Neural Style Transfer

최석재 *lingua@naver.com*

뉴럴 스타일 트랜스퍼

- 뉴럴 스타일 트랜스퍼는 원본 이미지의 기본 형상을 보존하면서
- 참조 이미지의 스타일을 원본 이미지에 적용하는 기술

- 한 이미지의 스타일을 다른 이미지에 적용하는 것으로서,
- 특히 원본 이미지에 예술 작품의 스타일을 입히려고 할 때 많이 사용된다

- 원본 이미지에서는 이미지의 구조와 형태를,
- 참조 이미지에서는 색상, 질감과 같은 스타일적 특성(예술적 특성)을 추출한다

구글 드라이브 연결

- from google.colab import drive
- drive.mount('/content/gdrive')

경로 설정

- from keras.models import Model
- from IPython.display import display
- from IPython.display import Image as _Imgdis
- from keras.preprocessing.image import array_to_img, img_to_array, load_img

- image_path = '/content/gdrive/MyDrive/pytest_img/opencv/' # 기본 경로
- save_path = '/content/gdrive/MyDrive/pytest_img/_generated_images/' # 기본 저장 경로

원본 이미지와 참조 이미지 경로 설정
- base_image_path = image_path+'seoul.png'
- style_reference_image_path = image_path+'starnight.png'

저장 경로 생성

기본 저장 경로 밑에 neural_style 이라는 폴더를 만든다

- if not os.path.exists(os.path.join(save_path, "neural_style/")):
  os.makedirs(os.path.join(os.path.join(save_path, "neural_style/")))

이미지 사이즈 확인

원본 이미지와 참조 이미지의 사이즈가 비슷해야 잘 된다
두 이미지를 같은 크기로 하였을 때 출력에 무리가 없는지 확인한다

- img_height = 400 # 이미지의 높이
- img_width = 600 # 이미지의 너비
- display(_Imgdis(filename=base_image_path, height=img_height, width=img_width))
- display(_Imgdis(filename=style_reference_image_path, height=img_height, width=img_width))

두 이미지의 본래 사이즈는 각각
(1440, 810) (1151, 889)

이미지 전처리 함수

- 이미지를 읽고, 전처리하는 함수를 정의한다

- import numpy as np
- import tensorflow as tf

- def preprocess_image(image_path):

```
    img = load_img(image_path, target_size=(img_height, img_width))        # 같은 사이즈로 맞춘다
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)                                      # 맨 앞에 배치 차원을 추가한다
    img = tf.keras.applications.vgg19.preprocess_input(img)

    return img
```

VGG19 모델에 맞게 입력 이미지 데이터를 전처리한다
이미지 데이터의 픽셀값을 스케일링하여 모델이 학습된 방식에 맞도록 조정한다

이미지 복원 함수

- VGG19 전처리 과정을 되돌려 넘파이 배열을 이미지로 변환하는 함수 정의

- def deprocess_image(img):

```
    img = img.reshape((img_height, img_width, 3))    # 이미지를 위에서 정의한 사이즈로 변환
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.68
    img = img[:, :, ::-1]                            # 채널의 순서를 VGG19가 학습된 RGB → BGR로 변경한다
    img = np.clip(img, 0, 255).astype("uint8")       # 값을 0~255 사이로 제한한다

    return img
```

VGG19.preprocess_input()에서는 전처리를 수행하면서 RGB 채널에서 이 세 값을 빼주므로
여기서는 이 세 값을 다시 더해주어 원래의 색상으로 복원한다
이 세 값은 ImageNet의 평균 픽셀 값

VGG19 모델 로드

- 완전연결층을 제외한 모델을 로드한다
- from tensorflow.keras.applications.vgg19 import VGG19
- model = VGG19(weights="imagenet", include_top=False)

모델 레이어 결과 가져오기

모델의 레이어들을 그 출력 결과와 함께 가져온다

- outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])

모델에 입력을 넣으면 모델의 모든 레이어 출력을 포함하는 딕셔너리를 반환하는 모델 생성
일반적으로 Functional API에서 outputs에는 최종 출력층을 넣지만,
여기에서는 필요한 레이어의 출력값을 쉽게 가져오도록 모든 레이어의 출력을 반환하게 한다

