DEEP LEARNING with Python

François Chollet

MANNING

5

Deep learning for computer vision

This chapter covers

- Understanding convolutional neural networks (convnets)
- Using data augmentation to mitigate overfitting
- Using a pretrained convnet to do feature extraction
- Fine-tuning a pretrained convnet
- Visualizing what convnets learn and how they make classification decisions

This chapter introduces convolutional neural networks, also known as *convnets*, a type of deep-learning model almost universally used in computer vision applications. You'll learn to apply convnets to image-classification problems—in particular those involving small training datasets, which are the most common use case if you aren't a large tech company.

5.2 Training a convnet from scratch on a small dataset

Having to train an image-classification model using very little data is a common situation, which you'll likely encounter in practice if you ever do computer vision in a professional context. A "few" samples can mean anywhere from a few hundred to a few tens of thousands of images. As a practical example, we'll focus on classifying images as dogs or cats, in a dataset containing 4,000 pictures of cats and dogs (2,000 cats, 2,000 dogs). We'll use 2,000 pictures for training—1,000 for validation, and 1,000 for testing.

In this section, we'll review one basic strategy to tackle this problem: training a new model from scratch using what little data you have. You'll start by naively training a small convnet on the 2,000 training samples, without any regularization, to set a baseline for what can be achieved. This will get you to a classification accuracy of 71%. At that point, the main issue will be overfitting. Then we'll introduce *data augmentation*, a powerful technique for mitigating overfitting in computer vision. By using data augmentation, you'll improve the network to reach an accuracy of 82%.

In the next section, we'll review two more essential techniques for applying deep learning to small datasets: *feature extraction with a pretrained network* (which will get you to an accuracy of 90% to 96%) and *fine-tuning a pretrained network* (this will get you to a final accuracy of 97%). Together, these three strategies—training a small model from scratch, doing feature extraction using a pretrained model, and fine-tuning a pretrained model—will constitute your future toolbox for tackling the problem of performing image classification with small datasets.

5.2.1 The relevance of deep learning for small-data problems

You'll sometimes hear that deep learning only works when lots of data is available. This is valid in part: one fundamental characteristic of deep learning is that it can find interesting features in the training data on its own, without any need for manual feature engineering, and this can only be achieved when lots of training examples are available. This is especially true for problems where the input samples are very high-dimensional, like images.

But what constitutes lots of samples is relative—relative to the size and depth of the network you're trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundred can potentially suffice if the model is small and well regularized and the task is simple. Because convnets learn local, translation-invariant features, they're highly data efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You'll see this in action in this section.

What's more, deep-learning models are by nature highly repurposable: you can take, say, an image-classification or speech-to-text model trained on a large-scale dataset and reuse it on a significantly different problem with only minor changes. Specifically, in the case of computer vision, many pretrained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. That's what you'll do in the next section. Let's start by getting your hands on the data.

5.2.2 Downloading the data

The Dogs vs. Cats dataset that you'll use isn't packaged with Keras. It was made available by Kaggle as part of a computer-vision competition in late 2013, back when convnets weren't mainstream. You can download the original dataset from www.kaggle.com/c/dogs-vs-cats/data (you'll need to create a Kaggle account if you don't already have one—don't worry, the process is painless).

The pictures are medium-resolution color JPEGs. Figure 5.8 shows some examples.

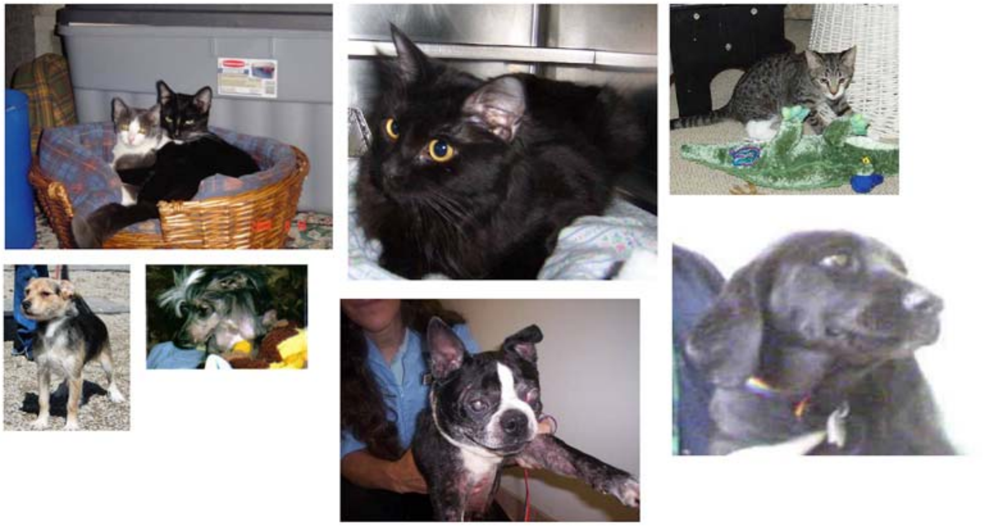

Figure 5.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples are heterogeneous in size, appearance, and so on.

Unsurprisingly, the dogs-versus-cats Kaggle competition in 2013 was won by entrants who used convnets. The best entries achieved up to 95% accuracy. In this example, you'll get fairly close to this accuracy (in the next section), even though you'll train your models on less than 10% of the data that was available to the competitors.

This dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543 MB (compressed). After downloading and uncompressing it, you'll create a new dataset containing three subsets: a training set with 1,000 samples of each class, a validation set with 500 samples of each class, and a test set with 500 samples of each class.

Following is the code to do this.

