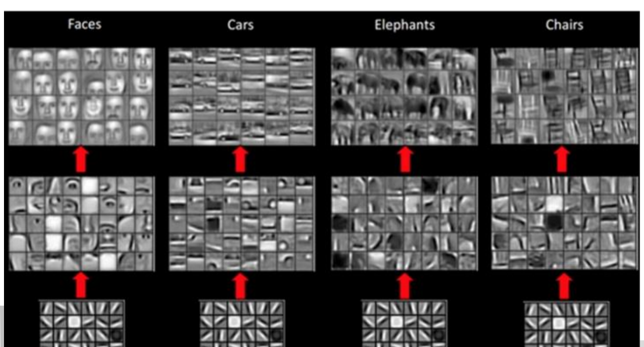
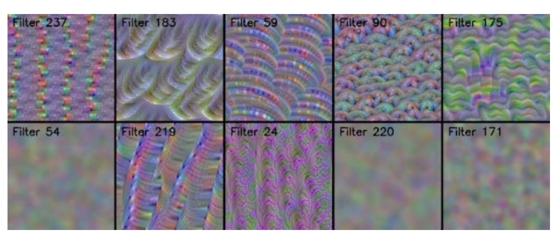
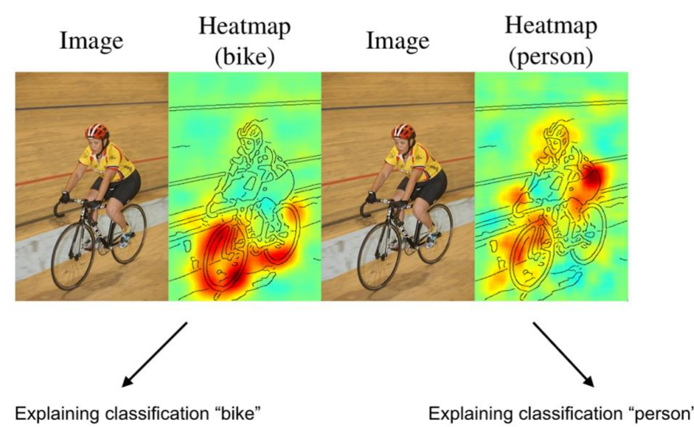
5.4Visualizing what convnets learn

- "black boxes": learning representations that are difficult to extract and present in a human-readable form
- convnets are highly amenable to visualization - *representations of visual concepts*
- Since 2013, a wide array of techniques have been developed for visualizing and interpreting these representations:
  - Visualizing intermediate convnet outputs (intermediate activations) — Understanding convnet layers transform their input, and individual convnet filters
  - Visualizing convnets filters — understanding precisely what visual pattern or concept each filter in a convnet is receptive to.
  - Visualizing heatmaps of class activation in an image—identified as belonging to a given class, localize objects in images.







5.4 Visualizing what convnets learn

5.4.1 Visualizing intermediate activations

- Visualizing intermediate activations -convolution and pooling layers
- How an input is decomposed into the different filters learned by the network.
- visualize feature maps - plotting the contents of every channel as a 2D image
- Let's start by loading the model that you saved in section 5.2:

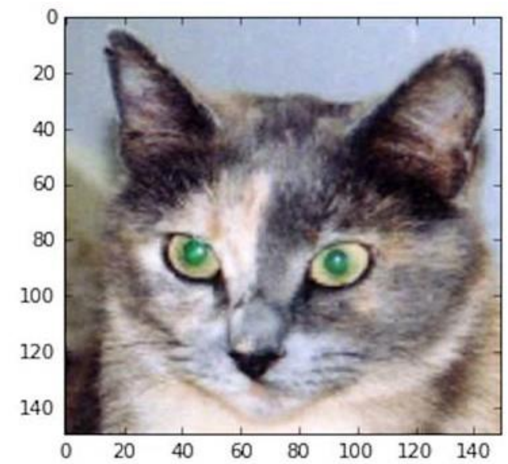
```
>>> from keras.models import load_model
>>> model= load_model('cats_and_dogs_small_2.h5') 앞에서 학습했던 결과를 다시 가져올 수 있음
>>> model.summary()  # As a reminder

Layer (type)                     Output Shape              Param #
================================================================
conv2d_5 (Conv2D)                (None, 148, 148, 32)      896
________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 74, 74, 32)            0
________________________________________________________________
conv2d_6 (Conv2D)                (None, 72, 72, 64)        18496
________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 36, 36, 64)            0
________________________________________________________________
conv2d_7 (Conv2D)                (None, 34, 34, 128)       73856
________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 17, 17, 128)           0
________________________________________________________________
conv2d_8 (Conv2D)                (None, 15, 15, 128)       147584
________________________________________________________________
max_pooling2d_8 (MaxPooling2 (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
================================================================
```

