous utiliserons les features maps de certaines couches afin d'effectuer le transfert d'informations des features représentation de l'image stylistique et le l'image réaliste vers une image de transfert.

Les features maps utilisés :

1. Pour extraire le contenu de l'image stylistique seront : 'block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'block5_conv1'.

2. Pour extraire le contenu de l'image réaliste sera 'block5_conv2'.

Un fois l'image jouée sur l'algorithme, il nous faudra récupérer les features representations au différents niveaux précisés au-dessus pour déterminer la perte entre l'image de transfert et la combinaison des informations des images style et contente pour ensuite rétro-propager l'erreur sur l'image transfert.

Les calculs du transfert de style

Pour propager le contenu d'une image à l'autre, nous passerons par une propagation de l'erreur entre les images style et content et l'image transfert . De ce fait, il nous faut calculer l'erreur entre l'image transfert et l'image réaliste sur la couche 'block4_conv2' et simultanément, l'image stylistique sur les couches précisées précédemment. Puis rétro-propager cette erreur sur l'ensemble de l'image transfert par l'intermédiaire d'un gradient.

Pour l'**image réaliste**, la formule est la suivante :

$$L_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2$$

Pour l'**image stylistique**, il nous faut au préalable calculer la matrice de gram avec la formule suivant sur les features maps obtenus des couches pour l'image stylistique et l'image généré :

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

puis calculer la perte avec la formule suivante :

$$L_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l$$

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

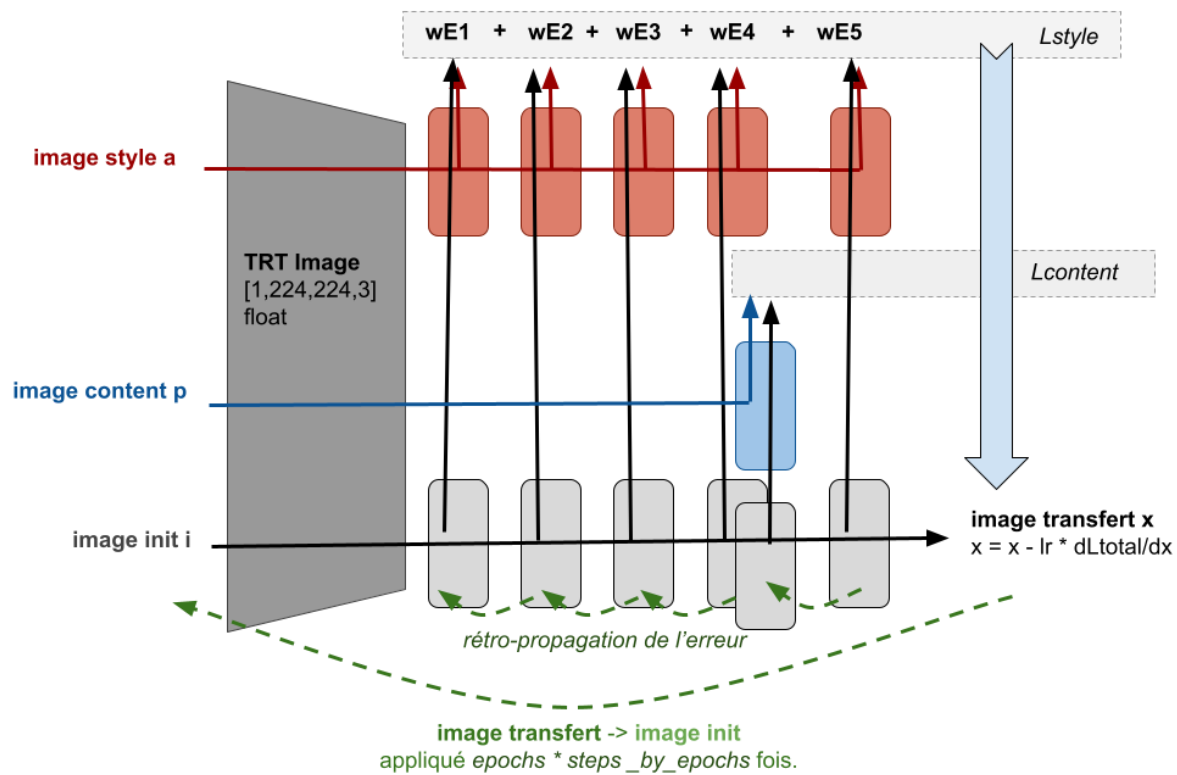
et enfin calculer la perte total :