Listing 5.4 Copying images to training, validation, and test directories

Path to the directory where the original dataset was uncompressed

Directory where you'll store your smaller dataset

```
import os, shutil

original_dataset_dir = '/Users/fchollet/Downloads/kaggle_original_data'

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
os.mkdir(base_dir)
```

Directories for the training, validation, and test splits

```
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)
```

Directory with training cat pictures

```
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)
```

Directory with training dog pictures

```
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)
```

Directory with validation cat pictures

```
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)
```

Directory with validation dog pictures

```
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)
```

Directory with test cat pictures

```
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)
```

Directory with test dog pictures

```
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)
```

Copies the first 1,000 cat images to train_cats_dir

```
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)
```

Copies the next 500 cat images to validation_cats_dir

```
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)
```

Copies the next 500 cat images to test_cats_dir

```
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)
```

```
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

Copies the first 1,000 dog images to train_dogs_dir

```
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

Copies the next 500 dog images to validation_dogs_dir

```
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

Copies the next 500 dog images to test_dogs_dir

As a sanity check, let's count how many pictures are in each training split (train/validation/test):

```
>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
>>> print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500
>>> print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500
>>> print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500
>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500
```

So you do indeed have 2,000 training images, 1,000 validation images, and 1,000 test images. Each split contains the same number of samples from each class: this is a balanced binary-classification problem, which means classification accuracy will be an appropriate measure of success.

5.2.3 Building your network

You built a small convnet for MNIST in the previous example, so you should be familiar with such convnets. You'll reuse the same general structure: the convnet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

But because you're dealing with bigger images and a more complex problem, you'll make your network larger, accordingly: it will have one more `Conv2D` + `MaxPooling2D` stage. This serves both to augment the capacity of the network and to further reduce the size of the feature maps so they aren't overly large when you reach the `Flatten` layer. Here, because you start from inputs of size 150 × 150 (a somewhat arbitrary choice), you end up with feature maps of size 7 × 7 just before the `Flatten` layer.

NOTE The depth of the feature maps progressively increases in the network (from 32 to 128), whereas the size of the feature maps decreases (from 148 × 148 to 7 × 7). This is a pattern you'll see in almost all convnets.

Because you're attacking a binary-classification problem, you'll end the network with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the network is looking at one class or the other.

Listing 5.5 Instantiating a small convnet for dogs vs. cats classification

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Let's look at how the dimensions of the feature maps change with every successive layer:

```
>>> model.summary()

Layer (type)                      Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                 (None, 148, 148, 32)  896
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)     (None, 74, 74, 32)    0
________________________________________________________________
conv2d_2 (Conv2D)                 (None, 72, 72, 64)    18496
________________________________________________________________
maxpooling2d_2 (MaxPooling2D)     (None, 36, 36, 64)    0
________________________________________________________________
conv2d_3 (Conv2D)                 (None, 34, 34, 128)   73856
________________________________________________________________
maxpooling2d_3 (MaxPooling2D)     (None, 17, 17, 128)   0
________________________________________________________________
conv2d_4 (Conv2D)                 (None, 15, 15, 128)   147584
________________________________________________________________
maxpooling2d_4 (MaxPooling2D)     (None, 7, 7, 128)     0
________________________________________________________________
flatten_1 (Flatten)               (None, 6272)          0
________________________________________________________________
dense_1 (Dense)                   (None, 512)           3211776
________________________________________________________________
```

```
dense_2 (Dense)                  (None, 1)             513
================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

For the compilation step, you'll go with the `RMSprop` optimizer, as usual. Because you ended the network with a single sigmoid unit, you'll use binary crossentropy as the loss (as a reminder, check out table 4.1 for a cheatsheet on what loss function to use in various situations).

Listing 5.6 Configuring the model for training

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

5.2.4 *Data preprocessing*

As you know by now, data should be formatted into appropriately preprocessed floating-point tensors before being fed into the network. Currently, the data sits on a drive as JPEG files, so the steps for getting it into the network are roughly as follows:

1. Read the picture files.
2. Decode the JPEG content to RGB grids of pixels.
3. Convert these into floating-point tensors.
4. Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

It may seem a bit daunting, but fortunately Keras has utilities to take care of these steps automatically. Keras has a module with image-processing helper tools, located at `keras.preprocessing.image`. In particular, it contains the class `ImageDataGenerator`, which lets you quickly set up Python generators that can automatically turn image files on disk into batches of preprocessed tensors. This is what you'll use here.

Listing 5.7 Using `ImageDataGenerator` to read images from directories

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150)
        batch_size=20,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
```

Rescales all images by 1/255

Target directory

Resizes all images to 150 × 150

Because you use binary_crossentropy loss, you need binary labels.

```
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

Understanding Python generators

A *Python generator* is an object that acts as an iterator: it's an object you can use with the `for ... in` operator. Generators are built using the `yield` operator.

Here is an example of a generator that yields integers:

```
def generator():
    i = 0
    while True:
        i += 1
        yield i

for item in generator():
    print(item)
    if item > 4:
        break
```

It prints this:

```
1
2
3
4
5
```

Let's look at the output of one of these generators: it yields batches of 150 × 150 RGB images (shape `(20, 150, 150, 3)`) and binary labels (shape `(20,)`). There are 20 samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it loops endlessly over the images in the target folder. For this reason, you need to `break` the iteration loop at some point:

```
>>> for data_batch, labels_batch in train_generator:
>>>     print('data batch shape:', data_batch.shape)
>>>     print('labels batch shape:', labels_batch.shape)
>>>     break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Let's fit the model to the data using the generator. You do so using the `fit_generator` method, the equivalent of `fit` for data generators like this one. It expects as its first argument a Python generator that will yield batches of inputs and targets indefinitely, like this one does. Because the data is being generated endlessly, the Keras model needs to know how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator—that is, after having run for steps_per_epoch gradient descent steps—the fitting process will go to the next epoch. In this case, batches are 20 samples, so it will take 100 batches until you see your target of 2,000 samples.

When using fit_generator, you can pass a validation_data argument, much as with the fit method. It's important to note that this argument is allowed to be a data generator, but it could also be a tuple of Numpy arrays. If you pass a generator as validation_data, then this generator is expected to yield batches of validation data endlessly; thus you should also specify the validation_steps argument, which tells the process how many batches to draw from the validation generator for evaluation.

Listing 5.8 Fitting the model using a batch generator

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50)
```

It's good practice to always save your models after training.

Listing 5.9 Saving the model

```
model.save('cats_and_dogs_small_1.h5')
```

Let's plot the loss and accuracy of the model over the training and validation data during training (see figures 5.9 and 5.10).

