2019.10.14

Style Transfer

Youngtaek Hong, PhD

Visualizing What Convnets Learn

Python Version

ConvNet의 학습 시각화하기

딥러닝의 '블랙박스'

- 학습된 표현에서 사람이 이해하기 쉬운 형태를 뽑아내거나 제시하기가 어려움

ConvNet

- 시각적인 개념을 학습한 것이기 때문에 시각화하기에 좋다.

대표적 기법 세가지

- **컨브넷 중간 층의 출력(중간 층에 있는 활성화)을 시각화하기**
  - 연속적 컨브넷층이 입력을 어떻게 변형시키는지 이해하고 개별적인 컨브넷 필터의 의미 파악에 도움이 된다.
- **컨브넷 필터를 시각화하기**
  - 컨브넷의 필터가 찾으려는 시각적인 패턴과 개념이 무엇인지 상세하게 이해하는 데 도움이 된다.
- **클래스 활성화에 대한 히트맵을 이미지에 시각화하기**
  - 이미지의 어느 부분이 주어진 클래스에 속하는 데 기여했는지 이해하고 이미지에서 객체의 위치를 추정하는 데 도움이 된다.

중간 층의 활성화 시각화하기

중간 층의 활성화 시각화

- 어떤 입력이 주어졌을 때 네트워크에 있는 여러 합성곱과 풀링 층이 출력하는 **특성 맵**을 그리는 것이다.
- 네트워크에 의해 학습된 필터들이 **어떻게 입력을 분해하는지** 보여준다.
- 넓이, 높이, 채널의 세 개 차원에 대해 특성 맵을 시각화하는 것이 좋다.
- 각 채널은 독립적인 특성을 인코딩하므로 특성 맵의 각 채널 내용을 독립적인 2D 이미지로 그리는 것이 좋다.

중간 층의 활성화 시각화하기

<모델 로드하기>

```
[ ]   import keras
      keras.__version__
```

```
[ ]   from keras.models import load_model

      model = load_model('cats_and_dogs_small_2.h5')
      model.summary()  # 기억을 되살리기 위해서 모델 구조를 출력합니다
```

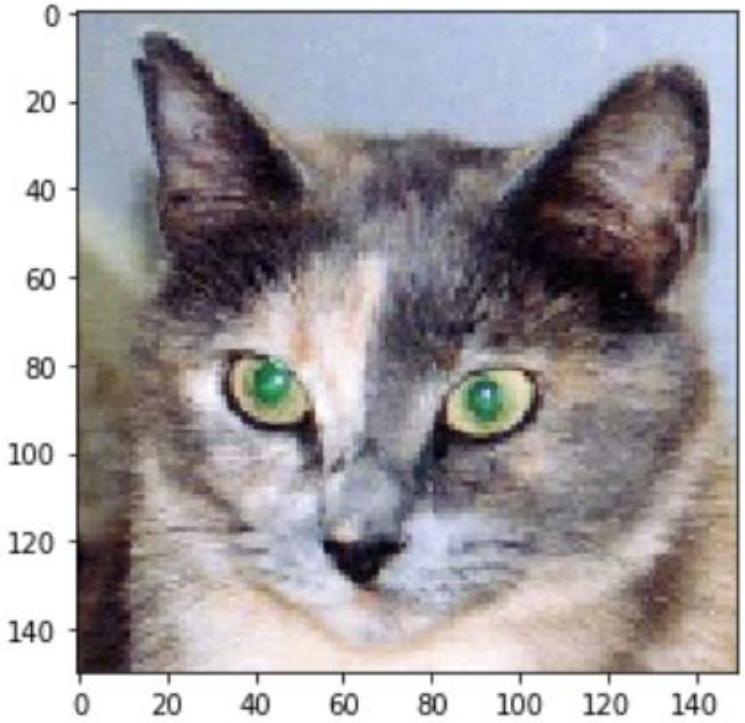
<입력 이미지 선택하기>

```
[ ]   img_path = './datasets/cats_and_dogs_small/test/cats/cat.1700.jpg'

      # 이미지를 4D 텐서로 변경합니다
      from keras.preprocessing import image
      import numpy as np

      img = image.load_img(img_path, target_size=(150, 150))
      img_tensor = image.img_to_array(img)
      img_tensor = np.expand_dims(img_tensor, axis=0)
      # 모델이 훈련될 때 입력에 적용한 전처리 방식을 동일하게 사용합니다
      img_tensor /= 255.

      # 이미지 텐서의 크기는 (1, 150, 150, 3)입니다
      print(img_tensor.shape)
```

중간 층의 활성화 시각화하기

<사진 출력하기>

```
[ ]   import matplotlib.pyplot as plt

[ ]   plt.imshow(img_tensor[0])
      plt.show()
```