- feature_extractor = Model(inputs=model.inputs, outputs=outputs_dict)

1. in model.layers에서 모델의 모든 레이어를 가져온다
2. (layer.name, layer.output) for layer에서 각 레이어의 이름과 해당 레이어의 출력 결과를 튜플로 묶는다
3. dict()에서 레이어의 이름과 출력을 dictionary 형태로 변환한다

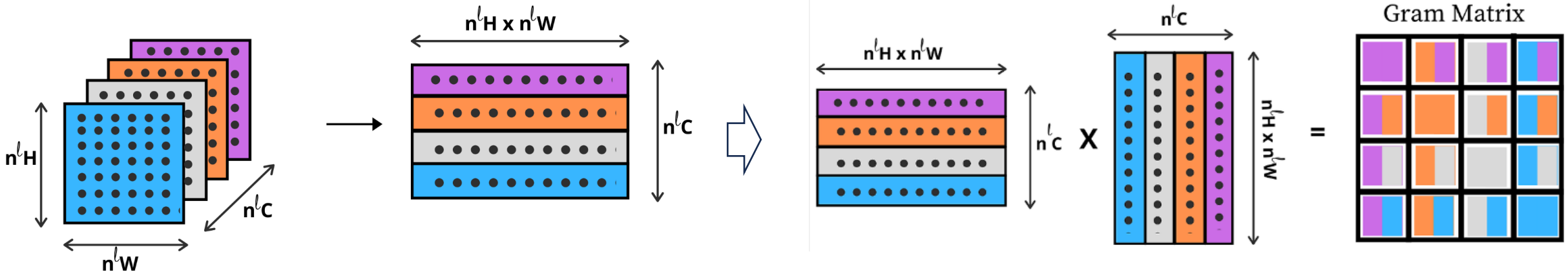
콘텐츠 손실함수

콘텐츠 즉, 원본 이미지의 기본 형상에 대한 손실함수 정의
base_img는 원본 이미지를,
combination_img는 스타일이 적용된 후의 조합 이미지로 둘의 차이를 손실값으로 정의한다
변환된 값이 원본 이미지와 지나치게 차이가 나지 않도록 조절하는 데 사용된다
너무 같아도 안되므로, 스타일 손실과의 사이에서 적절한 밸런스를 찾는다

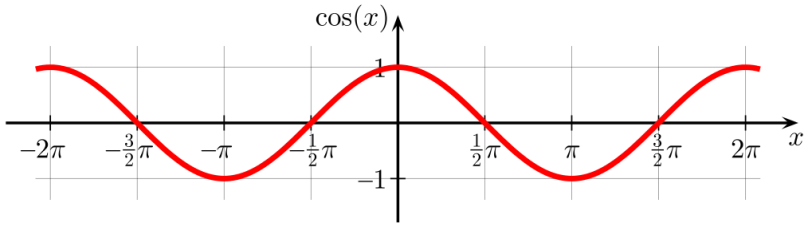
- def content_loss(base_img, combination_img):
    return tf.reduce_sum(tf.square(combination_img - base_img))

그람 행렬

- 그람 행렬은 벡터 사이의 모든 가능한 내적을 구하는 행렬
- 그람 행렬을 사용하면 각 채널의 상호 관계를 포착할 수 있다
- 각 채널은 이미지의 다른 측면을 포착하는데, 이들의 상호 관계를 포착하려는 것이 목적
- 각 채널을 독립적인 벡터로 처리하고, 이 벡터들 사이의 내적을 구해 그람 행렬을 만든다
- 벡터 사이의 내적은 두 벡터의 유사성을 표현한다