Listing 5.25 Preprocessing a single image

```
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) # (150, 150, 3)
img_tensor = np.expand_dims(img_tensor, axis=0) # 앞차원
# img_tensor = img_tensor.reshape((1,) + img_tensor.shape)
# 모델이 훈련될 때 입력에 적용한 전처리 방식
img_tensor /= 255.

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

Listing 5.25 Preprocessing a single image

```
import matplotlib.pyplot as plt

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



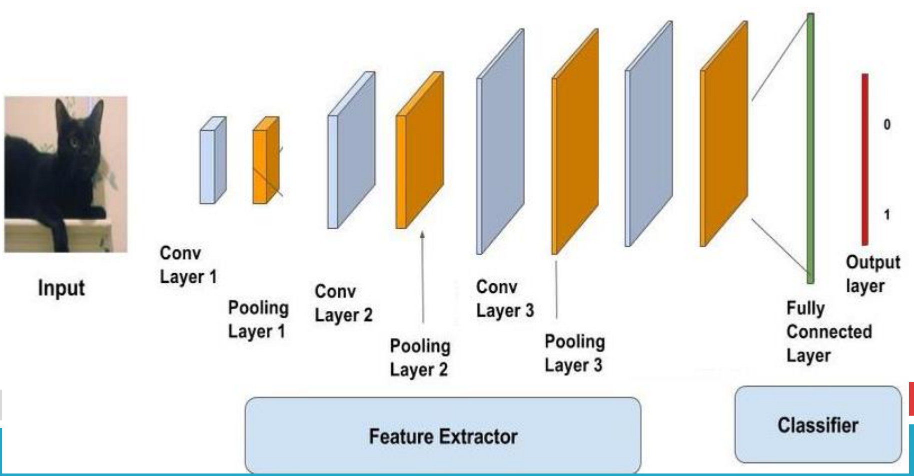
5.4 Visualizing what convnets learn

- outputs the activations of all convolution and pooling layers.
- Keras class `Model` – an input tensor (or list of input tensors) and an output tensor (or list of output tensors)

Listing 5.27 Instantiating a model from an input tensor and a list of output tensors

```
from keras import models

#  Extracts the outputs of the top eight layers
layer_outputs = [layer.output for layer in
                                   model.layers[:8]]
#  Creates a model that will return these outputs, given the model input
activation_model = models.Model(inputs=model.input,
                   outputs=layer_outputs) # 8 output layers
```

- multi-output model - one input and eight outputs, one output per layer activation



5.4 Visualizing what convnets learn

Listing 5.28 Running the model in predict mode

```
# Returns a list of 8 Numpy arrays: one array per layer activation
activations = activation_model.predict(img_tensor)
```

- activation of the first convolution layer for the cat image input:

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

첫 번째 conv의 필터의 수(채널 수)?

- It's a 148 × 148 feature map with 32 channels. Let's try plotting the **fourth channel** of the activation of the **first layer** of the original model (see figure 5.25).

Listing 5.29 Visualizing the fourth channel

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```

- This channel appears to encode a **diagonal edge** detector.

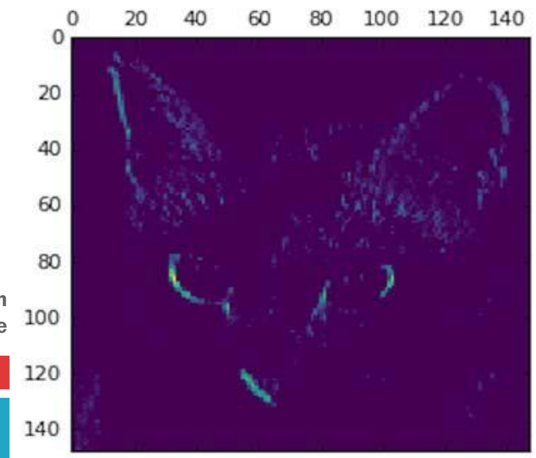


Figure 5.25 Fourth channel of the activation of the first layer on the test cat picture

5.4 Visualizing what convnets learn

- Let's try **the seventh** channel — the specific filters learned by convolution layers aren't deterministic.

