CNNを学習した内容を可視化する

19nm705x

Ou Yanghuizi

目 次

CONTENTS

中間層の出力を可視化する

中間層の活性化を可視化するには、特定の入力をもとに、CNNのさまざまな畳み込み層とプーリング層によって出力される特徴マップを表示します。

層の出力はよく層の活性化と呼ばれます。活性化は活性化関数の出力です。ここでは、特徴マップを幅、高さ、深さの三つの次元で可視化します。各チャネルがエンコードする特徴量は比較的独立しているため、これらの特徴マップを可視化する正しい方式は、各チャネルの内容を2次元画像として個別にプロットすることです。

```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary()  # 参考までに出力
```

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 148, 148, 32)	896
maxpooling2d_5 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_6 (Conv2D)	(None, 72, 72, 64)	18496
maxpooling2d_6 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_7 (Conv2D)	(None, 34, 34, 128)	73856
maxpooling2d_7 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_8 (Conv2D)	(None, 15, 15, 128)	147584
maxpooling2d_8 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten_2 (Flatten)	(None, 6272)	0
dropout_1 (Dropout)	(None, 6272)	0
dense_3 (Dense)	(None, 512)	3211776
dense_4 (Dense)	(None, 1)	513

```
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

リスト 5-25：単一の画像を前処理

```
# 5.2.2項でsmallデータセットを格納したディレクトリへのパスであることに注意
img_path = \
    '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.jpg'

# この画像を4次元テンソルとして前処理
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)

# このモデルの訓練に使用された入力が次の方法で前処理されていることに注意
img_tensor /= 255.

# 形状は(1, 150, 150, 3)
print(img_tensor.shape)
```

次に、入力画像を取得します。

リスト 5-26：テスト画像を表示

```
import matplotlib.pyplot as plt

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

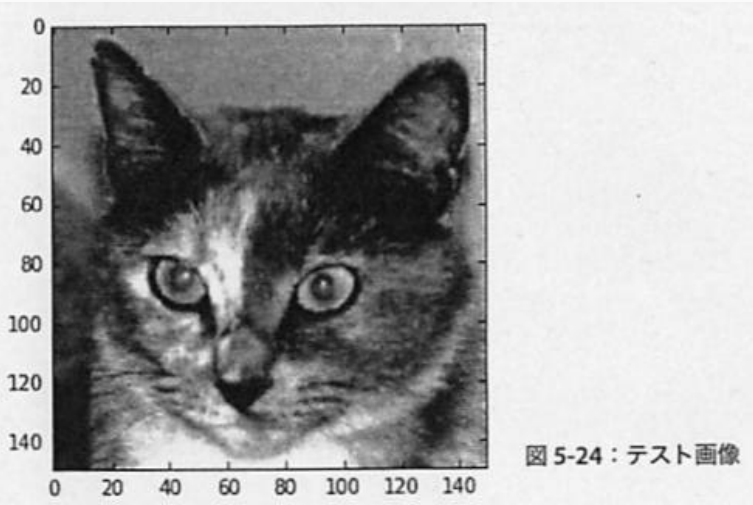


図 5-24：テスト画像

この画像を表示してみます。

リスト 5-27：入力テンソルと出力テンソルのリストに基づいてモデルをインスタンス化

```
from keras import models

# 出力側の8つの層から出力を抽出
layer_outputs = [layer.output for layer in model.layers[:8]]

# 特定の入力をもとに、これらの出力を返すモデルを作成
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

リスト 5-28：モデルを予測モードで実行

```
# 5つのNumPy配列（層の活性化ごとに1つ）のリストを返す
activations = activation_model.predict(img_tensor)
```

Kerasモデルをインスタンス化するには、入力テンソルと出力テンソルの 2 つの引数を指定します。

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 148, 148, 32)
```

リスト 5-29：3 番目のチャネルを可視化

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
plt.show()
```

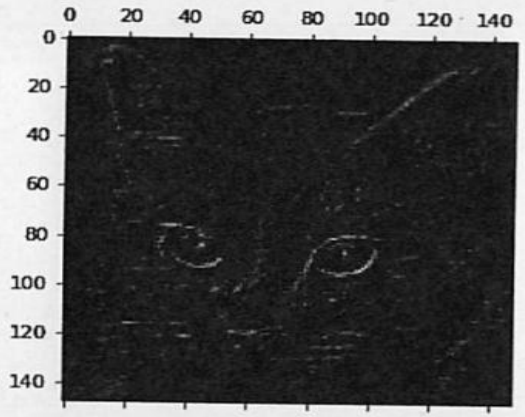


図 5-25：テストネコ画像に対する最初の層の活性化の 3 番目のチャネル

リスト 5-29のコードを実行した結果は図5-25のようになります。

リスト 5-30：30 番目のチャネルを可視化