Listing 5.10 Displaying curves of loss and accuracy during training

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

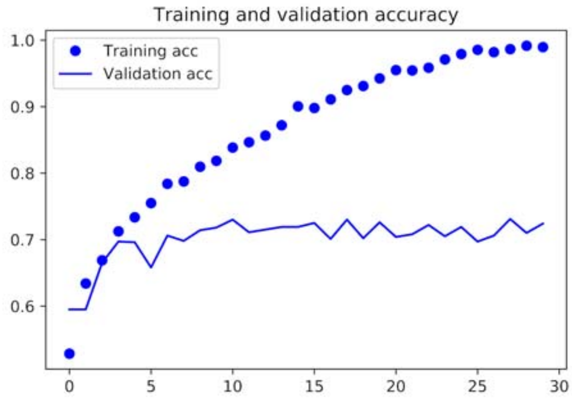

Figure 5.9 Training and validation accuracy

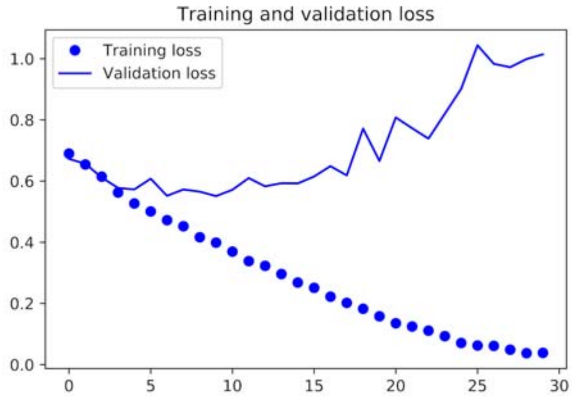

Figure 5.10 Training and validation loss

These plots are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy stalls at 70–72%. The validation loss reaches its minimum after only five epochs and then stalls, whereas the training loss keeps decreasing linearly until it reaches nearly 0.

Because you have relatively few training samples (2,000), overfitting will be your number-one concern. You already know about a number of techniques that can help mitigate overfitting, such as dropout and weight decay (L2 regularization). We're now going to work with a new one, specific to computer vision and used almost universally when processing images with deep-learning models: *data augmentation.*

5.2.5 Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering you unable to train a model that can generalize to new data. Given infinite data, your model would be exposed to every possible aspect of the data distribution at hand: you would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by *augmenting* the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by the `ImageDataGenerator` instance. Let's get started with an example.

Listing 5.11 Setting up a data augmentation configuration via `ImageDataGenerator`

```
datagen = ImageDataGenerator(
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over this code:

- `rotation_range` is a value in degrees (0–180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half the images horizontally—relevant when there are no assumptions of horizontal asymmetry (for example, real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Let's look at the augmented images (see figure 5.11).

Listing 5.12 Displaying some randomly augmented training images

```
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for
     fname in os.listdir(train_cats_dir)]

img_path = fnames[3]

img = image.load_img(img_path, target_size=(150, 150))
```

Module with image-preprocessing utilities

Chooses one image to augment

Reads the image and resizes it

```
x = image.img_to_array(img)

x = x.reshape((1,) + x.shape)

i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```

Converts it to a Numpy array with shape (150, 150, 3)

Reshapes it to (1, 150, 150, 3)

Generates batches of randomly transformed images. Loops indefinitely, so you need to break the loop at some point!

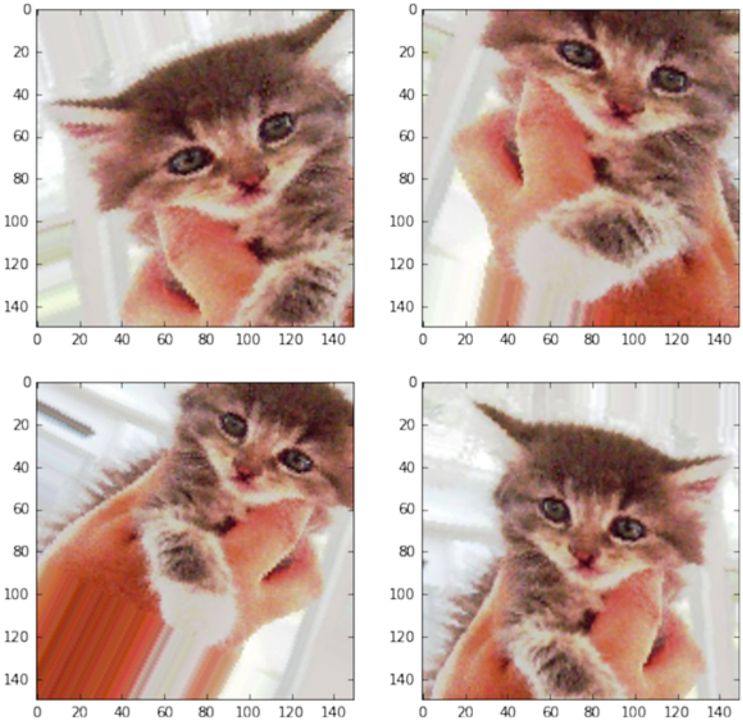

Figure 5.11 Generation of cat pictures via random data augmentation

If you train a new network using this data-augmentation configuration, the network will never see the same input twice. But the inputs it sees are still heavily intercorrelated, because they come from a small number of original images—you can't produce new information, you can only remix existing information. As such, this may not be enough to completely get rid of overfitting. To further fight overfitting, you'll also add a Dropout layer to your model, right before the densely connected classifier.

Listing 5.13 Defining a new convnet that includes dropout

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Let's train the network using data augmentation and dropout.

Listing 5.14 Training the convnet using data-augmentation generators

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

Note that the validation data shouldn't be augmented!

Target directory

Resizes all images to 150 × 150

Because you use binary_crossentropy loss, you need binary labels.

Let's save the model—you'll use it in section 5.4.

Listing 5.15 Saving the model

```
model.save('cats_and_dogs_small_2.h5')
```

And let's plot the results again: see figures 5.12 and 5.13. Thanks to data augmentation and dropout, you're no longer overfitting: the training curves are closely tracking the validation curves. You now reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