$$L_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x})$$

Que l'on propagera sur l'image transfert de la manière suivante :

$$\vec{x} = \vec{x} - \lambda \frac{\partial L_{total}}{\partial \vec{x}}$$

Conclusion



Schémas : illustration du programme de transfert de style

Cette méthode équivaut à 3 réseaux de neurones convolutifs de type VGG19 tronqué :

- 2 modèles pré-entraînés dont on ne propage pas l'erreur : modèle “style” en rouge et modèle “content” en bleu dont les features representation, respectives, sont représentées en rouge et bleu.
- 1 modèle aussi pré-entraîné dont en revanche on va propager l'erreur obtenue en la comparant aux sorties estimées que les images style et content cibles que l'on va rejouer epochs * steps_by_epochs fois sur la nouvelles images transfert estimée.

Remarques

Les résultats obtenus varient en fonction du coefficient de perte sur l'image content α et du coefficient de perte sur l'image style β . Ces hyperparamètres évalués sur le rendu subjectif de l'image, nous montre que le choix du ratio $\frac{\alpha}{\beta}$ est primordial pour le résultat final.

L'optimiseur utilisé et l'initialisation sont aussi des éléments importants dans le rendu photoréaliste de l'image générée. Dans l'implémentation, nous utiliserons l'optimiseur Adam qui engendre un hyperparamètre le learning rate λ . Nous prendrons l'image content comme initialisation.

Nous verrons dans l'implémentation et le traitement d'image que deux autres hyperparamètres peuvent apporter de meilleurs résultats. Notamment, le nombre d'époque/étape par époque `(epochs et steps)` ainsi que la perte de variation totale `(total_variation_weight)` qui est un terme de régularisation permettant de réduire les artefacts produits sur les composantes hautes fréquence de l'image.

Voyons maintenant une proposition de l'implémentation de cette méthode de "Style transfert".

Implémentation

J'ai implémenté mon code en utilisant le langage python avec les librairies Keras pour l'utilisation et la manipulation de l'algorithme VGG19 et Tensorflow pour la manipulation des tensors.

Configuration et librairies utilisées

Pour réaliser cette implémentation, j'ai configuré mon poste de travail afin d'avoir accès à une meilleure puissance de calcul par l'usage de mon GPU. La configuration de que j'ai utilisé est la suivante :

- tensorflow-gpu = 2.1.0
- tensorflow-gpu-estimator = 2.1.0
- python = 3.7.7
- pillow = 7.2.0
- numpy = 1.17.0
- keras-applications = 1.0.8
- keras-base = 2.3.1
- keras-gpu = 2.3.1
- keras-preprocessing = 1.1.0
- cudnn = 7.6.4
- CUDA = 10.1

Mon code est divisé en deux fichiers :

- main.py contenant le programme principale
- fonctions.py contenant les divers fonctions utilisés dans le main

Dans les deux fichiers, on y retrouve les librairies suivantes :

```
# Bibliothèque d'apprentissage profond & manipulation de tenseur
import tensorflow as tf
import keras
from keras import *
from tensorflow.keras.applications import vgg19
from keras.applications.vgg19 import VGG19
from keras.models import Model
from keras.optimizers import SGD

# Bibliothèque classique
import numpy as np
from matplotlib import pyplot as plt
```

```
# Image et animation
import PIL.Image

# Recueil de fonctions utilisés dans le programme principale
from functions import *

# Bibliothèque système
import time
import sys
import os
```

Pour pouvoir accès à l'utilisation du GPU, j'ai dû activé l'option gpu après avoir configuré ma machine :

main.py

```
# Config GPU
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.compat.v1.Session(config=config)
```

Chargement des images

Ensuite, nous devons passer par le chargement des images et un traitement pour pouvoir les adapter au modèle VGG19 disponible dans Keras. Les éléments en entrée sur des listes des images de taillé [224, 224] sur trois niveau de couleur 'rgb' avec un typage float32.