← 출력된 이미지

중간 층의 활성화 시각화하기

<케라스 모델 작성>

- 확인하고 싶은 특성 맵 추출을 위해서 이미지 배치를 입력으로 받아 모든 합성곱과 풀링 층의 활성화를 출력하는 케라스 모델 작성

```
[ ]   from keras import models

      # 상위 8개 층의 출력을 추출합니다:
      layer_outputs = [layer.output for layer in model.layers[:8]]
      # 입력에 대해 8개 층의 출력을 반환하는 모델을 만듭니다:
      activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

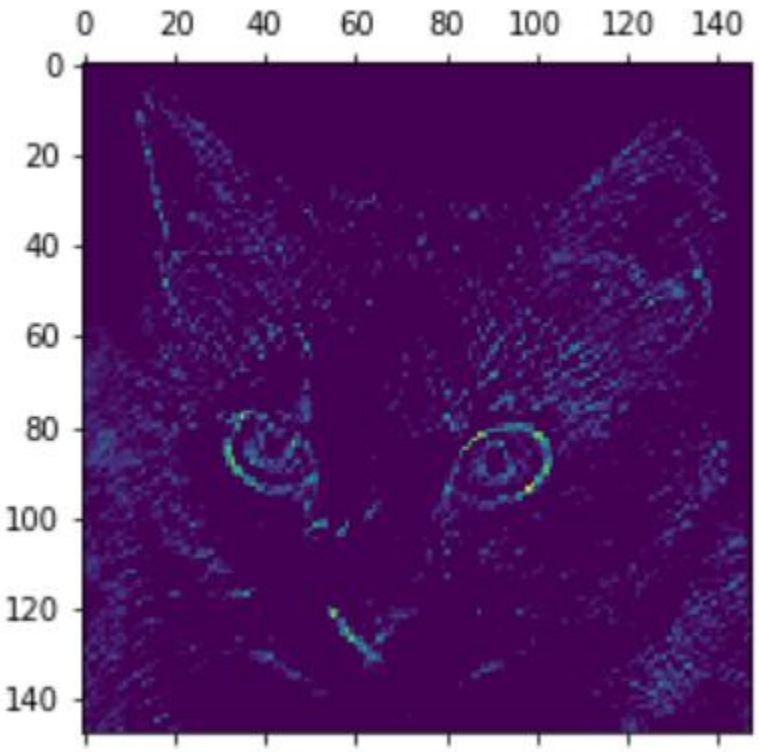
- 입력 이미지가 주입될 때 원본 모델의 활성화 값을 반환한다.
- 하나의 입력과 층의 활성화마다 하나씩 총 8개의 출력을 가진다.

```
# 층의 활성화마다 하나씩 8개의 넘파이 배열로 이루어진 리스트를 반환합니다:
activations = activation_model.predict(img_tensor)
```

중간 층의 활성화 시각화하기

<첫 번째 층의 활성화 중에서 20 번째 채널 그리기>

```
[ ]   plt.matshow(first_layer_activation[0, :, :, 19], cmap='viridis')
      plt.show()
```

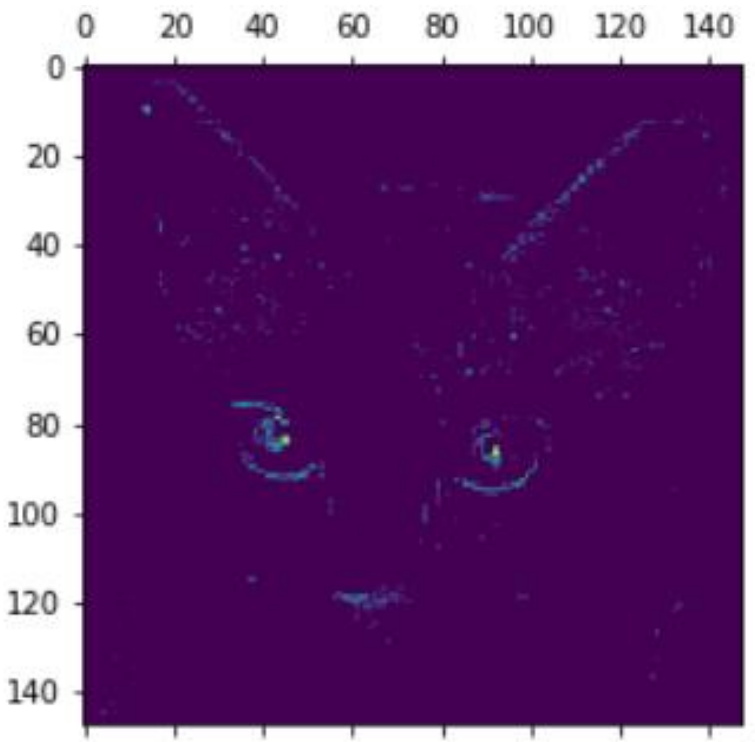


← 출력된 채널

중간 층의 활성화 시각화하기

<첫 번째 층의 활성화 중에서 16 번째 채널 그리기>

