天氣與人工智慧 II

Ch.5 Deep learning for computer vision

周哲維
david10188@gmail.com

Deep learning for computer vision

- 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.

5.1 Introduction to convnets

- We’re about to dive into the theory of what convnets are and why they have been so successful at computer vision tasks.
- But first, let’s take a practical look at a simple convnet example. It uses a convnet to classify MNIST digits, a task we performed in chapter 2 using a densely connected network (our test accuracy then was 97.8%).
- Even though the convnet will be basic, its accuracy will blow out of the water that of the densely connected model from chapter 2.
- The following lines of code show you what a basic convnet looks like. It’s a stack of `Conv2D` and `MaxPooling2D` layers. You’ll see in a minute exactly what they do.

5.1 Introduction to convnets

Listing 5.1 Instantiating a small convnet

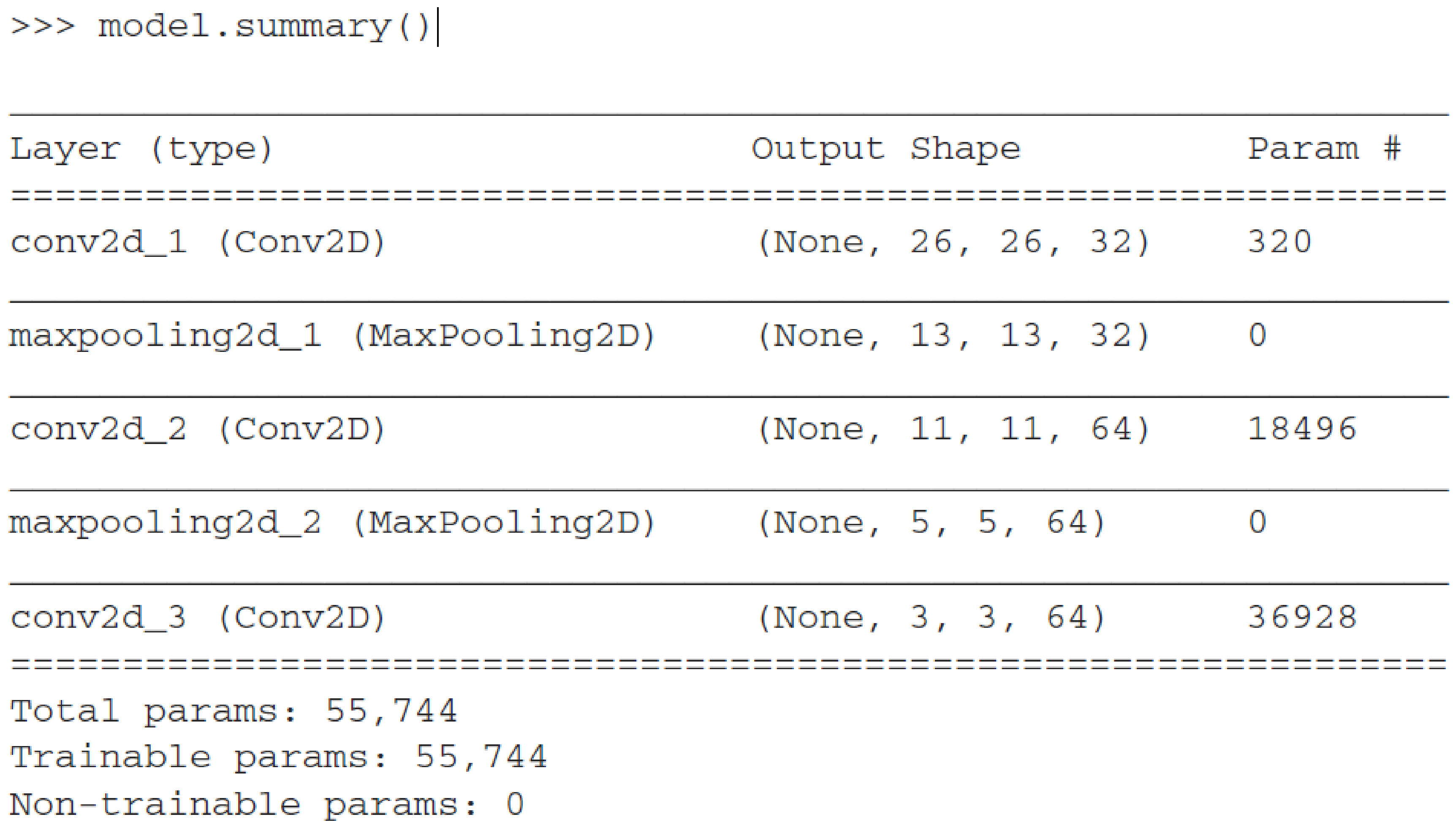
```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
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(64, (3, 3), activation='relu'))
```

- Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension).
- In this case, we'll configure the convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images.
- We'll do this by passing the argument `input_shape=(28, 28, 1)` to the first layer.

