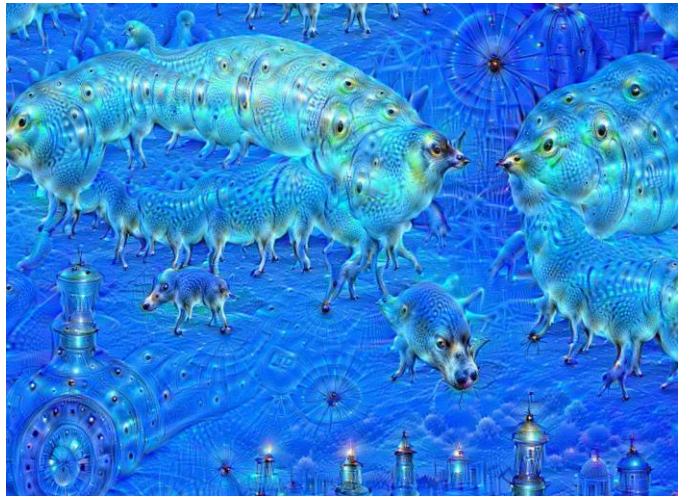
DeepDream

최석재 *lingua@naver.com*

DeepDream



- 딥드림은 합성곱 신경망이 학습한 표현을 사용하여
- 예술적으로 이미지를 조작하는 기법

- 사전에 훈련된 모델을 이용하여 입력된 이미지의 객체를 인식하고,
- 입력된 이미지에서 객체가 인식되면 그 특징을 더욱 강화한다
  - 예를 들어, 구름에서 동물의 모습이 조금 보일 때 그 특징을 강화하여 인식하는 것과 같은 원리이다
  - 사용하는 훈련된 모델은 ImageNet과 같이 많은 사물 이미지로 학습된 모델을 사용한다

- 딥드림은 신경망이 이미지를 어떻게 인식하고 해석하는지에 대한 이해를 줄 수 있다
- 하지만, 의미있는 딥드림 이미지 생성을 위해서는 다양한 파라미터 값을 잘 설정해야 한다

구글 드라이브 연결

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

CPU에서 작성하고, GPU에서 실행한다

입력할 이미지 출력

- 입력에 사용할 이미지를 출력해본다

- 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/cloud.png'
- display(_Imgdis(filename=image_path))

모델 구조 출력

- 여기서 사용할 모델은 Google에서 2015년 개발한 InceptionV3
- 이 모델은 1x1, 3x3, 5x5 등 다양한 크기의 필터를 사용하여
- 서로 다른 스케일의 특징을 추출하여 다양한 정보를 파악하게 하려한다
- https://keras.io/api/applications/inceptionv3/

- 다음의 코드로 모델의 구조를 확인한다

- from tensorflow.keras.applications.inception_v3 import InceptionV3
- model = InceptionV3()
- model.summary()

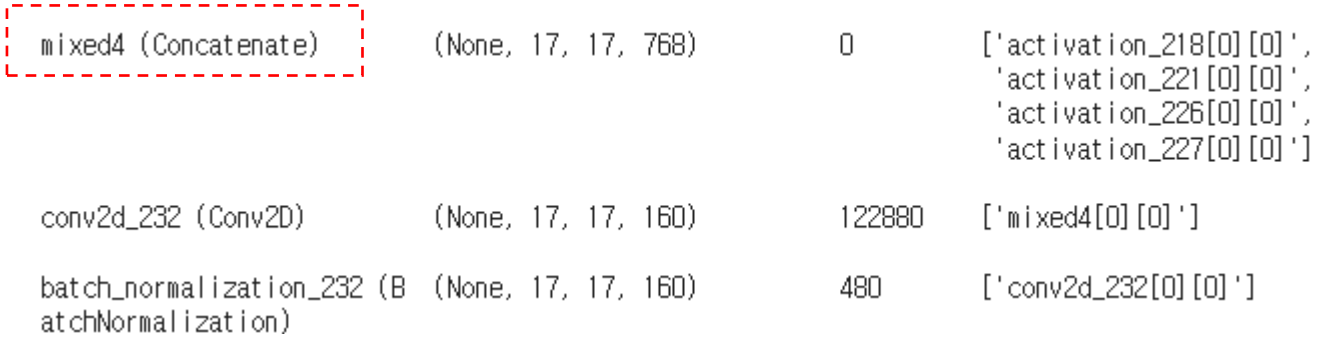
InceptionV3 모델 로드

- 완전연결층을 제외한 모델을 로드한다
- from tensorflow.keras.applications.inception_v3 import InceptionV3
- model = InceptionV3(weights="imagenet", include_top=False)