```
[ ]   plt.matshow(first_layer_activation[0, :, :, 15], cmap='viridis')
      plt.show()
```



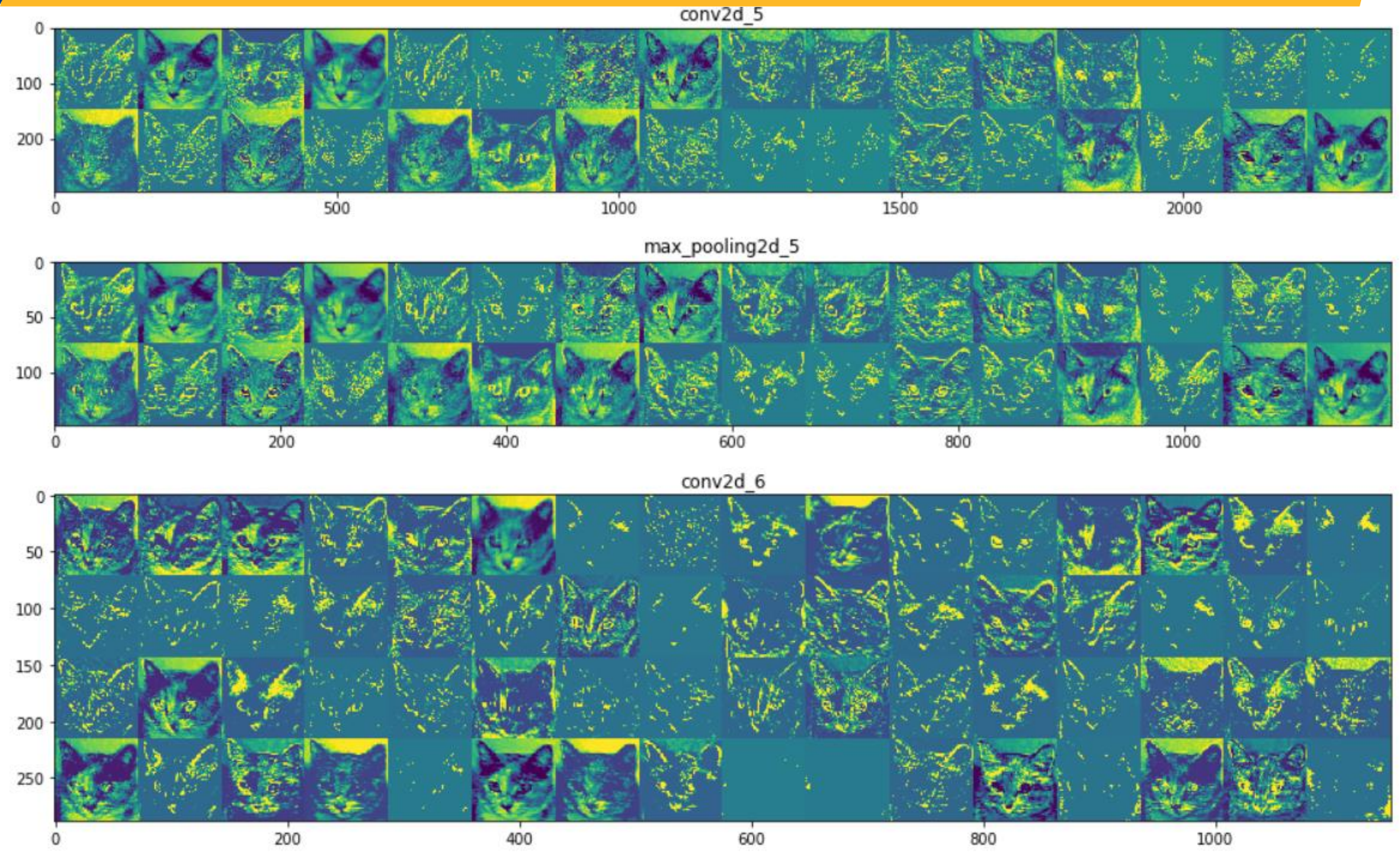
← 출력된 채널

중간 층의 활성화 시각화하기

<네트워크의 모든 활성화를 시각화>

```
[ ]  # 층의 이름을 그래프 제목으로 사용합니다
     layer_names = []
     for layer in model.layers[:8]:
         layer_names.append(layer.name)

     images_per_row = 16

     # 특성 맵을 그립니다
     for layer_name, layer_activation in zip(layer_names, activations):
         # 특성 맵에 있는 특성의 수
         n_features = layer_activation.shape[-1]

         # 특성 맵의 크기는 (1, size, size, n_features)입니다
         size = layer_activation.shape[1]

         # 활성화 채널을 위한 그리드 크기를 구합니다
         n_cols = n_features // images_per_row
         display_grid = np.zeros((size * n_cols, images_per_row * size))

         # 각 활성화를 하나의 큰 그리드에 채웁니다
         for col in range(n_cols):
             for row in range(images_per_row):
                 channel_image = layer_activation[0,
                                                  :, :,
                                                  col * images_per_row + row]
                 # 그래프로 나타내기 좋게 특성을 처리합니다
                 channel_image -= channel_image.mean()
                 channel_image /= channel_image.std()
                 channel_image *= 64
                 channel_image += 128
                 channel_image = np.clip(channel_image, 0, 255).astype('uint8')
                 display_grid[col * size : (col + 1) * size,
                              row * size : (row + 1) * size] = channel_image

         # 그리드를 출력합니다
         scale = 1. / size
         plt.figure(figsize=(scale * display_grid.shape[1],
                             scale * display_grid.shape[0]))
         plt.title(layer_name)
         plt.grid(False)
         plt.imshow(display_grid, aspect='auto', cmap='viridis')

     plt.show()
```

중간 층의 활성화 시각화하기



중간 층의 활성화 시각화하기

주목할 내용

- 첫 번째 층은 여러 종류의 에지 감지기를 모아 놓은 것 같다. 이 단계의 활성화에는 초기 사진에 있는 거의 모든 정보가 유지된다.
- 상위 층으로 갈수록 활성화는 점점 더 추상적으로 되고 시각적으로 이해하기 어려워진다. '고양이 귀', '고양이 눈 ' 과 같은 고수준의 개념을 인코딩하기 시작한다. 상위 층의 표현은 이미지의 시각적 콘텐츠에 관한 정보가 점점 줄어들고, 이미지의 클래스에 관한 정보가 점점 증가한다.
- 비어 있는 활성화가 층이 깊어짐에 따라 늘어난다. 첫 번째 층에서는 모든 필터가 입력 이미지에 활성화되었지만 층을 올라가면서 활성화되지 않는 필터들이 생긴다.

중간 층의 활성화 시각화하기

심층 신경망이 학습한 표현에서의 주요 특징

- 층에서 추출한 특성은 층의 깊이를 따라 점점 더 추상적이게 된다.
- 입력되는 원본 데이터에 대한 정보 정제 파이프라인처럼 작동
- 반복적인 변환을 통해 관계없는 정보를 걸러내고 유용한 정보는 강조되고 개선된다.

컨브넷 필터 시각화

컨브넷 필터 시각화

- 빈 입력 이미지에서 시작해서 특정 필터의 응답을 최대화하기 위해 컨브넷 입력 이미지에 경사 상승법을 적용하여 **선택된 필터가 최대로 응답하는 이미지를 결과**로 한다.

컨브넷 필터 시각화

<확률적 경사 상승법을 사용해 손실 함수를 정의>

```
[ ]  from keras.applications import VGG16
     from keras import backend as K

     model = VGG16(weights='imagenet',
                   include_top=False)

     layer_name = 'block3_conv1'
     filter_index = 0

     layer_output = model.get_layer(layer_name).output
     loss = K.mean(layer_output[:, :, :, filter_index])
```

```
[ ]  # gradients 함수가 반환하는 텐서 리스트(여기에서는 크기가 1인 리스트)에서 첫 번째 텐서를 추출합니다
     grads = K.gradients(loss, model.input)[0]
```

```
[ ]  # 0 나눗셈을 방지하기 위해 1e-5을 더합니다
     grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

- 경사 상승법 과정을 부드럽게 하기 위해 그래디언트 텐서를 L2 norm으로 나누어 정규화한다.
  → 따라서 입력 이미지에 적용할 수정량의 크기를 항상 일정 범위 안에 놓을 수 있다.