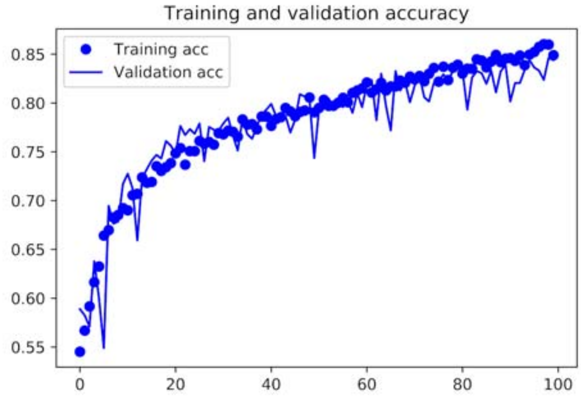

Figure 5.12 Training and validation accuracy with data augmentation

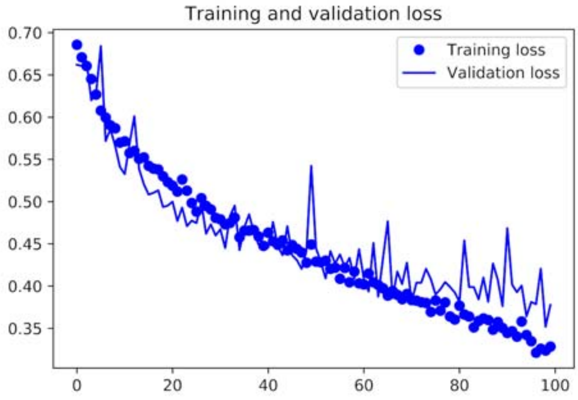

Figure 5.13 Training and validation loss with data augmentation

By using regularization techniques even further, and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), you may be able to get an even better accuracy, likely up to 86% or 87%. But it would prove difficult to go any higher just by training your own convnet from scratch, because you have so little data to work with. As a next step to improve your accuracy on this problem, you'll have to use a pretrained model, which is the focus of the next two sections.

5.4 Visualizing what convnets learn

It's often said that deep-learning models are "black boxes": learning representations that are difficult to extract and present in a human-readable form. Although this is partially true for certain types of deep-learning models, it's definitely not true for convnets. The representations learned by convnets are highly amenable to visualization, in large part because they're *representations of visual concepts*. Since 2013, a wide array of techniques have been developed for visualizing and interpreting these representations. We won't survey all of them, but we'll cover three of the most accessible and useful ones:

- *Visualizing intermediate convnet outputs (intermediate activations)*—Useful for understanding how successive convnet layers transform their input, and for getting a first idea of the meaning of individual convnet filters.
- *Visualizing convnets filters*—Useful for understanding precisely what visual pattern or concept each filter in a convnet is receptive to.
- *Visualizing heatmaps of class activation in an image*—Useful for understanding which parts of an image were identified as belonging to a given class, thus allowing you to localize objects in images.

For the first method—activation visualization—you'll use the small convnet that you trained from scratch on the dogs-versus-cats classification problem in section 5.2. For the next two methods, you'll use the VGG16 model introduced in section 5.3.

5.4.1 Visualizing intermediate activations

Visualizing intermediate activations consists of displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input (the output of a layer is often called its *activation*, the output of the activation function). This gives a view into how an input is decomposed into the different filters learned by the network. You want to visualize feature maps with three dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently 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')
>>> model.summary()  <1> As a reminder.
________________________________________________________________
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
```

Next, you'll get an input image—a picture of a cat, not part of the images the network was trained on.

Listing 5.25 Preprocessing a single image

```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.jpg'

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)
img_tensor /= 255.

<1> Its shape is (1, 150, 150, 3)
print(img_tensor.shape)
```

Preprocesses the image into a 4D tensor

Remember that the model was trained on inputs that were preprocessed this way.

Let's display the picture (see figure 5.24).

Listing 5.26 Displaying the test picture

```
import matplotlib.pyplot as plt

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

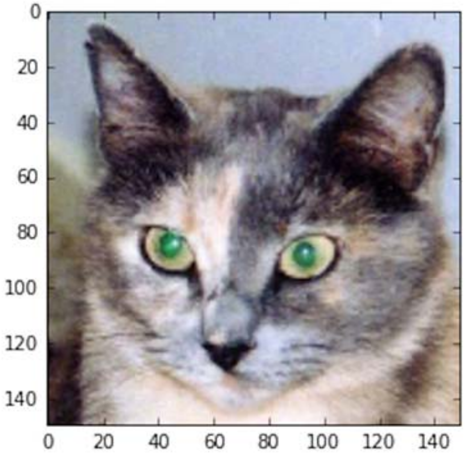

Figure 5.24 The test cat picture

In order to extract the feature maps you want to look at, you'll create a Keras model that takes batches of images as input, and outputs the activations of all convolution and pooling layers. To do this, you'll use the Keras class `Model`. A model is instantiated using two arguments: an input tensor (or list of input tensors) and an output tensor (or list of output tensors). The resulting class is a Keras model, just like the `Sequential` models you're familiar with, mapping the specified inputs to the specified outputs. What sets the `Model` class apart is that it allows for models with multiple outputs, unlike `Sequential`. For more information about the `Model` class, see section 7.1.

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

```
from keras import models

layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

Extracts the outputs of the top eight layers

Creates a model that will return these outputs, given the model input

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you've encountered a multi-output model in this book: until now, the models you've seen have had exactly one input and one output. In the general case, a model can have any number of inputs and outputs. This one has one input and eight outputs: one output per layer activation.

Listing 5.28 Running the model in predict mode

```
activations = activation_model.predict(img_tensor)
```

Returns a list of five Numpy arrays: one array per layer activation

For instance, this is the 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)
```

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')
```

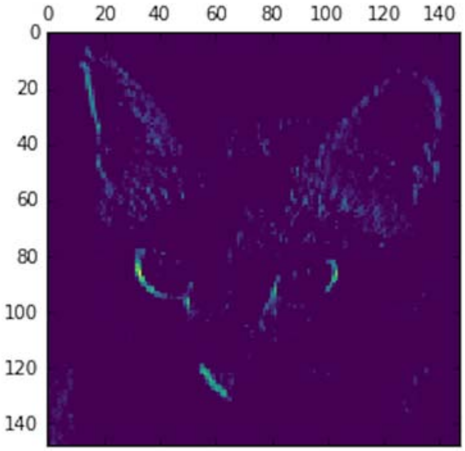

Figure 5.25 Fourth channel of the activation of the first layer on the test cat picture

This channel appears to encode a diagonal edge detector. Let’s try the seventh channel (see figure 5.26)—but note that your own channels may vary, because 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')
```

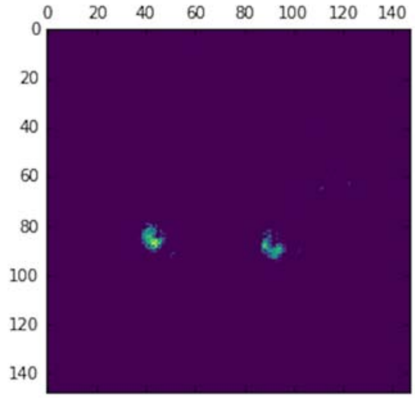

Figure 5.26 Seventh channel of the activation of the first layer on the test cat picture

This one looks like a “bright green dot” detector, useful to encode cat eyes. At this point, let’s plot a complete visualization of all the activations in the network (see figure 5.27). You’ll extract and plot every channel in each of the eight activation maps, and you’ll stack the results in one big image tensor, with channels stacked side by side.