Listing 5.30 Visualizing the seventh channel

```
plt.matshow(first_layer_activation[0, :, :, 7],
                                  cmap='viridis')
```

- This one looks like a "bright green dot" detector, useful to encode cat eyes.
- extract and plot every channel in each of the eight activation maps

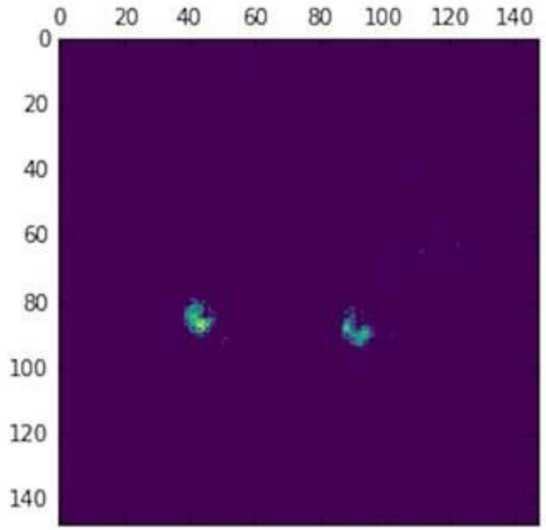
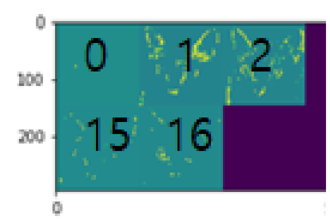


Figure 5.26 Seventh channel of the activation of the first layer on the test cat picture

Listing 5.31 Visualizing every channel in every intermediate activation

```
layer_names = [] # Names of the layers
for layer in model.layers[:8]:
    layer_names.append(layer.name)
images_per_row = 16
# Displays the feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1] # 맨뒤-32,32,64,64,128,128,128,128
    size = layer_activation.shape[1] # (1, size, size, n_features) # 148,74,72,…
    n_cols = n_features // images_per_row  # 32 // 16 몫 2
    display_grid = np.zeros((size * n_cols, images_per_row * size)) #(296,2368)
    for col in range(n_cols): # 2
        for row in range(images_per_row): # 16
            channel_image = layer_activation[0, :, :,
                                             col * images_per_row + row] #0~15, 16~31
            # Post-processes the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std() # z-dist, 0~1
            channel_image *= 64  중간에 중요한 정보가 많기 때문에..?      이거 안하고 해보면
            channel_image += 128 저자가 경험적으로 작성한 수치           흐리거나 까맣거나하는 걸 볼 수 있음
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
                          #np.clip(a, a_min, a_max, out=None)
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image
    scale = 1. / size # Displays the grid
    plt.figure(figsize=(scale * display_grid.shape[1], # 1./148 * 2368 = 16
                        scale * display_grid.shape[0])) # 1./148 * 296 = 2
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
plt.show()
```