```
plt.matshow(first_layer_activation[0, :, :, 30], cmap='viridis')
plt.show()
```

リスト 5-30 のコードを実行した結果は図 5-26 のようになります。

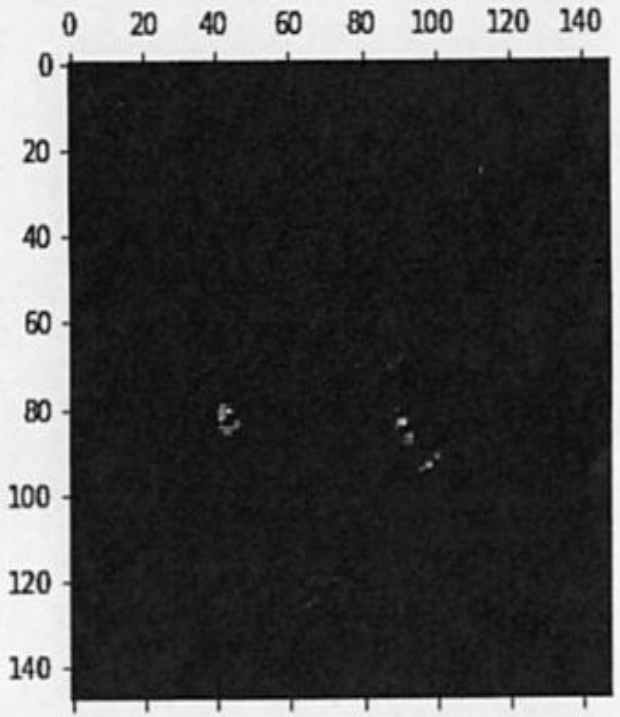
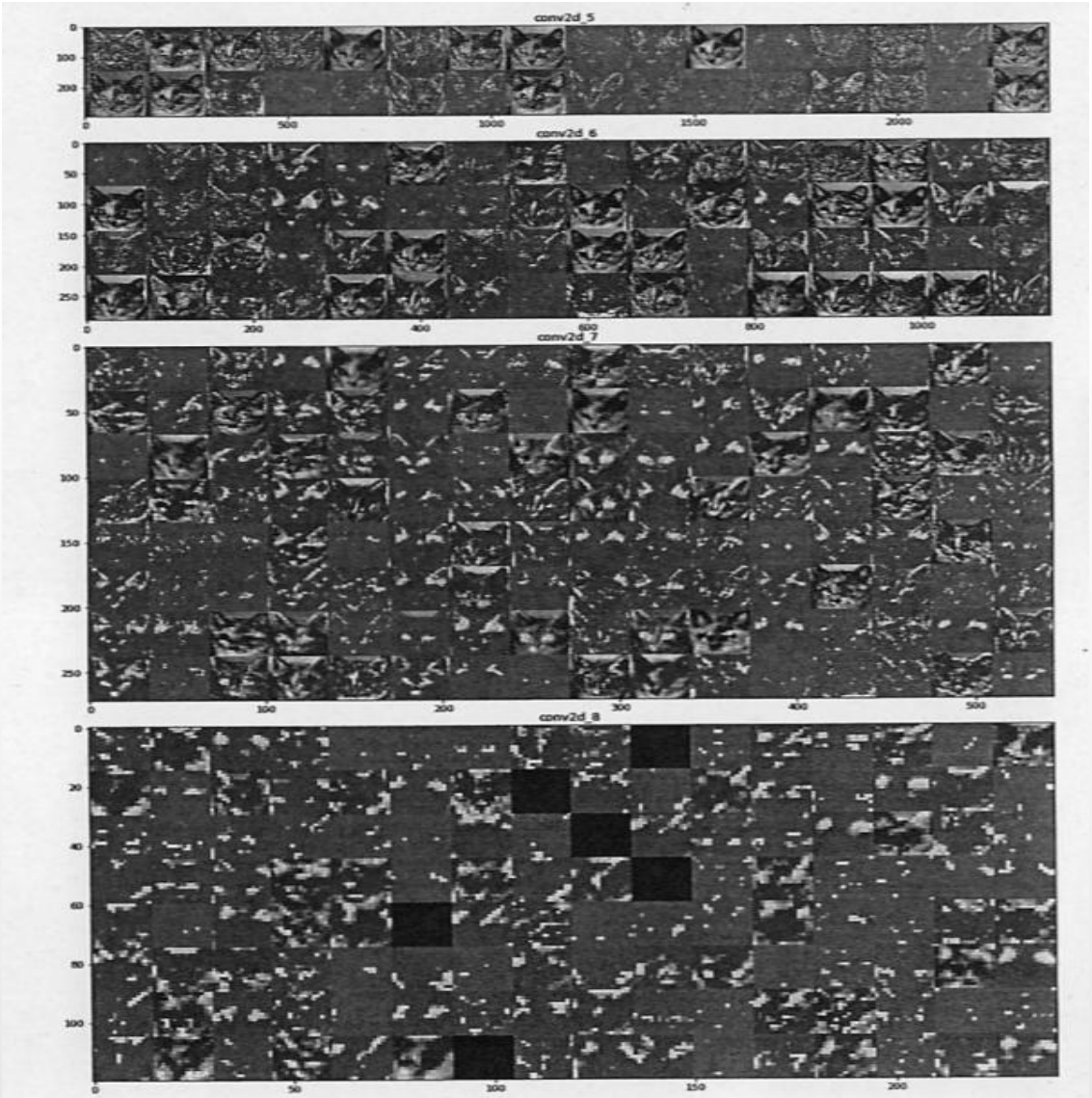


図 5-26：テスト猫画像に対する最初の層の活性化の 30 番目のチャネル

リスト 5-31：中間層の活性化ごとにすべてのチャネルを可視化