Listing 5.31 Visualizing every channel in every intermediate activation

```
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]

    size = layer_activation.shape[1]

    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            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

    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')
```

Names of the layers, so you can have them as part of your plot

Displays the feature maps

Number of features in the feature map

The feature map has shape (l, size, size, n_features).

Tiles the activation channels in this matrix

Tiles each filter into a big horizontal grid

Post-processes the feature to make it visually palatable

Displays the grid

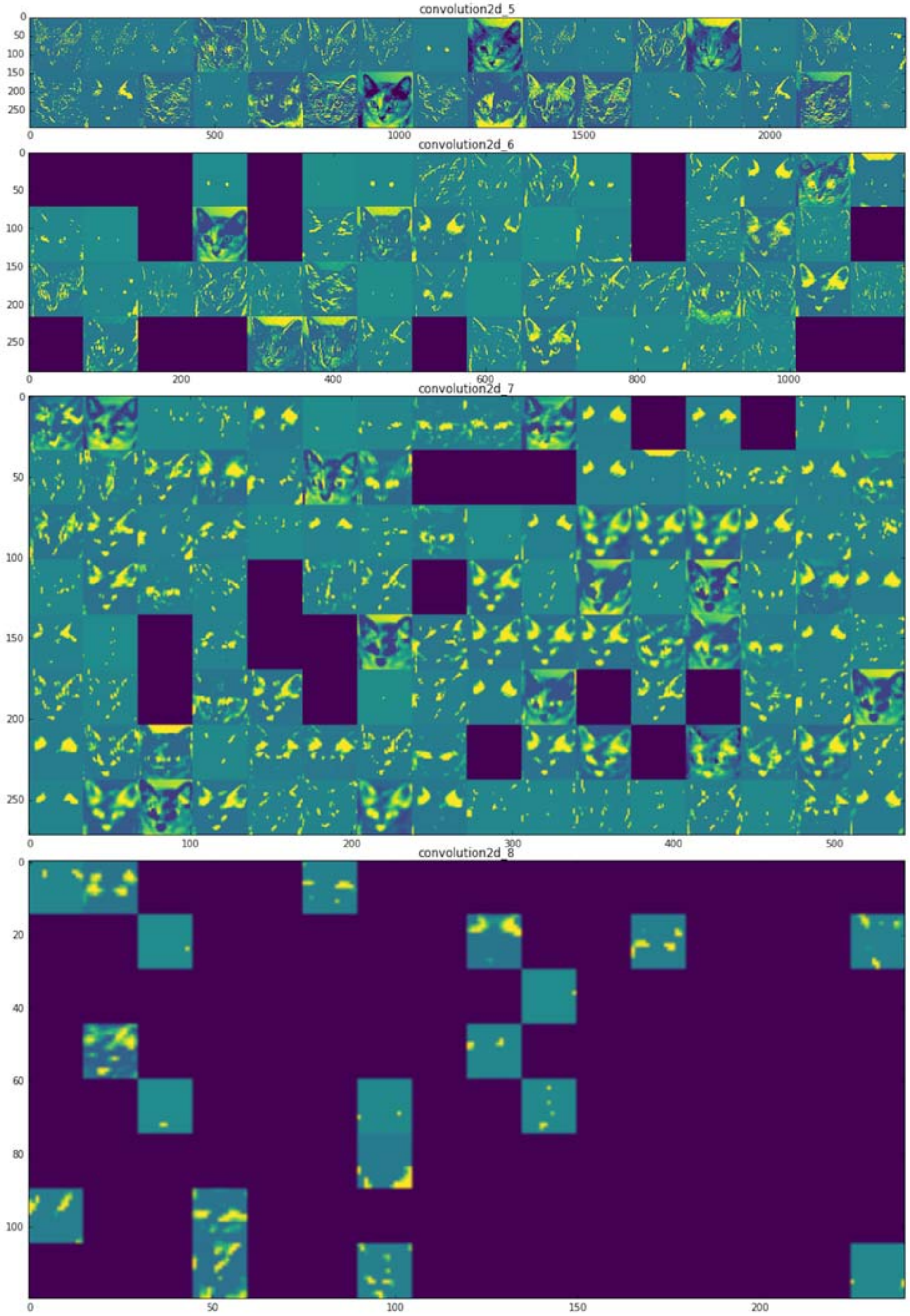

Figure 5.27 Every channel of every layer activation on the test cat picture

There are a few things to note here:

- The first layer acts as a collection of various edge detectors. At that stage, the activations retain almost all of the information present in the initial picture.
- As you go higher, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts such as “cat ear” and “cat eye.” Higher presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.
- The sparsity of the activations increases with the depth of the layer: in the first layer, all filters are activated by the input image; but in the following layers, more and more filters are blank. This means the pattern encoded by the filter isn’t found in the input image.

We have just evidenced an important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer become increasingly abstract with the depth of the layer. The activations of higher layers carry less and less information about the specific input being seen, and more and more information about the target (in this case, the class of the image: cat or dog). A deep neural network effectively acts as an *information distillation pipeline*, with raw data going in (in this case, RGB pictures) and being repeatedly transformed so that irrelevant information is filtered out (for example, the specific visual appearance of the image), and useful information is magnified and refined (for example, the class of the image).

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (bicycle, tree) but can’t remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from memory, chances are you couldn’t get it even remotely right, even though you’ve seen thousands of bicycles in your lifetime (see, for example, figure 5.28). Try it right now: this effect is absolutely real. You brain has learned to completely abstract its visual input—to transform it into high-level visual concepts while filtering out irrelevant visual details—making it tremendously difficult to remember how things around you look.

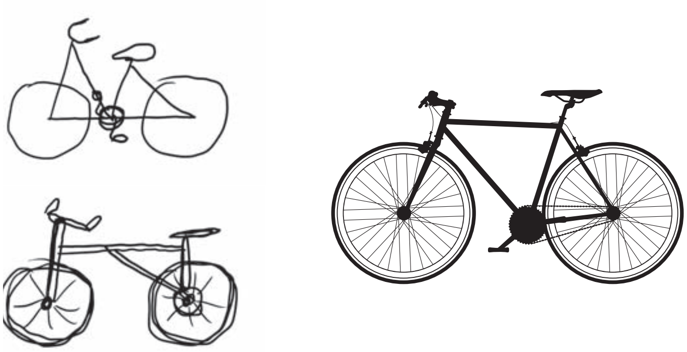

Figure 5.28 Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.