내적과 유사도

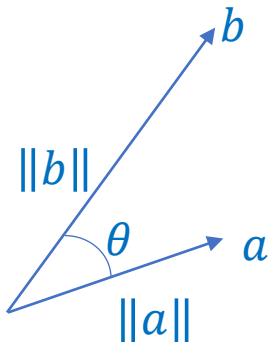


- 내적은 기하와 연산의 두 가지 방법으로 구할 수 있는데,
- 기하학적으로는 cosine 값이 들어가며, 이 값이 클수록 유사도가 높게 된다 (작은 θ)
- 따라서 연산을 통해 구한 두 벡터의 내적도 클수록 유사도가 높다고 할 수 있다

$$\mathrm{a} \cdot \mathrm{b} = a_1 b_1 + a_2 b_2 + \cdots = \sum_{i=1}^{n} a_i b_i$$

$$\mathrm{a} \cdot \mathrm{b} = \|a\|\|b\| \cos\theta$$

$$cosine\ similarity(a, b) = \frac{a \cdot b}{\|a\|\|b\|}$$



두 벡터가 평행하여 각도가 0에 가까워질수록 코사인 값은 1에 가까워지며,
각도가 커지면 코사인 값은 1 이하의 값이 되어 내적 값은 작아진다
코사인 값은 -1에서 1 사이를 가지므로, 두 벡터가 유사할수록 내적 값이 커진다

그람 행렬 함수

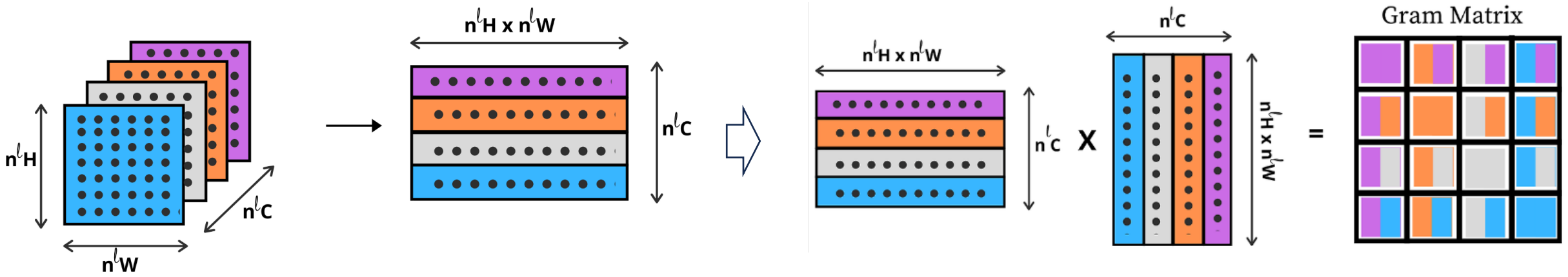
- def gram_matrix(x):

  x = tf.transpose(x, (2, 0, 1)) # (높이, 너비, 채널) → (채널, 높이, 너비) 순서로 바꾼다

  features = tf.reshape(x, (tf.shape(x)[0], -1)) # 각 채널을 행으로, 나머지는 -1로 (높이x너비) 열로 삼아 2D 행렬 구성

  gram = tf.matmul(features, tf.transpose(features)) # 만들어진 2D features를 그 전치와 행렬곱하여 그람 행렬을 만든다

  return gram



스타일 손실함수

스타일, 즉 참조 이미지가 갖는 스타일에 대한 손실함수를 정의
참조 이미지와 스타일이 적용된 후의 이미지가 얼마나 차이나는지를 손실값으로 정의한다
스타일은 개별 픽셀의 차이보다는 채널 사이의 상호관계 차이로 보는 것이 더 타당하기 때문

- def style_loss(style_img, combination_img):

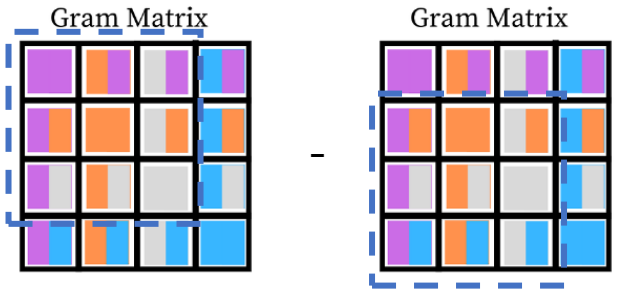
```
S = gram_matrix(style_img)              # 스타일 이미지의 그람 행렬 계산
C = gram_matrix(combination_img)        # 스타일이 적용된 후의 이미지인 combination_img의 그람 행렬 계산
channels = 3
size = img_height * img_width

return tf.reduce_sum(tf.square(S-C)) / (4.0 * (channels ** 2) * (size ** 2))
```