```
# プロットの一部として使用する層の名前
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

# 特徴マップを表示
for layer_name, layer_activation in zip(layer_names, activations):
    # 特徴マップに含まれている特徴量の数
    n_features = layer_activation.shape[-1]

    # 特徴マップの形状(1, size, size, n_features)
    size = layer_activation.shape[1]

    # この行列で活性化のチャネルをタイル表示
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    # 各フィルタを1つの大きな水平グリッドでタイル表示
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0, :, :,
                                             col * images_per_row + row]
            # 特徴量の見た目をよくするための後処理
            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

    # グリッドを表示
    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()
```



ここで、すべての中間層の活性化を完全に可視化してみます。８つの活性化マップごとにすべてのチャネルを抽出してプロットし、結果は１つの大きな画像テンソルにまとめて、チャネルを順番に積み上げて行きます。

CNNのフィルタを可視化する

CNNが学習したフィルタを調べる簡単な方法の１つは、各フィルタが応答することになっている視覚パターンを表示してみることです。この場合は、入力空間で勾配上昇法を使用できます。つまり、空の入力画像から始めて、CNNの入力画像の値に勾配降下法を適用することで、特定のフィルタの応答を最大化します。結果として得られる入力画像は、選択されたフィルタの応答性が最も高いものになります。

そのための手続きは単純です。特定の畳み込み層で特定のフィルタの値を最大化する損失関数を組み立てた後、確率的勾配降下法を使って入力画像の値を調整することで、この活性化の値を最大化します。

リスト 5-32：フィルタを可視化するための損失テンソルの定義

```
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])
```

リスト 5-33：入力に関する損失関数の勾配を取得

```
# gradientsの呼び出しはテンソル（この場合はサイズ1）のリストを返す
# このため、最初の要素（テンソル）だけを保持する
grads = K.gradients(loss, model.input)[0]
```

リスト 5-34：勾配の正規化

```
# 除算の前に1e-5を足すことで、0による除算を回避
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

リスト 5-35：入力値を NumPy 配列で受け取り、出力値を NumPy 配列で返す関数

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

# さっそくテストしてみる
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

リスト 5-36：確率的勾配降下法を使って損失値を最大化

```
# 最初はノイズが含まれたグレースケール画像を使用
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

# 勾配上昇法を40ステップ実行
step = 1.    # 各勾配の更新の大きさ
for i in range(40):
    # 損失値と勾配値を計算
    loss_value, grads_value = iterate([input_img_data])
    # 損失が最大になる方向に入力画像を調整
    input_img_data += grads_value * step
```