En d'autres mots, l'objet d'entrée est un tensor [1,224,224,3] de type float32.

main.py

```
# Chargement des images
content_image =
tf.keras.preprocessing.image.load_img('...\data\Labrador.jpg',
color_mode='rgb')
style_image =
tf.keras.preprocessing.image.load_img('...\data\Kandinsky.jpg',
color_mode='rgb')

# Récupérer la taille des images d'origine
size_originale_content = np.size(content_image,0) ,
```

```

np.size(content_image,1)
size_originale_style = np.size(style_image,0) , np.size(style_image,1)
# Convertir les images doivent être contenu dans un array (1,224,224,3)
content_image = np.reshape(content_image,
                            (1,np.size(content_image,0),
                             np.size(content_image,1),
                             np.size(content_image,2)))
style_image = np.reshape(style_image,
                          (1, np.size(style_image,0),
                           np.size(style_image,1),
                           np.size(style_image,2)))

# Re-tailler les images
content_image = tf.image.resize(content_image, [224,224], method='nearest')
style_image = tf.image.resize(style_image, [224,224], method='nearest')

# Conversion en float 32
content_image = tf.image.convert_image_dtype(content_image, tf.float32)
style_image = tf.image.convert_image_dtype(style_image, tf.float32)

```

Le modèle de transfert de style

Une fois les images chargées dans le bon format. Nous allons configurer et créer le modèle. Pour le modèle, nous utiliserons le modèle VGG19 pré-entraîné sur la base d'image imagenet présent dans keras.applications.

D'après la publication, le transfert se réalise sur une partie du modèle VGG19. Précisément, on récupérera les features maps des couches mentionnées dans le chapitre précédent. Pour les récupérer et les énumérer puis créer notre modèle, on procédera de la manière suivante :

main.py

```

# Listes des "feature maps" à récupérer pour l'image content
content_layers = ['block4_conv2']

# Listes des "feature maps" à récupérer pour l'image style
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',

```

```

        'block5_conv1'
    ]

# Récupération de la taille des deux listes
num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
num_layers = (num_content_layers, num_style_layers)

# Création du modèle
custom_model = create_model(style_layers, content_layers)

```

functions.py

```

# Méthode permettant de créer le modèle convolutif basé sur un VGG19 et
# entraîné sur la base imagenet
# avec en sortie les features maps définis en argument de la méthode et
# nous empêcherons l'entraînement des paramètres.
def create_model(style_layers, content_layers):
    vgg = tf.keras.applications.VGG19(include_top=False,
    weights='imagenet')
    vgg.trainable = False
    style_outputs = [vgg.get_layer(name).output for name in style_layers]
    content_outputs = [vgg.get_layer(name).output for name in
    content_layers]
    outputs = style_outputs + content_outputs
    model = tf.keras.Model([vgg.input], outputs)
    model.summary()
    return model

```

ce qui nous donne le résultat suivant :

Model: "model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, None, None, 3)]	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928

block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295168
block3_conv2 (Conv2D)	(None, None, None, 256)	590080
block3_conv3 (Conv2D)	(None, None, None, 256)	590080
block3_conv4 (Conv2D)	(None, None, None, 256)	590080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1180160
block4_conv2 (Conv2D)	(None, None, None, 512)	2359808
block4_conv3 (Conv2D)	(None, None, None, 512)	2359808
block4_conv4 (Conv2D)	(None, None, None, 512)	2359808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2359808
block5_conv2 (Conv2D)	(None, None, None, 512)	2359808
=====		
Total params:		15,304,768
Trainable params:		0
Non-trainable params:		15,304,768

Les caractéristiques de représentation

Ensuite, nous récupérerons les features représentation calculées sur les deux

images “content” et “style”. Pour ce faire, j’ai utilisé 3 méthodes imbriquées.

La première “chapeau” est *get_outputs* qui retourne un dictionnaire contenant les features représentation pour les features maps précédemment citées dans le format attendu c’est-à-dire dans la forme de Gram pour les couches de l’image “style” et dans la forme nominale pour l’image “content”.