컨브넷 필터 시각화

<주어진 입력 이미지에 대해 손실 텐서와 그래디언트 텐서를 계산>

```
[ ]   iterate = K.function([model.input], [loss, grads])

      # 테스트:
      import numpy as np
      loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

<파이썬 루프를 만들어 확률적 경사 상승법을 구성>

```
[ ]   # 잡음이 섞인 회색 이미지로 시작합니다
      input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

      # 업데이트할 그래디언트의 크기
      step = 1.
      for i in range(40):   # 경사 상승법을 40회 실행합니다
          # 손실과 그래디언트를 계산합니다
          loss_value, grads_value = iterate([input_img_data])
          # 손실을 최대화하는 방향으로 입력 이미지를 수정합니다
          input_img_data += grads_value * step
```

컨브넷 필터 시각화

<텐서 처리 함수>

- 결과 이미지 텐서를 [0, 255] 사이의 정수(출력 가능한 이미지)로 변경하기 위한 후처리를 해준다.

```
[ ]  def deprocess_image(x):
         # 텐서의 평균이 0, 표준 편차가 0.1이 되도록 정규화합니다
         x -= x.mean()
         x /= (x.std() + 1e-5)
         x *= 0.1

         # [0, 1]로 클리핑합니다
         x += 0.5
         x = np.clip(x, 0, 1)

         # RGB 배열로 변환합니다
         x *= 255
         x = np.clip(x, 0, 255).astype('uint8')
         return x
```

컨브넷 필터 시각화

<층의 이름과 필터 번호를 입력으로 받는 함수>

- 필터 활성화를 최대화하는 패턴을 이미지 텐서로 출력한다.

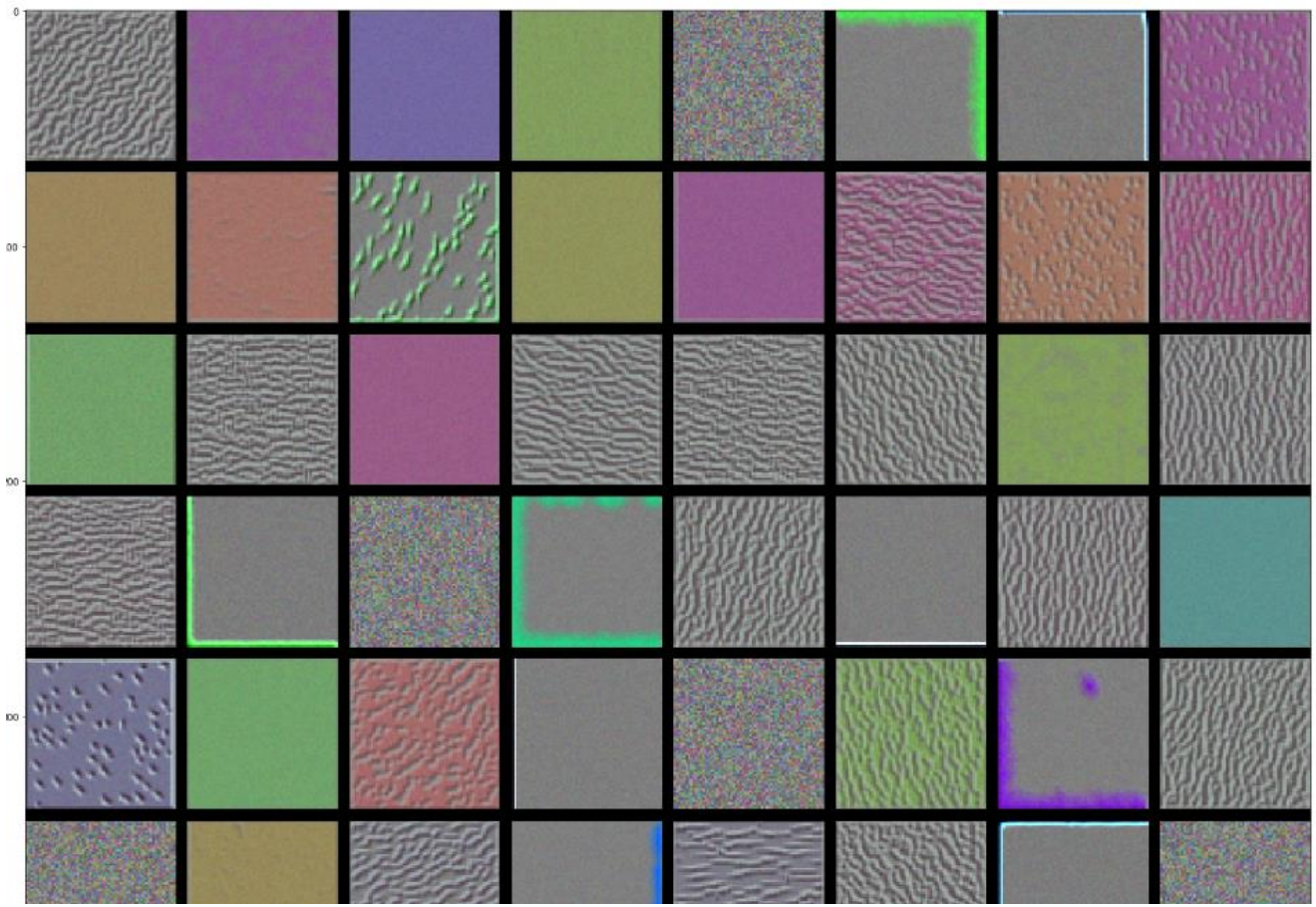
```
[ ]   def generate_pattern(layer_name, filter_index, size=150):
          # 주어진 층과 필터의 활성화를 최대화하기 위한 손실 함수를 정의합니다
          layer_output = model.get_layer(layer_name).output
          loss = K.mean(layer_output[:, :, :, filter_index])