가중치 기여도 설정

- 인식된 객체의 특성을 강화하기 위하여 중간층의 가중치를 조절한다
- 앞쪽 레이어는 주로 이미지의 기본적인 요소인 이미지의 경계(edge), 가로/세로/대각선, 색상, 질감 등을 포착한다
- 뒷쪽의 깊은 레이어는 복잡한 특성인 눈, 귀, 사람, 자동차, 도시와 같은 형태 및 상황을 추상적으로 포착한다
- 보통 이 둘 사이의 중간 정도에서 조절하기 위하여 중간층 레이어의 가중치 기여도를 조절한다
- InceptionV3의 mixed 레이어는 다양한 크기를 결합한 레이어로서 많은 정보가 결합된 레이어이다

- layer_settings = {
  "mixed4": 1.0,
  "mixed5": 1.5,
  "mixed6": 2.0,
  "mixed7": 2.5,
  }



mixed4 (Concatenate)	(None, 17, 17, 768)	0	['activation_218[0][0]', 'activation_221[0][0]', 'activation_226[0][0]', 'activation_227[0][0]']
conv2d_232 (Conv2D)	(None, 17, 17, 160)	122880	['mixed4[0][0]']
batch_normalization_232 (BatchNormalization)	(None, 17, 17, 160)	480	['conv2d_232[0][0]']

모델 중간 레이어 결과 가져오기

- 다음의 중첩 리스트 컴프리헨션으로 앞에서 지정한 레이어를
- 모델에서 그 출력 결과와 함께 가져온다

- outputs_dict = dict(

  [(layer.name, layer.output) for layer in [model.get_layer(name) for name in layer_settings.keys()]]

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

1. layer_settings.keys()에서 설정한 레이어의 이름을 추출한다
2. model.get_layer(name) for name에서 모델에서 선택된 레이어를 가져온다
3. 선택된 레이어들을 리스트로 만든다

4. (layer.name, layer.output) for layer에서 각 레이어의 이름과 해당 레이어의 출력 결과를 받아 튜플로 묶는다
5. dict()에서 레이어의 이름과 출력을 dictionary 형태로 변환한다

손실함수

① 현재 레이어가 이미지를 인식한 결과에서 중요 부분을 얻기 위해 가장자리를 제외
② 출력 강도를 강조하기 위하여 값을 제곱
③ 레이어 출력의 전체적인 강도를 얻기 위하여 평균 계산
④ 설정된 가중치를 곱하여 계산 결과 조정
⑤ loss += 를 통하여 이제까지 계산된 값을 현재까지의 손실에 추가

손실을 크게 함으로써 이미지가 인식한 특징이 증폭되게 한다
뒤에서는 이 커진 손실값을 더 크게 하여 이미지를 업데이트한다
그 결과 인식된 특징은 더 과장된다

- 사용할 손실함수 정의
- 그러나 일반적인 손실함수가 아니라, 이미지를 과장되게 할 수 있는 함수이다

- import tensorflow as tf

- def compute_loss(input_image):
  features = feature_extractor(input_image) # 모델 레이어의 출력 결과를 추출
  loss = tf.zeros(shape=()) # 손실을 0으로 초기화
  for name in features.keys(): # 출력 레이어 각각에 대하여 (mixed4~mixed7)
  coeff = layer_settings[name] # 적용할 가중치를 설정한 값에서 가져온다 (1.0~2.5)
  activation = features[name] # 현재 레이어가 이미지를 인식한 결과
  loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :])) # 위 박스 참조. (배치차원, 높이, 너비, 색상)

  return loss

여기에서 loss 변수는 일반적인 개념의 손실값이라기보다는 "각 레이어의 극대화된 결괏값"이지만,
딥드림은 모델이 인식한 이미지의 일부 특징을 극대화하는 것이기 때문에
과장되게 인식하고 있다는 의미에서 손실값이라는 표현을 사용한다
이 값을 학습에서의 손실 지표로 사용하고, 일정 임계값에 이를 때까지 커지도록 유도하여
과장된 인식을 더욱 크게 만들려고 한다