5.1 Introduction to convnets

- Let’s display the architecture of the convnet so far:



```
>>> model.summary()

________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)    320
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)    0
________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 11, 64)    18496
________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)      0
________________________________________________________________
conv2d_3 (Conv2D)                (None, 3, 3, 64)      36928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

5.1 Introduction to convnets

- You can see that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`.
- The width and height dimensions tend to shrink as you go deeper in the network. The number of channels is controlled by the first argument passed to the `Conv2D` layers (32 or 64).
- The next step is to feed the last output tensor (of shape `(3, 3, 64)`) into a densely connected classifier network like those you're already familiar with: a stack of `Dense` layers.
- These classifiers process vectors, which are 1D, whereas the current output is a 3D tensor. First we have to flatten the 3D outputs to 1D, and then add a few `Dense` layers on top.

5.1 Introduction to convnets

- These classifiers process vectors, which are 1D, whereas the current output is a 3D tensor. First we have to flatten the 3D outputs to 1D, and then add a few `Dense` layers on top.

Listing 5.2 Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64,
activation='relu'))
model.add(layers.Dense(10,
activation='softmax'))
```

- We’ll do 10-way classification, using a final layer with 10 outputs and a softmax activation.

5.1 Introduction to convnets

- Here’s what the network looks like now:

```
>>> model.summary()

Layer (type)                     Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)    320
________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)    0
________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 11, 64)    18496
________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)      0
________________________________________________________________
conv2d_3 (Conv2D)                (None, 3, 3, 64)      36928
________________________________________________________________
flatten_1 (Flatten)              (None, 576)           0
________________________________________________________________
dense_1 (Dense)                  (None, 64)            36928
________________________________________________________________
dense_2 (Dense)                  (None, 10)            650
================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

5.1 Introduction to convnets

- As you can see, the `(3, 3, 64)` outputs are flattened into vectors of shape `(576,)` before going through two `Dense` layers.
- Now, let’s train the convnet on the MNIST digits. We’ll reuse a lot of the code from the MNIST example in chapter 2.

5.1 Introduction to convnets

Listing 5.3 Training the convnet on MNIST images

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
model.compile(optimizer='rmsprop’,
             loss='categorical_crossentropy’,
             metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

5.1 Introduction to convnets

- Let's evaluate the model on the test data:

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.9908000000000001
```

- Whereas the densely connected network from chapter 2 had a test accuracy of 97.8%, the basic convnet has a test accuracy of 99.3%: we decreased the error rate by 68%(relative). Not bad!
- But why does this simple convnet work so well, compared to a densely connected model?
- To answer this, let's dive into what the `Conv2D` and `MaxPooling2D` layers do.

5.1.1 The convolution operation

- The fundamental difference between a densely connected layer and a convolution layer is this: `Dense` layers learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels), whereas convolution layers learn local patterns (see figure 5.1): in the case of images, patterns found in small 2D windows of the inputs.
- In the previous example, these windows were all 3 × 3.

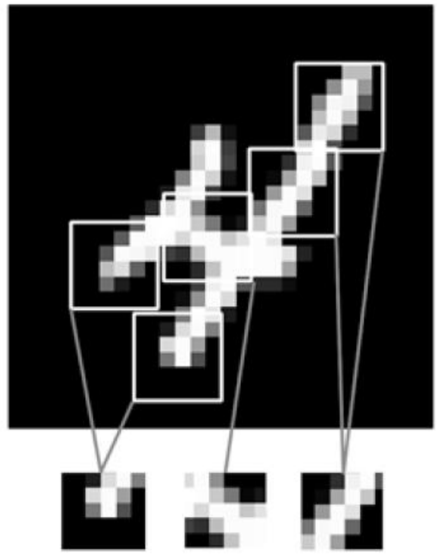


Figure 5.1 Images can be broken into local patterns such as edges, textures, and so on.

5.1.1 The convolution operation

- This key characteristic gives convnets two interesting properties:
  - *The patterns they learn are translation invariant.* After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner. A densely connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets data efficient when processing images (because *the visual world is fundamentally translation invariant*): they need fewer training samples to learn representations that have generalization power.
  - *They can learn spatial hierarchies of patterns (see figure 5.2)*. A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts (because *the visual world is fundamentally spatially hierarchical*).