Cette méthode utilise une sous-méthode qui récupère les features représentations après avoir joué l’image dans l’algorithme, *get_feature_representation*. On y retrouve aussi une méthode dédiée au calcul de la matrice de Gram, *gram_matrix*.

main.py

```
# Stockage des features representation pour chacune des images
style_outputs_target = dict()
content_outputs_target = dict()
style_outputs_target = get_outputs( style_layers,content_layers,
                                     custom_model,tf.constant(style_image),
                                     num_layers)
content_outputs_target = get_outputs( style_layers,content_layers,
                                     custom_model,tf.constant(content_image),
                                     num_layers)

# Stockage dans un unique dictionnaire
outputs_target = dict()
outputs_target['style_outputs'] = style_outputs_target
outputs_target['content_outputs'] = content_outputs_target
```

functions.py

```
# Méthode permettant de récupérer les features representations final de
l'image
def get_outputs(style_layers, content_layers, custom_model, image,
num_layers):

    num_content_layers , num_style_layers = num_layers

    style_features, content_features = \
        get_feature_representations( custom_model,
                                     image,
                                     num_style_layers)

    # Conversion liste en tenseur
```

```

style_features = \
    [tf.convert_to_tensor(style_feature) for style_feature in
style_features]
content_features = \
    [tf.convert_to_tensor(content_feature) for content_feature in
content_features]

# Calculer la matrice de GRAM pour chacune des features representation
des couches styles
gram_style_features = \
    [gram_matrix(style_feature) for style_feature in style_features]

# Stockage dans un unique dictionnaire
content_dict = {content_name:value
                 for content_name, value
                 in zip(content_layers, content_features)}

style_dict = {style_name:value
              for style_name, value
              in zip(style_layers, gram_style_features)}

outputs = {'content':content_dict, 'style':style_dict}

return outputs

```

J'ai choisi de calculer la matrice de Gram indépendamment ainsi que la récupération des features représentation dans des fonctions indépendantes afin d'une part d'alléger la méthode `get_outputs` et pour une meilleure lecture de code.

functions.py

```

# Méthode permettant de récupérer les features représentations du modèle
pour l'image
def get_feature_representations(model, image, nbr_layers):
    # calculer les features de l'image
    content_outputs = model(image)

    # Récupérer les features representations content et style à partir de
notre modèle à partir nos layers intermédiaires spécifiques
    # style_features -> Nous voulons récupérer les features
representation des layers 'block1_conv1', 'block2_conv1', 'block3_conv1',
'block4_conv1', 'block5_conv1' pour l'image style
    # style_features -> Nous voulons récupérer les features

```



```

representation des layers 'block4_conv2' pour l'image content
    style_features = [style_layer[0] for style_layer in
content_outputs[:nbr_layers]]
    content_features = [content_layer[0] for content_layer in
content_outputs[nbr_layers:]]
    return style_features, content_features

```

functions.py

```

# Méthode calculant la matrice de gram
def gram_matrix(input_tensor):
    channels = int(input_tensor.shape[-1])
    a = tf.reshape(input_tensor, [-1, channels])
    n = tf.shape(a)[0]
    gram = tf.matmul(a, a, transpose_a=True)
    return gram / tf.cast(n, tf.float32)

```

Traitement du transfert de style

Pour le transfert de style, nous avons besoin de partir d'une image initiale (ici, j'ai utilisé l'image "content"). Une fois l'image initialisée, Nous devons définir l'optimiseur (ici nous avons choisi l'optimiseur Adam). Cet optimiseur est crucial dans le temps de traitement et le rendu photoréaliste.

J'ai aussi défini des coefficient de poids de perte pour l'image "style" et l'image "content" comme défini dans la partie théorique précédente.

main.py

```

# Initialisation de l'image à styliser
init_image = tf.Variable(content_image)

# Usage d'un adam optimizer
opt = tf.optimizers.Adam(learning_rate=0.02)

# Stockage des poids des loss content et loss style
content_weight = 1e4
style_weight= 1e-2
loss_weights = (style_weight, content_weight)

# Variable de la perte de variation totale

```

```
total_variation_weight = 30
```

Pour réaliser l'entraînement de l'algorithme sur les jeux de données, j'ai procédé par plusieurs méthodes imbriquées calculées sur des époques (epochs) et étapes (steps).