S – C 를 통하여 두 이미지의 스타일적 특징 차이를 구할 수 있다
이 차이는 각 채널들이 어떻게 상호작용하는지에 대한 정보의 차이. 이 차이를 제곱하여(tf.square()) 강도를 강조한다
스타일의 유사성이 떨어질수록 손실값이 커지는 것으로 본다
분모 부분은 상수 4.0 * (채널수^2) * (이미지의 크기(높이x너비)^2)) 로서 정규화하기 위한 값

총 변동 손실 함수

- 이미지에서 근접한 부분의 변동을 최소화
- 이미지를 매끄럽게 하는 효과를 갖는다



이 두 행렬을 빼게 되면
세로 방향으로 인접한 픽셀과의 차이를
얻을 수 있게 된다

- def total_variation_loss(x):

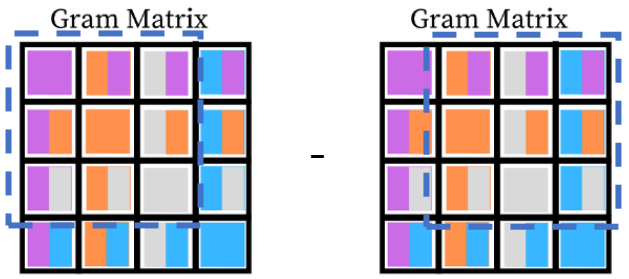
마지막 행을 제외한 모든 행, 마지막 열을 제외한 모든 열 - 첫 행을 제외한 모든 행, 마지막 열을 제외한 모든 열

```
a = tf.square(x[:, : img_height - 1, : img_width - 1, :] - x[:, 1:, : img_width - 1, :])
```

마지막 행을 제외한 모든 행, 마지막 열을 제외한 모든 열 - 마지막 행을 제외한 모든 행, 첫 열을 제외한 모든 열

```
b = tf.square(x[:, : img_height - 1, : img_width - 1, :] - x[:, : img_height - 1, 1:, :])

return tf.reduce_sum(tf.pow(a+b, 1.25))
```



이 두 행렬을 빼게 되면
가로 방향으로 인접한 픽셀과의 차이를
얻을 수 있게 된다

총 변동량을 구하기 위하여 세로 방향 차이의 제곱(square)과 가로 방향 차이의 제곱을 더한 뒤,
차이를 적절히 강조하기 위하여 1.25 제곱을 한다
모든 차원에 걸쳐 수행하는 tf.reduce_sum()으로 배치 차원의 모든 이미지가 한 번에 계산된다

파라미터 설정

스타일을 추출하기 위해 사용할 네트워크 층

- style_layer_names = ["block1_conv1", "block2_conv1", "block3_conv1", "block4_conv1", "block5_conv1"]

콘텐츠(원본 이미지)에서 기본 형상 특징을 추출하기 위해 사용할 네트워크 층

- content_layer_name = "block5_conv2" # 콘텐츠 손실에 사용할 층

스타일을 얼마나 원본 이미지에 반영할 것인지를 결정하는 가중치 설정

- total_variation_weight = 1e-6 # 총 변동 손실의 기여 가중치 (이미지를 얼마나 매끄럽게 표현할 것인가)
- style_weight = 1e-6 # 스타일 손실의 기여 가중치 (전체 이미지에서 반영할 스타일의 비중)
- content_weight = 2.5e-8 # 콘텐츠 손실의 기여 가중치 (전체 이미지에서 반영할 콘텐츠의 비중)

최종 손실 함수 (1/2)