이미지 업데이트 함수

- 결과 이미지를 업데이트하는 함수 정의

- def gradient_ascent_step(image, learning_rate):

```
    with tf.GradientTape() as tape:          # 그래디언트 값(grads)을 직접 다루기 위해 tf.GradientTape()을 사용
      tape.watch(image)                      # 어떤 값을 추적할지를 .watch()로 명시
      loss = compute_loss(image)             # 앞에서 작성한 손실함수로 이미지에 대한 손실값을 계산
    grads = tape.gradient(loss, image)       # 손실함수에 대한 이미지의 그래디언트를 계산
    grads = tf.math.l2_normalize(grads)      # 안정적인 학습을 위해 그래디언트 크기를 일정 수준으로 유지하는 L2 정규화를 한다
    image += learning_rate * grads           # 이미지를 그래디언트가 가리키는 방향으로 업데이트

    return loss, image
```

여기에서 loss는 각 레이어의 극대화된 결괏값이고, 이를 이용해 만든 grads는
손실을 극대화하는 값에 대한 기울기(과장된 값에 대한 기울기)이므로 기본적으로 높은값을 가지게 된다
여기에 learning_rate를 곱하여 이 영향을 어느 정도의 크기로 반영할지를 결정한다
이를 이미지 픽셀값에 더하면 픽셀값은 그래디언트가 가리키는 방향으로 이동하며 이미지를 변경한다
이 결과로 이미지는 원래보다 더 특정 특징을 강하게 표현하게 된다
예를 들어, 그래디언트가 이미지에서 '강아지'의 특징을 일부 포착하여 이를 강화하였다면
이미지는 강아지를 더 뚜렷하게 보여주는 쪽으로 업데이트된다

이미지 업데이트 반복 함수

- 이미지 업데이트를 반복하기 위한 함수 정의

- def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):

```
    for i in range(iterations):
        loss, image = gradient_ascent_step(image, learning_rate)        # 이미지 업데이트 과정 반복
        if max_loss is not None and loss > max_loss:                    # 과도한 변형을 막기 위해 임계값 설정
            break
        print(f"...스텝 {i}에서 손실값: {loss:.2f}")

    return image                                                        # 최종 업데이트된 이미지 반환
```

이미지 전처리 함수

- def preprocess_image(image_path):

```
    img = load_img(image_path)                  # 이미지를 불러온다
    img = img_to_array(img)                     # 넘파이 배열로 변환
    img = np.expand_dims(img, axis=0)           # 딥러닝 모델 구조에 맞게 맨 앞에 배치 차원을 추가한다
    img = tf.keras.applications.inception_v3.preprocess_input(img)

    return img
```

InceptionV3 모델에 맞게 입력 이미지 데이터를 전처리한다
이미지 데이터의 픽셀값을 정규화(-1~1)하여 모델이 학습된 방식에 맞도록 조정한다

이미지 복원 함수

- 인셉션의 preprocess_input()에서는 -1~1 사이의 값으로 정규화 과정을 거치므로
- 이를 다시 복원하기 위하여 1을 더하고 127.5를 곱하여 0~255 값으로 만든다

- def deprocess_image(img):

```
    img = img.reshape((img.shape[1], img.shape[2], 3))  # 이미지를 (높이, 너비, 3)의 구조로 만든다
    img += 1.0
    img *= 127.5
    img = np.clip(img, 0, 255).astype("uint8")          # 값을 0~255 사이의 값으로 제한한다

    return img
```

전처리 수행

- original_img = preprocess_image(image_path)
- original_shape = original_img.shape[1:3]