main.py

```
# ===== Debut du traitement =====
print('===== Debut du traitement =====')
start = time.time()

epochs = 50
steps_per_epoch = 100
step = 0
fname = "image_generée_iteration_%d.jpg" % 0
tensor_to_image(init_image).save("../data/output\" + fname)
for n in range(epochs):
    for m in range(steps_per_epoch):
        step += 1
        train( opt, init_image, style_layers, content_layers,
                custom_model, num_layers, outputs_target, loss_weights,
total_variation_weight)
        init_image = init_image
        print(".",end='')
        print("Train step: {}".format(step))
        n = n + 1
        fname = "image_generée_iteration_%d.jpg" % n
        tensor_to_image(init_image).save("../data/output\" + fname)

end = time.time()
print("Total time: {:.1f}".format(end-start))
print('===== Fin du traitement =====')
```

La méthode *train* nous permet d'évaluer la perte totale obtenue sur l'image "style" et l'image "content" en fonction de l'image initialisée (les formules sont présentées dans la chapitre théorique). Comparativement au feature representation obtenu respectivement sur l'image "style" et l'image "content", nous les évaluons par rapport à la "prédiction" effectuée sur l'image initialisée. Le différentiel obtenu sera retropropogé grâce au calcul de perte total avec l'optimiseur sur l'ensemble de l'image. Ces étapes sont réalisées dans les instructions suivantes :

```
grad = tape.gradient(loss, image_init)
```

```
opt.apply_gradients([(grad, image_init)])
```

functions.py

```
# Calculer les gradients par rapport à l'image d'entrée
def train( opt, image_init, style_layers,
          content_layers, custom_model,
          num_layers, target, loss_weights, total_variation_weight =
0):

    with tf.GradientTape() as tape:
        outputs = get_outputs( style_layers,
                                content_layers,
                                custom_model,
                                image_init,
                                num_layers)

        loss = style_content_loss(outputs, target, loss_weights, num_layers)
        loss += total_variation_weight * tf.image.total_variation(image_init)

    grad = tape.gradient(loss, image_init)

    opt.apply_gradients([(grad, image_init)])
    image_init.assign(clip_0_1(image_init))

    return image_init
```

Concernant le calcul de perte totale, nous devons récupérer les différentes features représentations établies sur les images “style” et “content” puis les évaluons par rapport aux résultats obtenues sur l’image initialisées afin de calculer les pertes théoriquement vu précédemment.

functions.py

```
# Calculer la perte totale
def style_content_loss(outputs, target, loss_weights, num_layers):

    # Récupérer les outputs de l'image d'initialisation
    style_outputs = dict(outputs['style'].items())
    content_outputs = dict(outputs['content'].items())
    style_targets = dict(target['style_outputs'].items())
    style_targets_outputs = dict(style_targets['style'].items())
```

```

content_targets = dict(target['content_outputs'].items())
content_targets_outputs = dict(content_targets['content'].items())

style_weight, content_weight = loss_weights
num_style_layers, num_content_layers = num_layers

# Calculer la perte style
style_loss = 0
for name in style_outputs.keys():
    sqrt = tf.square(style_outputs[name]-style_targets_outputs[name])
    sca = tf.reduce_sum(sqrt)
    style_loss += 1./5. * sca
style_loss *= (4. * (3 ** 2) * ((224*224) ** 2))
style_loss *= style_weight / float(num_style_layers)

# Calculer la perte content
content_loss = 0
for name in content_outputs.keys():
    sqrt =
tf.square(content_outputs[name]-content_targets_outputs[name])
    sca = tf.reduce_sum(sqrt)
    content_loss += sca)
content_loss *= 1./2.
content_loss *= content_weight / float(num_content_layers)

loss = style_loss + content_loss

return loss

```

Une fois la perte évaluée et rétro propager sur l'image, nous avons obtenu un tensor ayant des données que nous devons normalisée entre les bornes 0 et 1. Une méthode *clip_0_1* est donc nécessaire. Elle assigne les valeurs normalisées de l'image générée entre 0 et 1 qui correspondent à 0 -> 255 dans l'encodage 'rgb'.

functions.py

```

def clip_0_1(image):
    return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

```


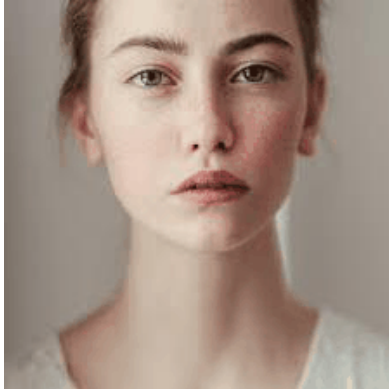

Résultats

Rendu final

Suite à plusieurs tests pour déterminer les hyperparamètres, j'ai remarqué qu'ils dépendent de la similarité des éléments des images content et style.