- 조합된 이미지, 원본 이미지, 참조 이미지를 입력으로 받아 그들의 차이를 토대로 최종 손실을 계산한다

- def compute_loss(combination_image, base_image, style_reference_image):

```
    # 원본 이미지, 참조 이미지, 조합된 이미지를 배치 차원으로 결합한다 (3, 높이, 너비, 채널)
    input_tensor = tf.concat([base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)          # 입력 데이터를 모델을 사용하여 처리한 결과로부터 특징을 추출
    loss = tf.zeros(shape=())                           # 손실을 0으로 초기화
    layer_features = features[content_layer_name]       # 콘텐츠 손실에 사용할 층의 특징을 추출
    base_image_features = layer_features[0, :, :, :]    # 원본 이미지의 특징을 추출
    combination_features = layer_features[2, :, :, :]   # 조합 이미지의 특징을 추출
    loss += content_weight * content_loss(base_image_features, combination_features)
        # 콘텐츠 손실을 계산하고, 이를 콘텐츠 손실 가중치와 곱한 뒤, 전체 손실에 추가한다
```

최종 손실 함수 (2/2)

```python
def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat([base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)
    loss = tf.zeros(shape=())           # 손실을 0으로 초기화
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(base_image_features, combination_features)

    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        style_loss_value = style_loss(style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * style_loss_value

    loss += total_variation_weight * total_variation_loss(combination_image)
    return loss
```

```
for layer_name in style_layer_names:            # 스타일 추출에 사용할 층
    layer_features = features[layer_name]       # 스타일 손실에 사용할 층의 특징을 추출
    style_reference_features = layer_features[1, :, :, :]       # 스타일 이미지의 특징을 추출
    combination_features = layer_features[2, :, :, :]           # 조합 이미지의 특징을 추출
    style_loss_value = style_loss(style_reference_features, combination_features)       # 각 층의 스타일 손실 계산
    loss += (style_weight / len(style_layer_names)) * style_loss_value
            # 스타일 손실을 (스타일 손실 가중치/층 수)와 곱한 뒤, 전체 손실에 추가한다 (층의 모든 계산 결과를 누적)

loss += total_variation_weight * total_variation_loss(combination_image)
return loss     # 이미지의 매끄러움에 관계되는 조합 이미지의 총 변동 손실을 계산하고, 이를 총 변동 손실 가중치와 곱한 뒤, 전체 손실에 추가한다
```

손실값 정리

- 콘텐츠 손실 content_loss()
  - 콘텐츠 이미지와 조합 이미지 사이의 구조적 유사성을 측정
  - 이 손실값이 낮다면, 조합 이미지가 원본 이미지의 주요 특징과 구조를 잘 보존한다는 것을 의미

- 스타일 손실 style_loss()
  - 스타일 이미지와 조합 이미지 사이의 스타일적 유사성을 측정
  - 이 손실값이 낮다면, 조합 이미지가 참조 이미지의 스타일을 잘 반영한다는 것을 의미

- 총 변동 손실 total_variation_loss()
  - 조합 이미지의 시각적 부드러움을 측정
  - 이 손실값이 낮다면, 조합 이미지가 매끄럽고 자연스럽다는 것을 의미

- 최종 손실 compute_loss()
  - 콘텐츠 손실, 스타일 손실, 총변동 손실을 종합적으로 측정
  - 이 손실값이 낮다면, 위 3개의 손실값이 모두 낮다는 것을 의미
  - 콘텐츠가 보존되면서 스타일이 적절히 적용되고, 시각적으로 안정적이라는 것을 나타낸다

최종 손실값 및 경사 계산

- 조합된 이미지의 최종 손실값을 계산하고, 그라디언트를 계산하여 스타일을 원하는 방식으로 최적화한다
- 이 함수를 반복적으로 사용하여 조합 이미지를 반복적으로 조정한다

- def compute_loss_and_grads(combination_image, base_image, style_reference_image):

```
    with tf.GradientTape() as tape:                                    # 그라디언트 계산
      # 조합 이미지, 원본 이미지, 참조 이미지를 사용하여 최종 손실값을 계산한다
      loss = compute_loss(combination_image, base_image, style_reference_image)

    # 최종 손실값에 대한 그라디언트를 계산하여 손실을 최소화하는 방향과 크기를 파악한다
    grads = tape.gradient(loss, combination_image)

    return loss, grads
```

옵티마이저 정의

- 학습률 100에서 시작하여 스텝마다 10%씩 감소시키는 옵티마이저 정의
- 처음에는 빠르게 수렴하고, 후반에 정밀하게 수렴

- optimizer = tf.keras.optimizers.SGD(
    tf.keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.90
)
)