この時点で、確率的勾配降下法を実行するPythonループを定義できます。

リスト 5-37：テンソルを有効な画像に変換するユーティリティ関数

```
def deprocess_image(x):

    # テンソルを正規化：中心を0、標準偏差を0.1にする
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # [0, 1]でクリッピング
    x += 0.5
    x = np.clip(x, 0, 1)

    # RGB配列に変換
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

結果として得られる画像テンソルは、形状が(1,150,150,3)の浮動小数点数型のテンソルです。このテンソルに含まれている値は、[0,255]の範囲の整数ではない可能性があります。このため、このテンソルの後処理を行うことで、表示可能な画像に変換する必要があります。

リスト 5-38：フィルタを可視化するための関数

```
def generate_pattern(layer_name, filter_index, size=150):

    # ターゲット層のn番目のフィルタの活性化を最大化する損失関数を構築
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])

    # この損失関数を使って入力画像の勾配を計算
    grads = K.gradients(loss, model.input)[0]

    # 正規化トリック：勾配を正規化
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

    # 入力画像に基づいて損失値と勾配値を返す関数
    iterate = K.function([model.input], [loss, grads])

    # 最初はノイズが含まれたグレースケール画像を使用
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    # 勾配上昇法を40ステップ実行
    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)
```

```
>>> plt.imshow(generate_pattern('block3_conv1', 0))
>>> plt.show()
```

結果は図 5-29 のようになります。

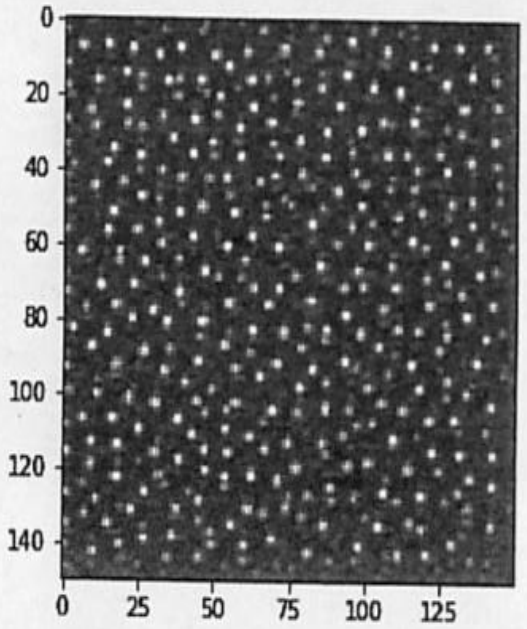


図 5-29：層 block3_conv1 の 0 番目のチャネルの応答を最大化するパターン

これらの要素をpython関数にまとめてみます。

リスト 5-39：層の各フィルタの応答パターンで構成されたグリッドの生成

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

    results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3))

    for i in range(8):  # resultsグリッドの行を順番に処理
        for j in range(8):  # resultsグリッドの列を順番に処理

            # layer_nameのフィルタi + (j * 8)のパターンを生成
            filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

            # resultsグリッドの矩形(i, j)に結果を配置
            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グリッドを表示
    plt.figure(figsize=(20, 20))
    plt.imshow(results)
    plt.show()
```