5.1.1 The convolution operation

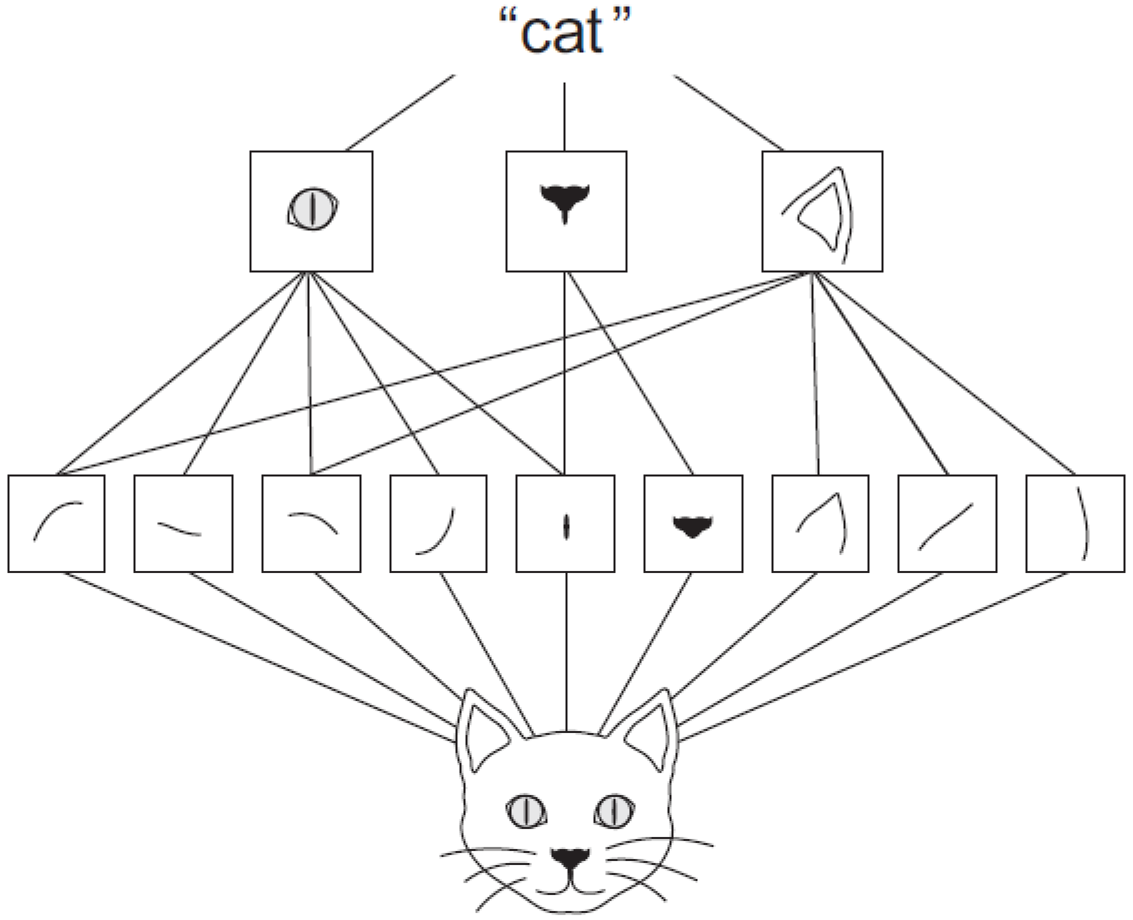


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as “cat.”

5.1.1 The convolution operation

- Convolutions operate over 3D tensors, called *feature maps*, with two spatial axes (*height* and *width*) as well as a *depth* axis (also called the *channels* axis).
- For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray).
- The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an *output feature map*.
- This output feature map is still a 3D tensor: it has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for *filters*.
- Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept “presence of a face in the input,” for instance.

5.1.1 The convolution operation

- In the MNIST example, the first convolution layer takes a feature map of size `(28, 28, 1)` and outputs a feature map of size `(26, 26, 32)`: it computes 32 filters over its input.
- Each of these 32 output channels contains a 26 × 26 grid of values, which is a *response map* of the filter over the input, indicating the response of that filter pattern at different locations in the input (see figure 5.3).
- That is what the term *feature map* means: every dimension in the depth axis is a feature (or filter), and the 2D tensor `output[:, :, n]` is the 2D spatial *map* of the response of this filter over the input.