전처리 수행

(배치, 높이, 너비, 채널)에서 높이와 너비 정보 추출

파라미터 설정

- 딥드림 수행에 필요한 파라미터를 설정한다

- step = 20 # 학습률
- num_octave = 3 # 이미지를 3가지 옥타브(이미지의 스케일)로 처리
- octave_scale = 1.4 # 각 옥타브에서 수행할 이미지 크기 조정 비율
- iterations = 30 # 각 옥타브에서 수행할 최대 반복 횟수
- max_loss = 15.0 # 허용하는 최대 손실값(임계값)

옥타브 스케일

- 각 옥타브에 따른 다른 스케일의 이미지를 만들 준비를 한다

- successive_shapes = [original_shape] # 원본 이미지의 높이와 너비 정보
- for i in range(1, num_octave):
  shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
  successive_shapes.append(shape)
- successive_shapes = successive_shapes[::-1] # 순서 반전

356x644 size의 이미지를 예로 들면 다음과 같다

① 첫 번째 순회 (i==1)
- original_shape의 (356, 644)가 하나씩 dim으로 들어간다
- $356/1.4^1 = 254$
- $644/1.4^1 = 460$
- (254, 460)

② 두 번째 순회 (i==2)
- $356/1.4^2 = 181$
- $644/1.4^2 = 328$
- (181, 328)

원본 순회1 순회2 작은 스케일에서 큰 스케일 순서로 변경
- [(356, 644), (254, 460), (181, 328)] → [(181, 328), (254, 460), (356, 644)]

옥타브별 이미지 특징 강화

- 각 옥타브마다 이미지를 처리하여 특징을 강화한다

원본 이미지를 가장 작은 옥타브 크기인 successive_shapes[0]으로 조정한다
이 이미지는 디테일을 복원하는 데 사용된다
- shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

원본 이미지의 복사본을 만든다
복사본은 각 옥타브에서 수정된다
- img = tf.identity(original_img)

옥타브별 이미지 특징 강화

- for i, shape in enumerate(successive_shapes):

```
print(f"{shape} 크기의 {i}번째 옥타브 처리")
img = tf.image.resize(img, shape)                # 각 옥타브에서 복사본 이미지(img)를 해당 옥타브의 크기로 조정한다
img = gradient_ascent_loop(                      # img에 대하여 이미지 업데이트를 반복하여 특징을 강화한다
    img, iterations=iterations, learning_rate=step, max_loss=max_loss
)
upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape)
same_size_original = tf.image.resize(original_img, shape)
lost_detail = same_size_original - upscaled_shrunk_original_img
img += lost_detail
```

- 앞에서 조정된 가장 작은 원본 이미지를 현재 옥타브의 크기로 조정한다 (업스케일링)
- 원본 이미지도 같은 크기로 조정한다
- 업스케일링으로 손실된 디테일(lost_detail)을 계산한다
- 손실된 디테일을 이미지(img)에 더하여 디테일을 복원한다

축소된 이미지를 업스케일링하면 모든 디테일이 복원되지는 않는다
원본을 축소시켜 얻은 픽셀값에서 업스케일링 픽셀값을 빼면 손실된 디테일이 계산된다
이 값을 현재 img에 더하여 디테일을 복원한다

옥타브별 이미지 특징 강화

다음도 for 문 안에 있어야 한다
다음 옥타브를 위하여 현재 반복문에서 사용된 크기로 원본 이미지의 크기를 조정한다
즉, 원본 이미지를 다음 옥타브 크기인 successive_shapes[1]의 사이즈로 조정한다
원본 이미지를 작은 크기에서 점차 키워가며 앞의 내용을 반복하게 된다

shrunk_original_img = tf.image.resize(original_img, shape)

shrunk_original_img를 successive_shapes[0]에서 [1]로 업데이트

```
shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

img = tf.identity(original_img)
for i, shape in enumerate(successive_shapes):
  print(f"{shape} 크기의 {i}번째 옥타브 처리")
  img = tf.image.resize(img, shape)
  img = gradient_ascent_loop(
      img, iterations=iterations, learning_rate=step, max_loss=max_loss
  )
  upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape)
  same_size_original = tf.image.resize(original_img, shape)
  lost_detail = same_size_original - upscaled_shrunk_original_img
  img += lost_detail

  shrunk_original_img = tf.image.resize(original_img, shape)
```