このコードを実行した結果は図 5-30～図 5-33 のようになります。

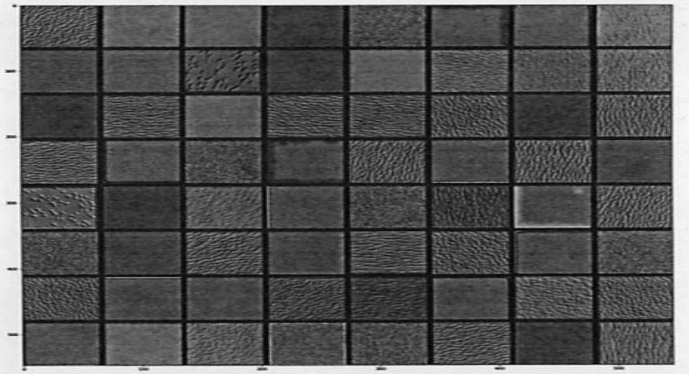


図 5-30：block1_conv1 層のフィルタパターン

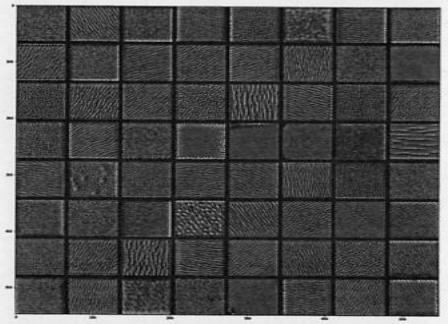


図 5-31：block2_conv1 層のフィルタパターン

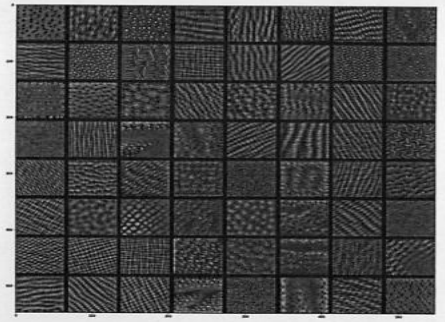


図 5-32：block3_conv1 層のフィルタパターン

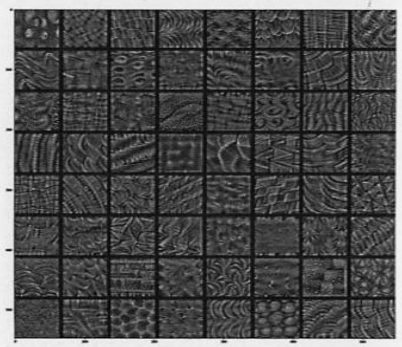
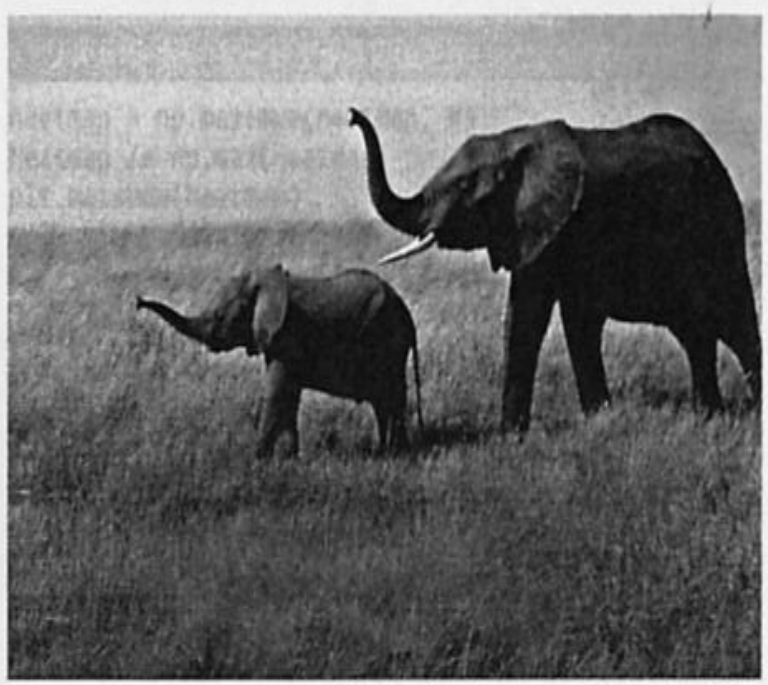


図 5-33：block4_conv1 層のフィルタパターン

クラスの活性化をヒートマップとして可視化する

この手法は、画像のどの部分がCNNの最終的な分類の決め手になったのかを理解するのに役立ちます。

これはCAMと総称される可視化手法であり、入力画像からクラス活性化のヒートマップを生成します。



※ Creative Commons ライセンスの素材

図 5-34：アフリカゾウのテスト画像

２つの質問を考えています。
この画像にアフリカゾウが含まれているとネットワークが考えた理由は何か?
アフリカゾウは画像のどの部分に含まれているか?