초기 학습률, 100 배치 스텝마다 학습률 감소, 감소 비율

이미지 전처리

- 원본 이미지와 참조 이미지를 전처리하고,
- 원본 이미지를 전처리하여 조합 이미지 초기 상태를 만든다

- base_image = preprocess_image(base_image_path)
- style_reference_image = preprocess_image(style_reference_image_path)
- combination_image = tf.Variable(preprocess_image(base_image_path))

tf.GradientTape()을 사용할 수 있도록 tf.Variable()로 변경 가능한 텐서로 만든다

이미지 저장

GPU를 사용해 진행한다

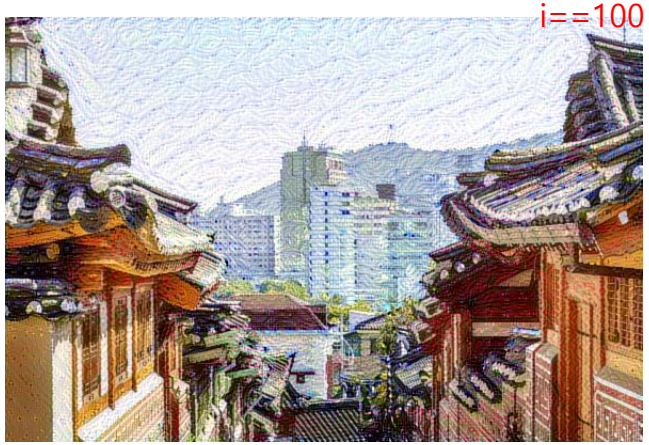
- from tensorflow import keras
- iterations = 4000

- for i in range(1, iterations + 1):
  loss, grads = **compute_loss_and_grads**(combination_image, base_image, style_reference_image) # 조합 이미지 반복 조정
  optimizer.apply_gradients([(grads, combination_image)]) # 옵티마이저를 이용하여 grads가 적은 방향으로 조합 이미지 업데이트
  if i % 100 == 0:
    print(f"{i}번째 반복: loss={loss:.2f}")
    img = deprocess_image(combination_image.numpy()) # 이미지 복원 함수로 출력할 수 있는 상태를 만든다
    fname = f"combination_image_at_iteration_{i}.png“ # 파일 이름 설정
    keras.utils.save_img(os.path.join(save_path, "neural_style/", fname), img) # 100번째마다 이미지 저장

생성 반복 과정

100번째 반복: loss=12426.37
200번째 반복: loss=9811.59
300번째 반복: loss=8781.66
400번째 반복: loss=8214.15
500번째 반복: loss=7847.91
600번째 반복: loss=7588.77
700번째 반복: loss=7394.89
800번째 반복: loss=7243.96
900번째 반복: loss=7123.04
1000번째 반복: loss=7024.04
1100번째 반복: loss=6941.69
1200번째 반복: loss=6872.26
1300번째 반복: loss=6813.27
1400번째 반복: loss=6762.69
1500번째 반복: loss=6719.05
1600번째 반복: loss=6681.19
1700번째 반복: loss=6648.24
1800번째 반복: loss=6619.48
1900번째 반복: loss=6594.18
2000번째 반복: loss=6571.91

2100번째 반복: loss=6552.25
2200번째 반복: loss=6534.87
2300번째 반복: loss=6519.45
2400번째 반복: loss=6505.75
2500번째 반복: loss=6493.57
2600번째 반복: loss=6482.72
2700번째 반복: loss=6473.09
2800번째 반복: loss=6464.48
2900번째 반복: loss=6456.81
3000번째 반복: loss=6449.93
3100번째 반복: loss=6443.81
3200번째 반복: loss=6438.31
3300번째 반복: loss=6433.38
3400번째 반복: loss=6428.98
3500번째 반복: loss=6425.03
3600번째 반복: loss=6421.50
3700번째 반복: loss=6418.32
3800번째 반복: loss=6415.48
3900번째 반복: loss=6412.92
4000번째 반복: loss=6410.63



100번째와 4000번째 비교

- 고흐 스타일의 터치가 점점 더 많이 표현된다