          # 손실에 대한 입력 이미지의 그래디언트를 계산합니다
          grads = K.gradients(loss, model.input)[0]

          # 그래디언트 정규화
          grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

          # 입력 이미지에 대한 손실과 그래디언트를 반환합니다
          iterate = K.function([model.input], [loss, grads])

          # 잡음이 섞인 회색 이미지로 시작합니다
          input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

          # 경사 상승법을 40 단계 실행합니다
          step = 1.
          for i in range(40):
              loss_value, grads_value = iterate([input_img_data])
              input_img_data += grads_value * step

          img = input_img_data[0]
          return deprocess_image(img)
```

컨브넷 필터 시각화

<모든 층에 있는 필터를 시각화>
- 각 합성곱 블록의 첫 번째 층만 살펴본다.

```
[ ]  for layer_name in ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1']:
         size = 64
         margin = 5

         # 결과를 담을 빈 (검은) 이미지
         results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3), dtype='uint8')

         for i in range(8):  # results 그리드의 행을 반복합니다
             for j in range(8):  # results 그리드의 열을 반복합니다
                 # layer_name에 있는 i + (j * 8)번째 필터에 대한 패턴 생성합니다
                 filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

                 # results 그리드의 (i, j) 번째 위치에 저장합니다
                 horizontal_start = i * size + i * margin
                 horizontal_end = horizontal_start + size
                 vertical_start = j * size + j * margin
                 vertical_end = vertical_start + size
                 results[horizontal_start: horizontal_end, vertical_start: vertical_end, :] = filter_img

         # results 그리드를 그립니다
         plt.figure(figsize=(20, 20))
         plt.imshow(results)
         plt.show()
```

컨브넷 필터 시각화



컨브넷 필터 시각화

주목할 내용

- 컨브넷의 각 층의 필터의 조합으로 입력을 표현할 수 있는 일련의 필터를 학습하는데, 푸리에 변환을 사용해 신호를 일련의 코사인 함수로 분해할 수 있는 것과 같다. 이 컨브넷 필터들은 모델의 상위 층으로 갈수록 점점 더 복잡해진다.

클래스 활성화의 히트맵 시각화하기

클래스 활성화의 히트맵 시각화하기

- **이미지의 어느 부분이 컨브넷의 최종 분류 결정에 기여하는지** 이해하는 데 유용하다.
- 분류에 실수가 있는 경우 컨브넷의 결정 과정을 디버깅하는 데 도움이 된다.
- 이미지에 특정 물체가 있는 위치를 파악하는 데 사용할 수도 있다.
- 일반적으로 클래스 활성화 맵(CAM) 시각화라고 한다.
    - 입력 이미지에 대한 클래스 활성화의 히트맵을 만든다.
    - 이는 특정 출력 클래스에 대해 입력 이미지의 모든 위치에 대해 계산된 2D 점수 그리드이다. 클래스에 대해 각 위치가 얼마나 중요한지를 알려준다.

방법)

- 입력 이미지가 주어지면 합성곱 층에 있는 특성 맵의 출력을 추출한다.
- 특성 맵의 모든 채널의 출력에 채널에 대한 클래스의 그래디언트 평균을 곱한다.

클래스 활성화의 히트맵 시각화하기

```
[ ]  from keras.applications.vgg16 import VGG16

     K.clear_session()

     # 이전 모든 예제에서는 최상단의 완전 연결 분류기를 제외했지만 여기서는 포함합니다
     model = VGG16(weights='imagenet')
```

```
[ ]  from keras.preprocessing import image
     from keras.applications.vgg16 import preprocess_input, decode_predictions
     import numpy as np

     # 이미지 경로
     img_path = './datasets/creative_commons_elephant.jpg'

     # 224 × 224 크기의 파이썬 이미징 라이브러리(PIL) 객체로 반환됩니다
     img = image.load_img(img_path, target_size=(224, 224))

     # (224, 224, 3) 크기의 넘파이 float32 배열
     x = image.img_to_array(img)

     # 차원을 추가하여 (1, 224, 224, 3) 크기의 배치로 배열을 변환합니다
     x = np.expand_dims(x, axis=0)

     # 데이터를 전처리합니다(채널별 컬러 정규화를 수행합니다)
     x = preprocess_input(x)
```

클래스 활성화의 히트맵 시각화하기

```
[ ]   preds = model.predict(x)
      print('Predicted:', decode_predictions(preds, top=3)[0])
```

```
Predicted: [('n02504458', 'African_elephant', 0.9094213), ('n01871265', 'tusker', 0.08618258), ('n02504013', 'Indian_elephant', 0.004354576)]
```

```
[ ]   np.argmax(preds[0])
```

386

← 예측 벡터에서 최대 활성화된 항목인 '아프리카 코끼리' 클래스에 대한 인덱스

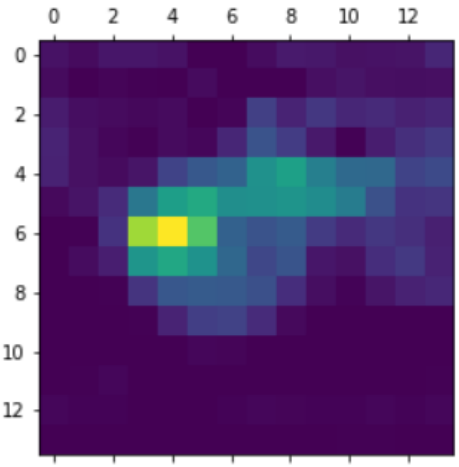
클래스 활성화의 히트맵 시각화하기