5.4.2 Visualizing convnet filters

Another easy way to inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to. This can be done with *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: you'll build a loss function that maximizes the value of a given filter in a given convolution layer, and then you'll 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

```
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])
```

To implement gradient descent, you'll need the gradient of this loss with respect to the model's input. To do this, you'll use the `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]
```

The call to gradients returns a list of tensors (of size 1 in this case). Hence, you keep only the first element—which is a tensor.

A non-obvious trick to use to help the gradient-descent process go smoothly is to 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)
```

Add 1e–5 before dividing to avoid accidentally dividing by 0.

Now you need a way to compute the value of the loss tensor and the gradient tensor, given an input image. You can define a Keras backend function to do this: `iterate` is 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))])
```

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

Listing 5.36 Loss maximization via stochastic gradient descent

Starts from a gray image with some noise

```
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])

    input_img_data += grads_value * step
```

Magnitude of each gradient update

Runs gradient ascent for 40 steps

Computes the loss value and gradient value

Adjusts the input image in the direction that maximizes the loss

The resulting image tensor is a floating-point tensor of shape `(1, 150, 150, 3)`, with values that may not be integers within [0, 255]. Hence, you need to postprocess this tensor to turn it into a displayable image. You do so with the following straightforward utility function.

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

```
def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    x += 0.5
    x = np.clip(x, 0, 1)

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

Normalizes the tensor: centers on 0, ensures that std is 0.1

Clips to [0, 1]

Converts to an RGB array

Now you have all the pieces. 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):
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output[:, :, :, filter_index])

    grads = K.gradients(loss, model.input)[0]

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

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

    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.

    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)
```

Builds a loss function that maximizes the activation of the nth filter of the layer under consideration

Computes the gradient of the input picture with regard to this loss

Normalization trick: normalizes the gradient

Returns the loss and grads given the input picture

Starts from a gray image with some noise

Runs gradient ascent for 40 steps

Let's try it (see figure 5.29):

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

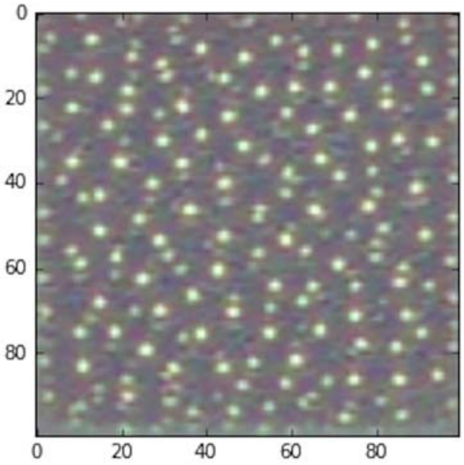

Figure 5.29 Pattern that the zeroth channel in layer `block3_conv1` responds to maximally

It seems that filter 0 in layer `block3_conv1` is responsive to a polka-dot pattern. Now the fun part: you can start visualizing every filter in every layer. For simplicity, you'll only look at the first 64 filters in each layer, and you'll only look at the first layer of each convolution block (`block1_conv1`, `block2_conv1`, `block3_conv1`, `block4_conv1`, `block5_conv1`). You'll arrange the outputs on an 8 × 8 grid of 64 × 64 filter patterns, with some black margins between each filter pattern (see figures 5.30–5.33).

Listing 5.39 Generating a grid of all filter response patterns in a layer

```
layer_name = 'block1_conv1'
size = 64
margin = 5

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

for i in range(8):
    for j in range(8):
        filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