リスト 5-40：VGG16 ネットワークと学習済みの重みを読み込む

```
from keras.applications.vgg16 import VGG16

# 出力側に全結合分類器が含まれていることに注意
# ここまでのケースでは、この分類器を削除している
model = VGG16(weights='imagenet')
```

リスト 5-41：VGG16 モデルに合わせて入力画像を前処理

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

# ターゲット画像へのローカルパス
img_path = '/Users/fchollet/Downloads/creative_commons_elephant.jpg'

# ターゲット画像を読み込む：imgはサイズが224×224のPIL画像
img = image.load_img(img_path, target_size=(224, 224))

# xは形状が(224, 224, 3)のfloat32型のNumPy配列
x = image.img_to_array(img)

# この配列をサイズが(1, 224, 224, 3)のバッチに変換するために次元を追加
x = np.expand_dims(x, axis=0)

# バッチの前処理（チャネルごとに色を正規化）
x = preprocess_input(x)
```

このモデルはサイズが224*224の画像で訓練されており、いくつかのルールに従って前処理され、keras.applications.vgg16.preprocess_inputというユーティリティ関数としてパッケージ化されています。

```
>>> preds = model.predict(x)
>>> print('Predicted:', decode_predictions(preds, top=3)[0])
Predicted: [('n02504458', 'African_elephant', 0.90942144),
            ('n01871265', 'tusker', 0.08618243),
            ('n02504013', 'Indian_elephant', 0.0043545929)]
```

- アフリカゾウ（確率は92.5%）
- 牙を持つ動物（確率は7%）
- インドゾウ（確率は0.4%）

次に、この画像に学習済みのネットワークを適用し、予測ベクトルを人が読めるフォーマットにデコードします。

リスト 5-42：Grad-CAM アルゴリズムの設定

```
# 予測ベクトルの「アフリカゾウ」エントリ
african_elephant_output = model.output[:, 386]

# VGG16の最後の畳み込み層であるblock5_conv3の出力特徴マップ
last_conv_layer = model.get_layer('block5_conv3')

# block5_conv3の出力特徴マップでの「アフリカゾウ」クラスの勾配
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]

# 形状が(512,)のベクトル：
# 各エントリは特定の特徴マップチャネルの勾配の平均強度
pooled_grads = K.mean(grads, axis=(0, 1, 2))

# 2頭のアフリカゾウのサンプル画像に基づいて、pooled_gradsと
# block5_conv3の出力特徴マップの値にアクセスするための関数
iterate = K.function([model.input],
                     [pooled_grads, last_conv_layer.output[0]])

# これら2つの値をNumPy配列として取得
pooled_grads_value, conv_layer_output_value = iterate([x])

# 「アフリカゾウ」クラスに関する「このチャネルの重要度」を
# 特徴マップ配列の各チャネルに掛ける
for i in range(512):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

# 最終的な特徴マップのチャネルごとの平均値が
# クラスの活性化のヒートマップ
heatmap = np.mean(conv_layer_output_value, axis=-1)
```

この画像において最もアフリカゾウのように見える部分を可視化するために、Grad-CAMプロセスを設定してみます。

リスト 5-43：ヒートマップの後処理

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

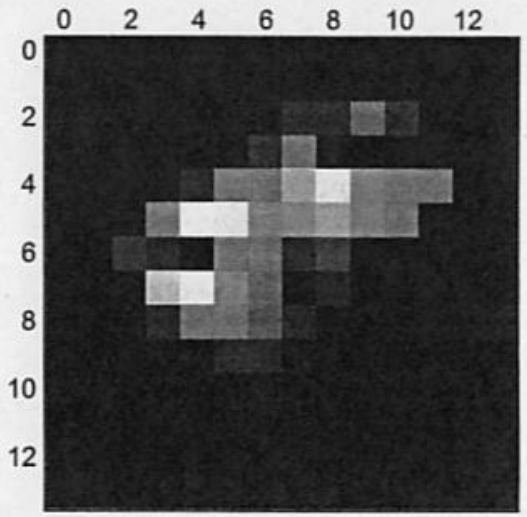


図 5-35：テスト画像でのアフリカゾウクラスの活性化ヒートマップ

リスト 5-44：ヒートマップを元の画像にスーパーインポーズ

```
import cv2

# cv2を使って元の画像を読み込む
img = cv2.imread(img_path)

# 元の画像と同じサイズになるようにヒートマップのサイズを変更
heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

# ヒートマップをRGBに変換
heatmap = np.uint8(255 * heatmap)

# ヒートマップを元の画像に適用
heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

# 0.4はヒートマップの強度係数
superimposed_img = heatmap * 0.4 + img

# 画像をディスクに保存
cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img)
```

最後に、OpenCVを使って、このヒートマップを元の画像にスーパーインポーズします。

図 5-36：クラス活性化ヒートマップを元の画像にスーパーインポーズ

結果は図5-36のようになります。