```
[ ]  # 예측 벡터의 '아프리카 코끼리' 항목
     african_elephant_output = model.output[:, 386]

     # VGG16의 마지막 합성곱 층인 block5_conv3 층의 특성 맵
     last_conv_layer = model.get_layer('block5_conv3')

     # block5_conv3의 특성 맵 출력에 대한 '아프리카 코끼리' 클래스의 그래디언트
     grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]

     # 특성 맵 채널별 그래디언트 평균 값이 담긴 (512,) 크기의 벡터
     pooled_grads = K.mean(grads, axis=(0, 1, 2))

     # 샘플 이미지가 주어졌을 때 방금 전 정의한 pooled_grads와 block5_conv3의 특성 맵 출력을 구합니다
     iterate = K.function([model.input], [pooled_grads, last_conv_layer.output[0]])

     # 두 마리 코끼리가 있는 샘플 이미지를 주입하고 두 개의 넘파이 배열을 얻습니다
     pooled_grads_value, conv_layer_output_value = iterate([x])

     # "아프리카 코끼리" 클래스에 대한 "채널의 중요도"를 특성 맵 배열의 채널에 곱합니다
     for i in range(512):
         conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

     # 만들어진 특성 맵에서 채널 축을 따라 평균한 값이 클래스 활성화의 히트맵입니다
     heatmap = np.mean(conv_layer_output_value, axis=-1)
```

<시각화를 위해 히트맵을 0과 1사이로 정규화>

```
[ ]  heatmap = np.maximum(heatmap, 0)
     heatmap /= np.max(heatmap)
     plt.matshow(heatmap)
     plt.show()
```

출력된 히트맵 →



클래스 활성화의 히트맵 시각화하기

<OpenCV 를 이용해 얻은 히트맵에 원본 이미지 겹친 이미지 만들기>

```
[ ]   import cv2

      # cv2 모듈을 사용해 원본 이미지를 로드합니다
      img = cv2.imread(img_path)

      # heatmap을 원본 이미지 크기에 맞게 변경합니다
      heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

      # heatmap을 RGB 포맷으로 변환합니다
      heatmap = np.uint8(255 * heatmap)

      # 히트맵으로 변환합니다
      heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

      # 0.4는 히트맵의 강도입니다
      superimposed_img = heatmap * 0.4 + img

      # 디스크에 이미지를 저장합니다
      cv2.imwrite('./datasets/elephant_cam.jpg', superimposed_img)
```

클래스 활성화의 히트맵 시각화하기

주목할 내용

- 코끼리 새끼의 귀가 강하게 활성화

  → 네트워크가 아프리카 코끼리와 인도 코끼리의 차이를 구분하는 방법일 것이라고 예측됨

Neural Style Transfer

Python Version

Neural Style Transfer

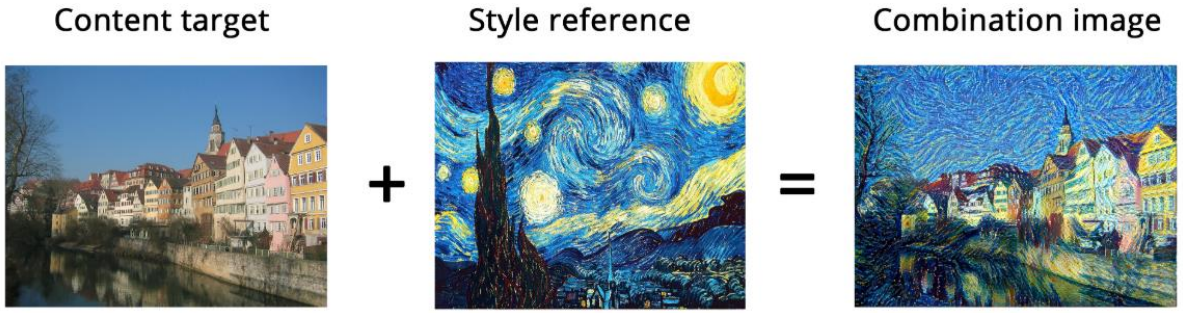
뉴럴 스타일 트랜스퍼

- 2015년 리온 게티스 등
- 타깃 이미지의 콘텐츠를 보존하면서 참조 이미지의 스타일을 타깃 이미지에 적용한다.
  - 스타일 : 질감, 색깔, 이미지에 있는 다양한 크기의 시각 요소
  - 콘텐츠 : 이미지에 있는 고수준의 대형 구조

최소화 손실 함수