highlevel로 갈수록 output들이 전혀 영향을 주지 못한다?
-> highlevel로 갈수록 그냥 검은색이 많아짐

5.4Visualizing what convnets learn

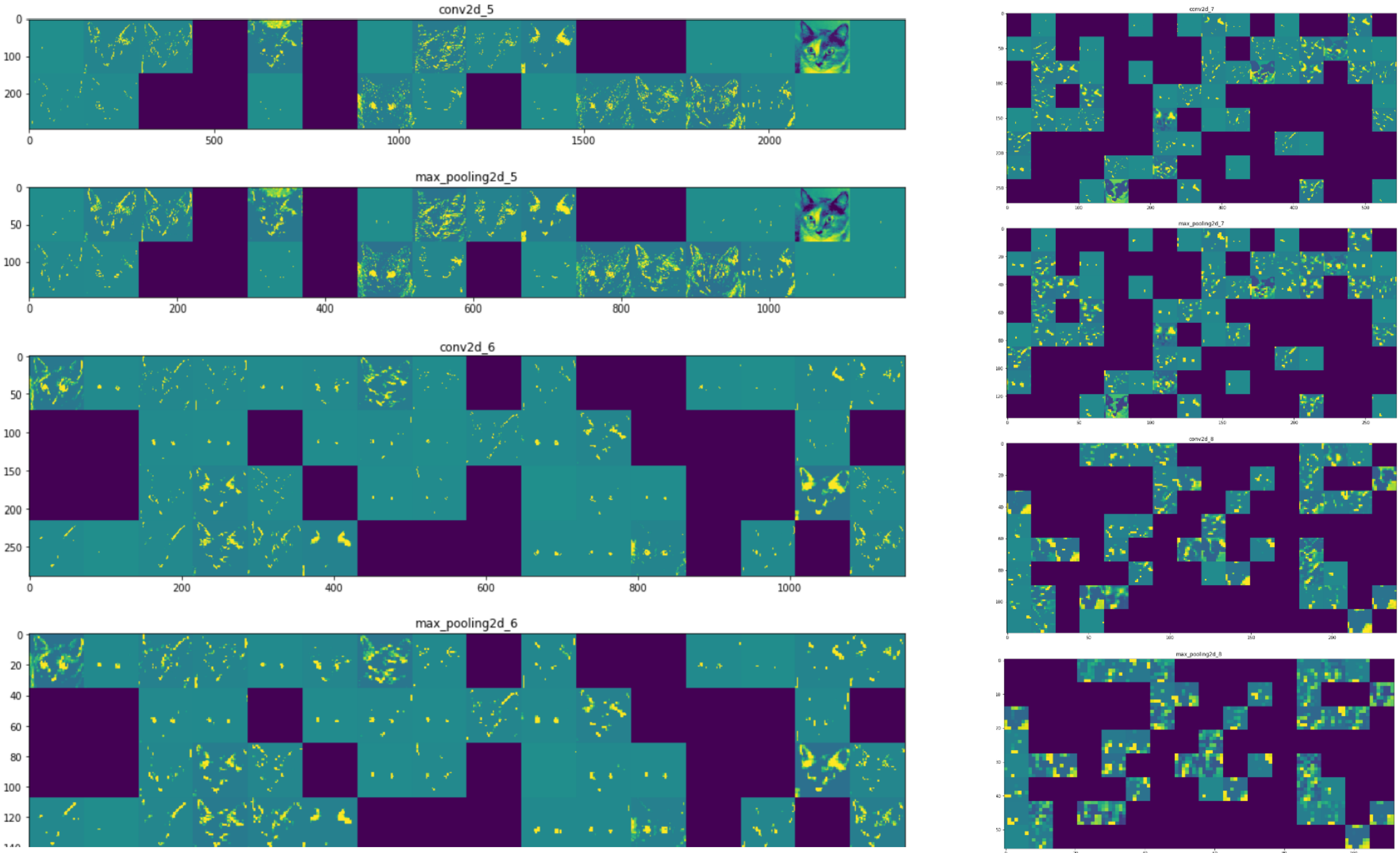


Figure 5.27 Every channel of every layer activation on the test cat picture

5.4Visualizing what convnets learn

- There are a few things to note here:
  - The first layer acts as a collection of various edge detectors in the initial picture.
  - As you go higher, the activations become increasingly abstract and less visually interpretable. – the more information about the target “cat ear” and “cat eye.”
  - The sparsity of the activations increases with the depth of the layer: no pattern encoded by the filter in the input image.
- A deep neural network effectively acts as an *information distillation pipeline*
- A human can remember which abstract objects were present in it (bicycle, tree) but can’t remember the specific appearance of these objects. (see, for example, figure 5.28).
- Your brain has learned to completely abstract its visual input—to transform it into high-level visual concepts while filtering out irrelevant visual

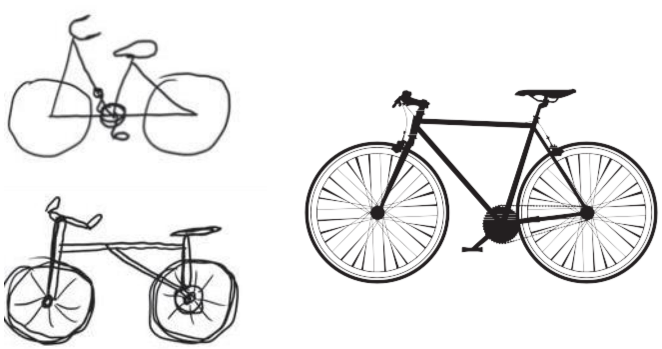
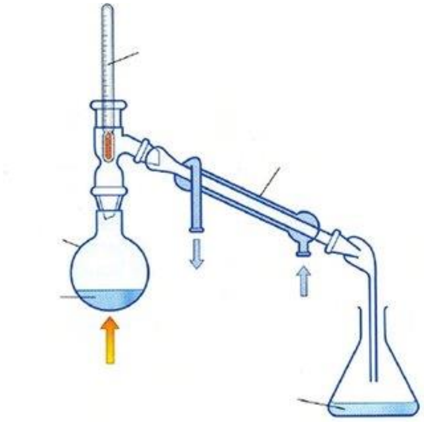


Figure 5.28 Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.