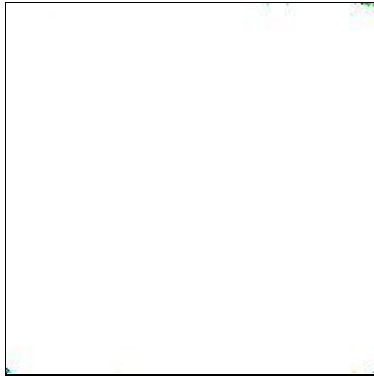
Les hyperparamètres sont les suivantes :

- coefficient de perte sur l'image content α
- coefficient de perte sur l'image style β
- learning rate λ
- epochs et steps
- Perte de variation totale (total_variation_weight)


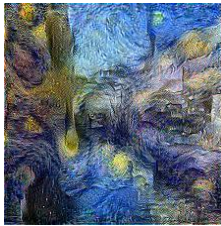
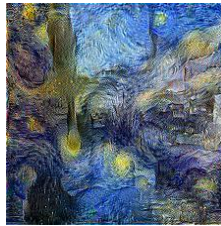


		
$\alpha = 1e15$ $\beta = 1e-5$ $\lambda = 0.027$ epochs : 15 (dont 100 steps) total_variation_weight = 50	$\alpha = 1e6$ $\beta = 1e-6$ $\lambda = 0.02$ epochs : 50 (dont 100 steps) total_variation_weight = 30	$\alpha = 1e6$ $\beta = 1e-4$ $\lambda = 0.025$ epochs : 10 (dont 100 steps) total_variation_weight = 10

Impact des coefficients de perte sur le rendu finale


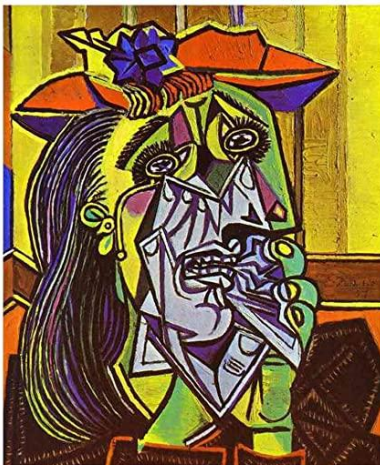



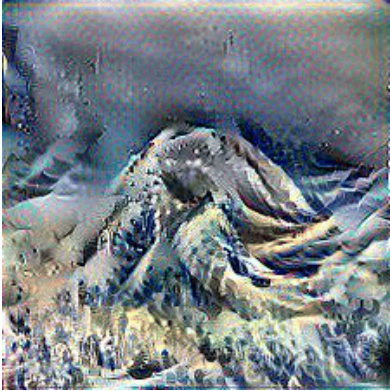
On remarque qu'il y a des seuils où les images ne sont plus exploitées.

Avec un $\beta \leq 1e-3$ l'image transfert est quasi blanche.		
Avec un $\alpha \geq 1e19$, l'image transfert est quasi blanche.		

Impact nombre d'époque

				
epochs = 10	epochs = 20	epochs = 30	epochs = 40	epochs = 50
On remarque que le nombre d'itération de l'algorithme détériore le contenu d'image au profil du style de la peinture.				

Similitudes entre image style et image content

Image content	Image style	Image transfert
		
		
<p>On remarque que la similitude entre les éléments des deux images est aussi un facteur du bon rendu de l'image transfert. Lorsque les concepts sont très éloignés, l'image transfert n'a plus de sens.</p>		

Conclusion

Les hyperparamètres sont cruciaux dans le rendu photoréaliste de l'image transfert. Il est vrai que les résultats sont assez subjectifs et aléatoires car nous devons déterminer les paramètres optimaux de façon empirique. De plus, les résultats sont très variables par rapport à la sensibilité du spectateur.