```
loss = distance(style(reference_image) - style(generated_image)) +
       distance(content(original_image) - content(generated_image))
```

Distance : norm 함수
Content : 이미지의 콘텐츠 표현 계산
Style : 이미지의 스타일 표현 계산



Neural Style Transfer

콘텐츠 손실

- 네트워크 하위 층의 활성화는 이미지에 대한 국부적인 정보를 담고 있다. 반면 상위 층의 활성화일수록 점점 전역적이고 추상적인 정보를 담게 된다.
- 컨브넷 층의 활성화는 **이미지를 다른 크기의 콘텐츠로 분해**한다고 볼 수 있다.
- 컨브넷 상위 층의 표현을 사용하면 전역적이고 추상적인 이미지 콘텐츠를 찾는다.
- 타깃과 생성된 이미지를 사전 훈련된 컨브넷에 주입하여 상위 층의 활성화를 계산한다.
- 상위 층에서 보았을 때 생성된 이미지와 원본 타깃 이미지를 비슷하게 만든다.
- 컨브넷의 상위 층이 보는 것이 입력 이미지의 콘텐츠라면 이미지의 콘텐츠를 보존하는 방법으로 사용할 수 있다.

Neural Style Transfer

스타일 손실

- 스타일 손실은 컨브넷의 여러 층을 사용하는데, 이는 하나의 스타일이 아니라 참조 이미지에서 컨브넷이 추출한 모든 크기의 스타일을 잡는 것이다.
- 층의 활성화 출력의 그람 행렬을 스타일 손실로 사용하였다.
- 그람 행렬은 층의 특성 맵들의 내적이다.
- 내적은 층의 특성 사이에 있는 **상관관계**를 표현하고 이해하는데, 이는 특정 크기의 공간적인 패턴 통계를 잡아낸다.
- 스타일 참조 이미지와 생성된 이미지로 층의 활성화를 계산한다.
- 스타일 손실은 그 안에 내재된 상관관계를 비슷하게 보존한다.
- 스타일 참조 이미지와 생성된 이미지에서 여러 크기의 텍스처가 비슷하게 보이도록 만든다.

Neural Style Transfer with Keras

과정)

- 스타일 참조 이미지, 타깃 이미지, 생성된 이미지를 위해 VGG19의 층 활성화를 동시에 계산하는 네트워크를 설정한다.
- 세 이미지에서 계산한 층 활성화를 사용하여 앞서 설명한 손실 함수를 정의합니다. 이 손실을 최소화하여 스타일 트랜스퍼를 구현한다.
- 손실 함수를 최소화할 경사 하강법 과정을 설정한다.

Neural Style Transfer with Keras

```
[ ]  from keras.preprocessing.image import load_img, img_to_array, save_img

     # 변환하려는 이미지 경로
     target_image_path = './datasets/portrait.png'
     # 스타일 이미지 경로
     style_reference_image_path = './datasets/popova.jpg'

     # 생성된 사진의 차원
     width, height = load_img(target_image_path).size
     img_height = 400
     img_width = int(width * img_height / height)
```

```
[ ]  import numpy as np
     from keras.applications import vgg19

     def preprocess_image(image_path):
         img = load_img(image_path, target_size=(img_height, img_width))
         img = img_to_array(img)
         img = np.expand_dims(img, axis=0)
         img = vgg19.preprocess_input(img)
         return img

     def deprocess_image(x):
         # ImageNet의 평균 픽셀 값을 더합니다
         x[:, :, 0] += 103.939
         x[:, :, 1] += 116.779
         x[:, :, 2] += 123.68
         # 'BGR'->'RGB'
         x = x[:, :, ::-1]
         x = np.clip(x, 0, 255).astype('uint8')
         return x
```

<컨브넷에 입출력할 이미지 처리를 위한 유틸리티 함수>

Neural Style Transfer with Keras

<플레이스홀더 설정과 모델 로드>

```
[ ]  from keras import backend as K

     target_image = K.constant(preprocess_image(target_image_path))
     style_reference_image = K.constant(preprocess_image(style_reference_image_path))

     # 생성된 이미지를 담을 플레이스홀더
     combination_image = K.placeholder((1, img_height, img_width, 3))

     # 세 개의 이미지를 하나의 배치로 합칩니다
     input_tensor = K.concatenate([target_image,
                                   style_reference_image,
                                   combination_image], axis=0)

     # 세 이미지의 배치를 입력으로 받는 VGG 네트워크를 만듭니다.
     # 이 모델은 사전 훈련된 ImageNet 가중치를 로드합니다
     model = vgg19.VGG19(input_tensor=input_tensor,
                         weights='imagenet',
                         include_top=False)
     print('모델 로드 완료.')
```

Neural Style Transfer with Keras

<콘텐츠 손실>

```
[ ]  def content_loss(base, combination):
         return K.sum(K.square(combination - base))
```

<스타일 손실>

```
[ ]  def gram_matrix(x):
         features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
         gram = K.dot(features, K.transpose(features))
         return gram


     def style_loss(style, combination):
         S = gram_matrix(style)
         C = gram_matrix(combination)
         channels = 3
         size = img_height * img_width
         return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

<변위 손실>

생성된 픽셀을 사용해 계산, 생성된 이미지가 공간적인 연속성을 가지도록 도와주고 픽셀의 격자 무늬가 과도하게 나타나는 것을 막아준다.

```
[ ]  def total_variation_loss(x):
         a = K.square(
             x[:, :img_height - 1, :img_width - 1, :] - x[:, 1:, :img_width - 1, :])
         b = K.square(
             x[:, :img_height - 1, :img_width - 1, :] - x[:, :img_height - 1, 1:, :])
         return K.sum(K.pow(a + b, 1.25))
```

Neural Style Transfer with Keras

```
[ ]  # 층 이름과 활성화 텐서를 매핑한 딕셔너리
     outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
     # 콘텐츠 손실에 사용할 층
     content_layer = 'block5_conv2'
     # 스타일 손실에 사용할 층
     style_layers = ['block1_conv1',
                     'block2_conv1',
                     'block3_conv1',
                     'block4_conv1',
                     'block5_conv1']
     # 손실 항목의 가중치 평균에 사용할 가중치
     total_variation_weight = 1e-4
     style_weight = 1.
     content_weight = 0.025

     # 모든 손실 요소를 더해 하나의 스칼라 변수로 손실을 정의합니다
     loss = K.variable(0.)
     layer_features = outputs_dict[content_layer]
     target_image_features = layer_features[0, :, :, :]
     combination_features = layer_features[2, :, :, :]
     loss += content_weight * content_loss(target_image_features,
                                           combination_features)
     for layer_name in style_layers:
         layer_features = outputs_dict[layer_name]
         style_reference_features = layer_features[1, :, :, :]
         combination_features = layer_features[2, :, :, :]
         sl = style_loss(style_reference_features, combination_features)
         loss += (style_weight / len(style_layers)) * sl
     loss += total_variation_weight * total_variation_loss(combination_image)
```

<세 손실의 가중치 평균을 최소화>
사용하는 스타일 참조 이미지와 콘텐츠 이미지에 따라 content_weight 계수 조정하는 것이 필요하다.
높을수록 타깃 콘텐츠가 생성된 이미지에 많이 나타난다.