        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

plt.figure(figsize=(20, 20))
plt.imshow(results)
```

Empty (black) image to store results

Iterates over the rows of the results grid

Iterates over the columns of the results grid

Generates the pattern for filter i + (j * 8) in layer_name

Puts the result in the square (i, j) of the results grid

Displays the results grid

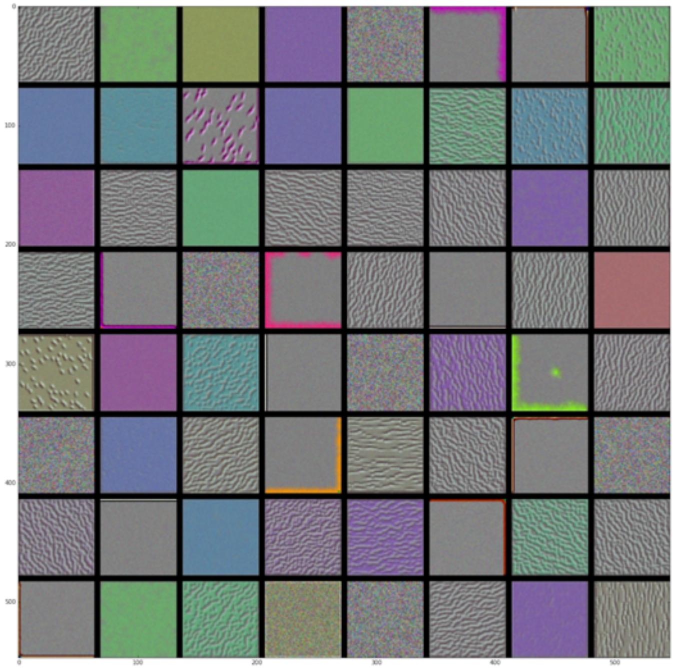

Figure 5.30 Filter patterns for layer `block1_conv1`

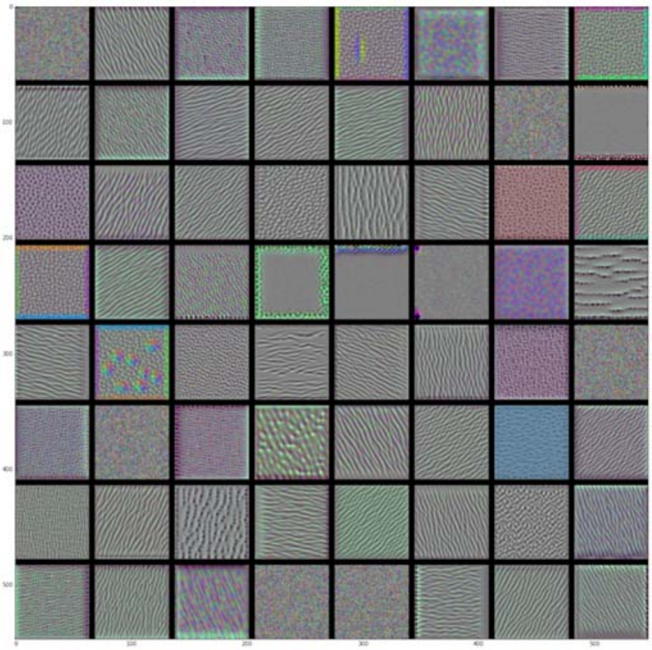

Figure 5.31 Filter patterns for layer `block2_conv1`

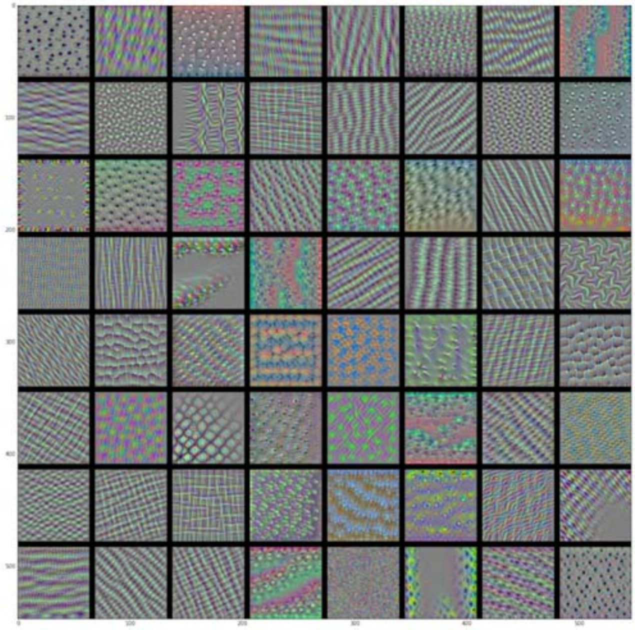

Figure 5.32 Filter patterns for layer `block3_conv1`

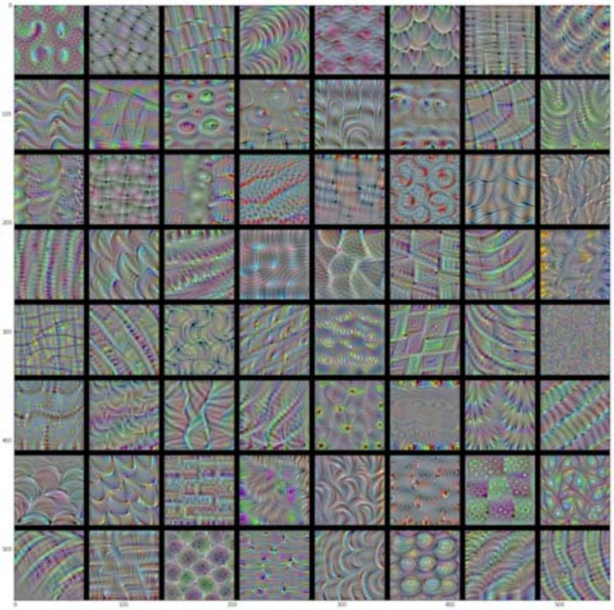

Figure 5.33 Filter patterns for layer `block4_conv1`

These filter visualizations tell you a lot about how convnet layers see the world: each layer in a convnet learns a collection of filters such that their inputs can be expressed as a combination of the filters. This is similar to how the Fourier transform decomposes signals onto a bank of cosine functions. The filters in these convnet filter banks get increasingly complex and refined as you go higher in the model:

- The filters from the first layer in the model (`block1_conv1`) encode simple directional edges and colors (or colored edges, in some cases).
- The filters from `block2_conv1` encode simple textures made from combinations of edges and colors.
- The filters in higher layers begin to resemble textures found in natural images: feathers, eyes, leaves, and so on.

5.4.3 *Visualizing heatmaps of class activation*

I'll introduce one more visualization technique: one that is useful for understanding which parts of a given image led a convnet to its final classification decision. This is helpful for debugging the decision process of a convnet, particularly in the case of a classification mistake. It also allows you to locate specific objects in an image.

This general category of techniques is called *class activation map* (CAM) visualization, and it consists of producing heatmaps of class activation over input images. A class activation heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class under consideration. For instance, given an image fed into a dogs-versus-cats convnet, CAM visualization allows you to generate a heatmap for the class "cat," indicating how cat-like different parts of the image are, and also a heatmap for the class "dog," indicating how dog-like parts of the image are.

The specific implementation you'll use is the one described in "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization."[2] It's very simple: it consists of taking the output feature map of a convolution layer, given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel. Intuitively, one way to understand this trick is that you're weighting a spatial map of "how intensely the input image activates different channels" by "how important each channel is with regard to the class," resulting in a spatial map of "how intensely the input image activates the class."

We'll demonstrate this technique using the pretrained VGG16 network again.

Listing 5.40 Loading the VGG16 network with pretrained weights

```
from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet')
```

Note that you include the densely connected classifier on top; in all previous cases, you discarded it.

Consider the image of two African elephants shown in figure 5.34 (under a Creative Commons license), possibly a mother and her calf, strolling on the savanna. Let's convert this image into something the VGG16 model can read: the model was trained on images of size 224 × 244, preprocessed according to a few rules that are packaged in the utility function `keras.applications.vgg16.preprocess_input`. So you need to load the image, resize it to 224 × 224, convert it to a Numpy `float32` tensor, and apply these preprocessing rules.

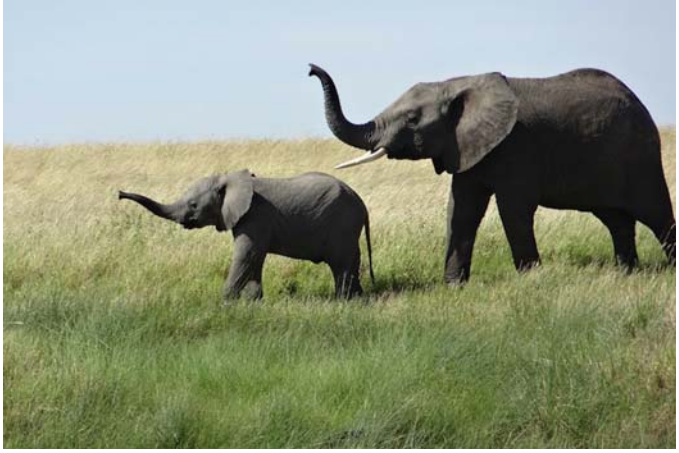

Figure 5.34 Test picture of African elephants

[2] Ramprasaath R. Selvaraju et al., arXiv (2017), https://arxiv.org/abs/ 1610.02391.

Listing 5.41 Preprocessing an input image for VGG16

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

img_path = '/Users/fchollet/Downloads/creative_commons_elephant.jpg'

img = image.load_img(img_path, target_size=(224, 224))

x = image.img_to_array(img)

x = np.expand_dims(x, axis=0)

x = preprocess_input(x)
```