5.4.2 Visualizing convnet filters

gradient ascent라는 말은 신경망에 사실 없음
-> 이게 Key Point

- Display the visual pattern of each filter – *gradient ascent in input space* : applying *gradient descent* to the value of the input image of a convnet so as to *maximize* the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.
- The process is simple: build a loss function that maximizes the value of a given filter in a given convolution layer, and then use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. For instance, here's a loss for the activation of filter 0 in the layer `block3_conv1` of the VGG16 network, pretrained on ImageNet.

Listing 5.32 Defining the loss tensor for filter visualization

또한 loss를 계산할 방법이 없다..?..
그래서 자기가 알아서 loss를 define한다
(진행 방향이 반대임): 필터가 기존 이미지 방향으로 어떻게 변화하는지를 보는 것이라
weight가 반대 방향으로 진행된다고 봐야함?

```
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 # (?,?,?,256)
loss = K.mean(layer_output[:, :, :, filter_index]) # 0
```

freezing때문에 weight를 바꾸고 싶어도 바꾸지 못함
-> 그래서 그림을 바꾸기로함

- gradient of this loss with respect to the model's input - **`gradients`** function packaged with the **`backend`** module of Keras.

Listing 5.33 Obtaining the gradient of the loss with regard to the input

```
grads = K.gradients(loss, model.input)[0] # filter 0, (?, ?, ?, 3)
# The call to gradients returns a list of tensors (of size 1 in this case).
```

- Normalize the gradient tensor by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor).
- This ensures that the magnitude of the updates done to the input image is always within the same range.

Listing 5.34 Gradient-normalization trick

```
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
# + 1e-5: avoid accidentally dividing by 0
```

- loss tensor and the gradient tensor computation for given an input image - define a Keras backend function: **iterate**, a function that takes a Numpy tensor (as a list of tensors of size 1) and returns a list of two Numpy tensors: the loss value and the gradient value.

Listing 5.35 Fetching Numpy output values given Numpy input values

```
iterate = K.function([model.input], [loss, grads])
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

5.4 Visualizing what convnets learn

- At this point, you can define a Python loop to do stochastic gradient descent.

Listing 5.36 Loss maximization via stochastic gradient descent

```
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.
  # Starts from a gray image with some noise
step = 1. # Magnitude of each gradient update
for i in range(40):   # Runs gradient ascent for 40 steps
    # Computes the loss value and gradient value
    loss_value, grads_value = iterate([input_img_data])
    # Adjusts the input image in the direction that maximizes the loss
    input_img_data += grads_value * step
```

- The resulting image tensor is a floating-point tensor of shape `(1, 150, 150, 3)` → values of integers within [0, 255].

Listing 5.37 Utility function to convert a tensor into a valid image

```
def deprocess_image(x):
    # Normalizes the tensor: centers on 0, ensures that std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1
    x += 0.5
    x = np.clip(x, 0, 1) # Clips to [0, 1]
    # Converts to an RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

5.4 Visualizing what convnets learn

- Let's put them together into a Python function that takes as input a layer name and a filter index, and returns a valid image tensor representing the pattern that maximizes the activation of the specified filter.

Listing 5.38 Function to generate filter visualizations

```
def generate_pattern(layer_name, filter_index, size=150): # block3_conv1, filter 0
    # Builds a loss function that maximizes the activation of the nth filter of the layer
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])

    # Computes the gradient of the input picture with regard to this loss
    grads = K.gradients(loss, model.input)[0]

    # Normalization trick: normalizes the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5) # L2

    # Returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # Starts from a gray image with some noise
    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    step = 1. # Magnitude of each gradient update
    for i in range(40):   # Runs gradient ascent for 40 steps
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step
    img = input_img_data[0]
    return deprocess_image(img) # numpy→image format
```

- Let's try it (see figure 5.29):

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

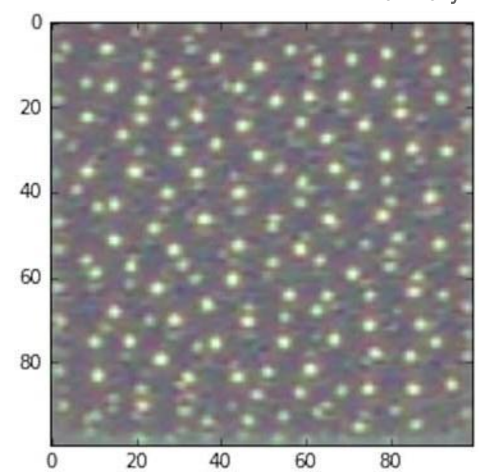


Figure 5.29 Pattern that the zeroth channel in layer block3_conv1 responds to maximally