Neural Style Transfer with Keras

```
[ ]  # 손실에 대한 생성된 이미지의 그래디언트를 구합니다
     grads = K.gradients(loss, combination_image)[0]

     # 현재 손실과 그래디언트의 값을 추출하는 케라스 Function 객체입니다
     fetch_loss_and_grads = K.function([combination_image], [loss, grads])


     class Evaluator(object):

         def __init__(self):
             self.loss_value = None
             self.grads_values = None

         def loss(self, x):
             assert self.loss_value is None
             x = x.reshape((1, img_height, img_width, 3))
             outs = fetch_loss_and_grads([x])
             loss_value = outs[0]
             grad_values = outs[1].flatten().astype('float64')
             self.loss_value = loss_value
             self.grad_values = grad_values
             return self.loss_value

         def grads(self, x):
             assert self.loss_value is not None
             grad_values = np.copy(self.grad_values)
             self.loss_value = None
             self.grad_values = None
             return grad_values

     evaluator = Evaluator()
```

<경사 하강법 단계 설정>
L-BFGS 알고리즘 사용

Neural Style Transfer with Keras

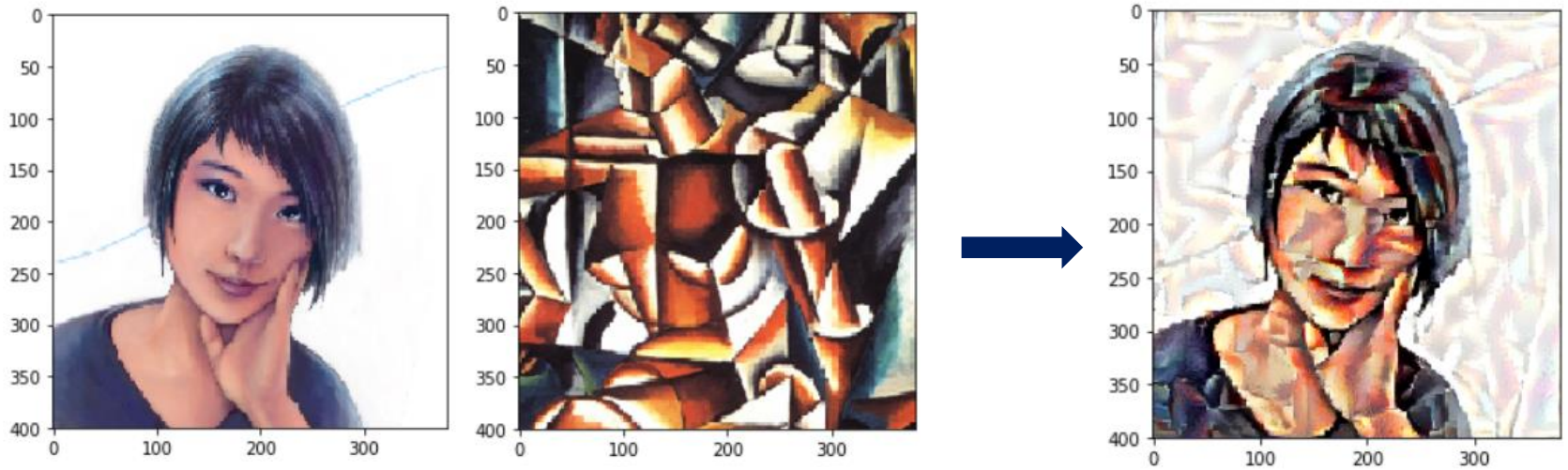
<경사 하강법 단계를 수행하고 알고리즘 반복마다 생성된 이미지를 저장>

```
[ ]  from scipy.optimize import fmin_l_bfgs_b
     import time

     result_prefix = 'style_transfer_result'
     iterations = 20

     # 뉴럴 스타일 트랜스퍼의 손실을 최소화하기 위해 생성된 이미지에 대해 L-BFGS 최적화를 수행
     # 초기 값은 타깃 이미지입니다
     # scipy.optimize.fmin_l_bfgs_b 함수가 벡터만 처리할 수 있기 때문에 이미지를 펼칩니다.
     x = preprocess_image(target_image_path)
     x = x.flatten()
     for i in range(iterations):
         print('반복 횟수:', i)
         start_time = time.time()
         x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x,
                                          fprime=evaluator.grads, maxfun=20)
         print('현재 손실 값:', min_val)
         # 생성된 현재 이미지를 저장합니다
         img = x.copy().reshape((img_height, img_width, 3))
         img = deprocess_image(img)
         fname = result_prefix + '_at_iteration_%d.png' % i
         save_img(fname, img)
         end_time = time.time()
         print('저장 이미지: ', fname)
         print('%d 번째 반복 완료: %ds' % (i, end_time - start_time))
```

Neural Style Transfer with Keras

```
[ ]  from matplotlib import pyplot as plt
```

```
[ ]  # 콘텐츠 이미지
     plt.imshow(load_img(target_image_path, target_size=(img_height, img_width)))
     plt.figure()

     # 스타일 이미지
     plt.imshow(load_img(style_reference_image_path, target_size=(img_height, img_width)))
     plt.figure()

     # 생성된 이미지
     plt.imshow(img)
     plt.show()
```



Neural Style Transfer with Keras

주목할 내용

- 스타일 이미지의 텍스처가 두드러지고 비슷한 패턴이 많을 때 잘 작동한다.
- 콘텐츠 타깃을 알아 보기 위해 수준 높은 이해가 필요하지 않을 때 잘 작동한다.
- 느리지만 간단한 변환을 수행하기 때문에 작고 빠른 컨브넷을 사용해 학습할 수 있다.