Tout de même, la méthode "Image Style Transfert" présentée dans la documentation de Leon A. Gatys, Alexander S. Ecker et Matthias Bethge est très intéressante car elle m'a permis de comprendre un cas d'usage de deep learning dans le domaine de la vision par ordinateur et d'appréhender comment des concepts abstraits d'un style artistique peuvent être extraits puis transposés sur une autre image réaliste sans en détériorer son contenu.

Annexe

Guide d'installation et utilisation du code

SPEC

Operating System	Windows 10 Famille
GPU	GeForce GTX 1650 Ti
CUDA Version	10.1

Step 1 : Installation CUDA

<https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html>

lib : cuDNN = 7.6.4

Step 2 : Installation Anaconda

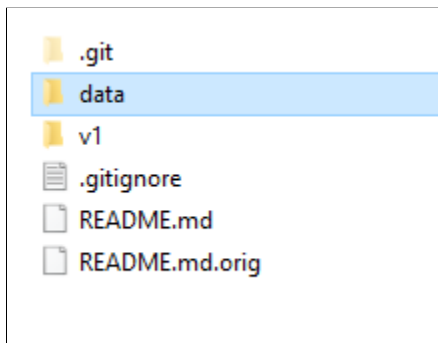
<https://docs.anaconda.com/anaconda/install/>

Step 3 : Installation environnement virtuel

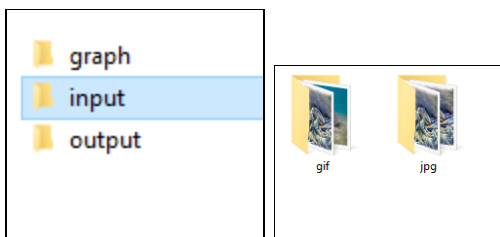
Jouer les commandes suivantes en séquence dans Anaconda Prompt :

```
conda create -n tf-gpu tensorflow-gpu = 2.1.0
conda activate tf-gpu
conda install tensorflow-gpu-estimator = 2.1.0
conda install python = 3.7.7
conda install pillow = 7.2.0
conda install numpy = 1.17.0
conda install keras-applications = 1.0.8
conda install keras-base = 2.3.1
conda install keras-gpu = 2.3.1
conda install keras-preprocessing = 1.1.0
```

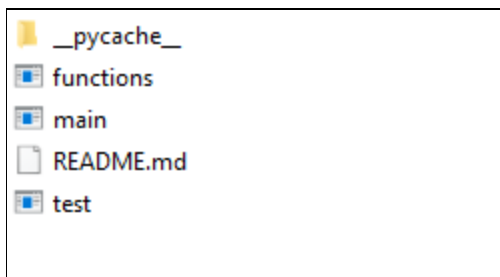
Step 4 : Répertoire et manipulation



- le répertoire *data* contient les images en entrée et les images générée (+graph)
- le répertoire *v1* contient le programme



- le répertoire *graph* contient le graph du modèle
- le répertoire *input* contient les images style et content
- Le répertoire *output* contient les images transfert en .jpg et .gif



- le fichier *main.py* est le programme principale
- le fichier *functions.py* contient les fonctions utilisées par le programmes principales
- *Ne pas prendre en compte le fichier test.py*

Comment lancer le programme ?

1. Ajouter vos images content et style dans le dossier input
2. Modifier le répertoire dans le fichier main.py

```

# Config GPU
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.compat.v1.Session(config=config)

# Chargement des images
content_image = tf.keras.preprocessing.image.load_img('..\data\input\Turtle.jpg', color_mode='rgb')
style_image = tf.keras.preprocessing.image.load_img('..\data\input\Kanagawa.jpg', color_mode='rgb')

# Récupérer la taille des images d'origine
size_originale_content = np.size(content_image,0) , np.size(content_image,1)
size_originale_style = np.size(style_image,0) , np.size(style_image,1)

```

3. > ouvrir un console de commande
 > se placer dans le répertoire du code
 > entrer `python main.py`