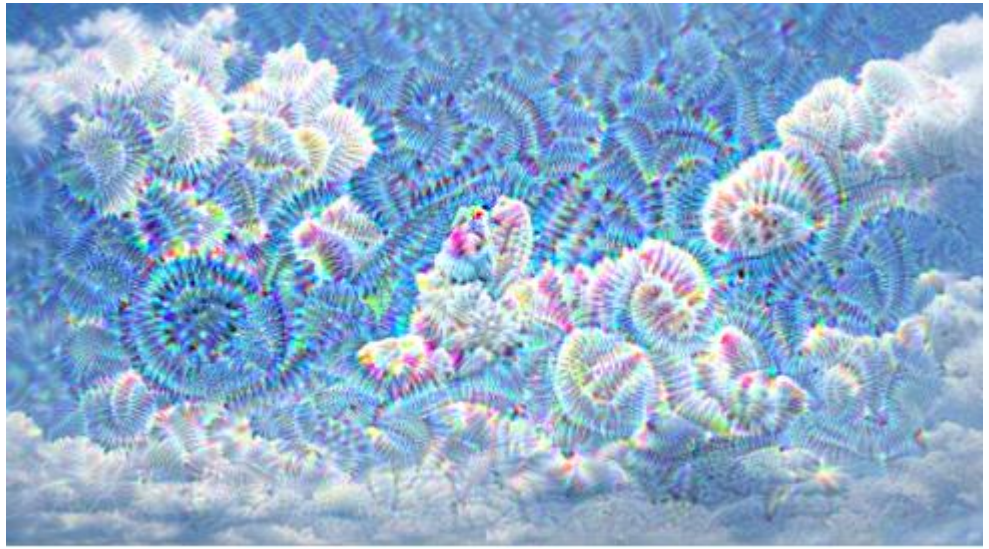
진행 과정

```
(181, 328) 크기의 0번째 옥타브 처리
...스텝 0에서 손실값: 0.83
...스텝 1에서 손실값: 0.98
...스텝 2에서 손실값: 1.57
...스텝 3에서 손실값: 2.20
...스텝 4에서 손실값: 2.92
...스텝 5에서 손실값: 3.60
...스텝 6에서 손실값: 4.28
...스텝 7에서 손실값: 5.07
...스텝 8에서 손실값: 5.79
...스텝 9에서 손실값: 6.63
...스텝 10에서 손실값: 7.57
...스텝 11에서 손실값: 8.42
...스텝 12에서 손실값: 9.35
...스텝 13에서 손실값: 10.41
...스텝 14에서 손실값: 11.47
...스텝 15에서 손실값: 12.52
...스텝 16에서 손실값: 13.65
...스텝 17에서 손실값: 14.62
(254, 460) 크기의 1번째 옥타브 처리
...스텝 0에서 손실값: 1.65
...스텝 1에서 손실값: 3.00
...스텝 2에서 손실값: 4.27
...스텝 3에서 손실값: 5.43
...스텝 4에서 손실값: 6.55
...스텝 5에서 손실값: 7.56
...스텝 6에서 손실값: 8.60
...스텝 7에서 손실값: 9.60
...스텝 8에서 손실값: 10.62
...스텝 9에서 손실값: 11.49
...스텝 10에서 손실값: 12.43
...스텝 11에서 손실값: 13.30
...스텝 12에서 손실값: 14.26
```

```
(356, 644) 크기의 2번째 옥타브 처리
...스텝 0에서 손실값: 1.71
...스텝 1에서 손실값: 3.06
...스텝 2에서 손실값: 4.32
...스텝 3에서 손실값: 5.40
...스텝 4에서 손실값: 6.51
...스텝 5에서 손실값: 7.56
...스텝 6에서 손실값: 8.66
...스텝 7에서 손실값: 9.84
...스텝 8에서 손실값: 11.22
...스텝 9에서 손실값: 12.94
```

최종 이미지 출력

- import matplotlib.pyplot as plt
- plt.imshow(deprocess_image(img.numpy())) # 이미지 복원 함수 사용
- plt.axis('off')
- plt.show()



다른 이미지 출력 결과

coast.jpg

pastry.png