Local path to the target image

Python Imaging Library (PIL) image of size 224 × 224

float32 Numpy array of shape (224, 224, 3)

Adds a dimension to transform the array into a batch of size (1, 224, 224, 3)

Preprocesses the batch (this does channel-wise color normalization)

You can now run the pretrained network on the image and decode its prediction vector back to a human-readable format:

```
>>> preds = model.predict(x)
>>> print('Predicted:', decode_predictions(preds, top=3)[0])
Predicted:', [(u'n02504458', u'African_elephant', 0.92546833),
(u'n01871265', u'tusker', 0.070257246),
(u'n02504013', u'Indian_elephant', 0.0042589349)]
```

The top three classes predicted for this image are as follows:

- African elephant (with 92.5% probability)
- Tusker (with 7% probability)
- Indian elephant (with 0.4% probability)

The network has recognized the image as containing an undetermined quantity of African elephants. The entry in the prediction vector that was maximally activated is the one corresponding to the "African elephant" class, at index 386:

```
>>> np.argmax(preds[0])
386
```

To visualize which parts of the image are the most African elephant–like, let's set up the Grad-CAM process.

Listing 5.42 Setting up the Grad-CAM algorithm

```
african_e66lephant_output = model.output[:, 386]

last_conv_layer = model.get_layer('block5_conv3')
```

"African elephant" entry in the prediction vector

Output feature map of the block5_conv3 layer, the last convolutional layer in VGG16

Gradient of the "African elephant" class with regard to the output feature map of block5_conv3

Vector of shape (512,), where each entry is the mean intensity of the gradient over a specific feature-map channel

```
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]

pooled_grads = K.mean(grads, axis=(0, 1, 2))

iterate = K.function([model.input],
                     [pooled_grads, last_conv_layer.output[0]])

pooled_grads_value, conv_layer_output_value = iterate([x])

for i in range(512):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

heatmap = np.mean(conv_layer_output_value, axis=-1)
```

Values of these two quantities, as Numpy arrays, given the sample image of two elephants

The channel-wise mean of the resulting feature map is the heatmap of the class activation.

Multiplies each channel in the feature-map array by "how important this channel is" with regard to the "elephant" class

Lets you access the values of the quantities you just defined: pooled_grads and the output feature map of block5_conv3, given a sample image

For visualization purposes, you'll also normalize the heatmap between 0 and 1. The result is shown in figure 5.35.

Listing 5.43 Heatmap post-processing

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

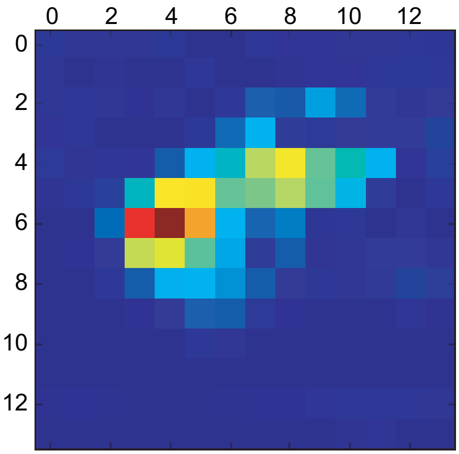

Figure 5.35 African elephant class activation heatmap over the test picture

Finally, you'll use OpenCV to generate an image that superimposes the original image on the heatmap you just obtained (see figure 5.36).

Listing 5.44 Superimposing the heatmap with the original picture

```
import cv2

img = cv2.imread(img_path)

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

heatmap = np.uint8(255 * heatmap)

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

superimposed_img = heatmap * 0.4 + img

cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img)
```

Uses cv2 to load the original image

Resizes the heatmap to be the same size as the original image

Converts the heatmap to RGB

0.4 here is a heatmap intensity factor.

Applies the heatmap to the original image

Saves the image to disk

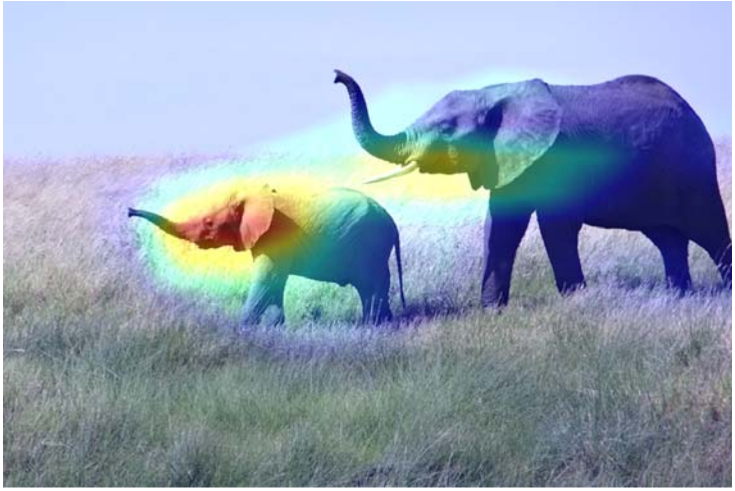

Figure 5.36 Superimposing the class activation heatmap on the original picture

This visualization technique answers two important questions:

- Why did the network think this image contained an African elephant?
- Where is the African elephant located in the picture?

In particular, it's interesting to note that the ears of the elephant calf are strongly activated: this is probably how the network can tell the difference between African and Indian elephants.

Chapter summary

- Convnets are the best tool for attacking visual-classification problems.
- Convnets work by learning a hierarchy of modular patterns and concepts to represent the visual world.
- The representations they learn are easy to inspect—convnets are the opposite of black boxes!
- You're now capable of training your own convnet from scratch to solve an image-classification problem.
- You understand how to use visual data augmentation to fight overfitting.
- You know how to use a pretrained convnet to do feature extraction and fine-tuning.
- You can generate visualizations of the filters learned by your convnets, as well as heatmaps of class activity.