5.1.1 The convolution operation

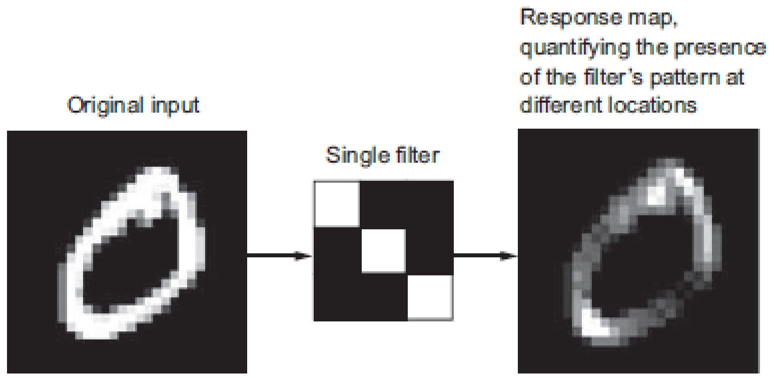

Original input
Single filter
Response map, quantifying the presence of the filter's pattern at different locations

Figure 5.3 The concept of a *response map*: a 2D map of the presence of a pattern at different locations in an input

- Convolutions are defined by two key parameters:
  - *Size of the patches extracted from the inputs*—These are typically 3 × 3 or 5 × 5. In the example, they were 3 × 3, which is a common choice.
  - *Depth of the output feature map*—The number of filters computed by the convolution. The example started with a depth of 32 and ended with a depth of 64.

5.1.1 The convolution operation

- In Keras `Conv2D` layers, these parameters are the first arguments passed to the layer: `Conv2D(output_depth, (window_height, window_width))`.
- A convolution works by *sliding* these windows of size 3 × 3 or 5 × 5 over the 3D input feature map, stopping at every possible location, and extracting the 3D patch of surrounding features (shape `(window_height, window_width, input_depth)`).
- Each such 3D patch is then transformed (via a tensor product with the same learned weight matrix, called the *convolution kernel*) into a 1D vector of shape `(output_depth,)`.
- All of these vectors are then spatially reassembled into a 3D output map of shape `(height, width, output_depth)`.

5.1.1 The convolution operation

- Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input).
- For instance, with 3 × 3 windows, the vector `output[i, j, :]` comes from the 3D patch `input[i-1:i+1, j-1:j+1, :]`.
- The full process is detailed in figure 5.4.

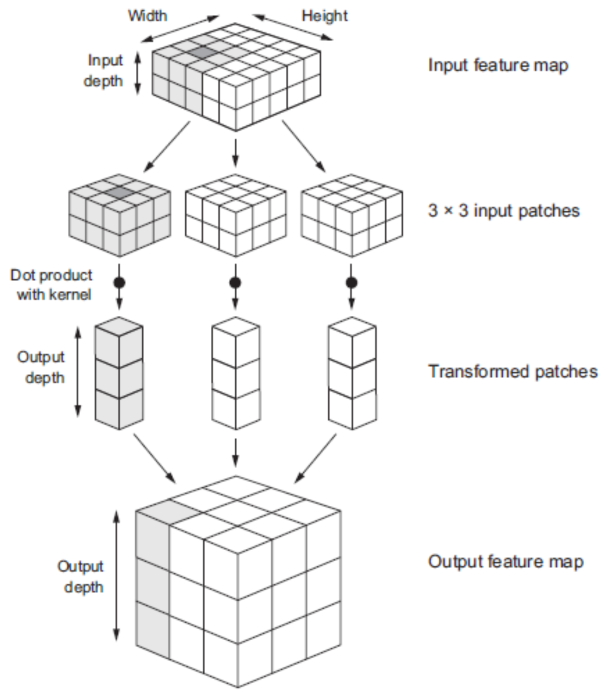


Figure 5.4 How convolution works

5.1.1 The convolution operation

- Note that the output width and height may differ from the input width and height.
- They may differ for two reasons:
    - Border effects, which can be countered by padding the input feature map
    - The use of *strides*, which I'll define in a second
- Let's take a deeper look at these notions.

5.1.1 The convolution operation

UNDERSTANDING BORDER EFFECTS AND PADDING

- Consider a 5 × 5 feature map (25 tiles total). There are only 9 tiles around which you can center a 3 × 3 window, forming a 3 × 3 grid (see figure 5.5). Hence, the output feature map will be 3 × 3.
- It shrinks a little: by exactly two tiles alongside each dimension, in this case.
- You can see this border effect in action in the earlier example: you start with 28 × 28 inputs, which become 26 × 26 after the first convolution layer.

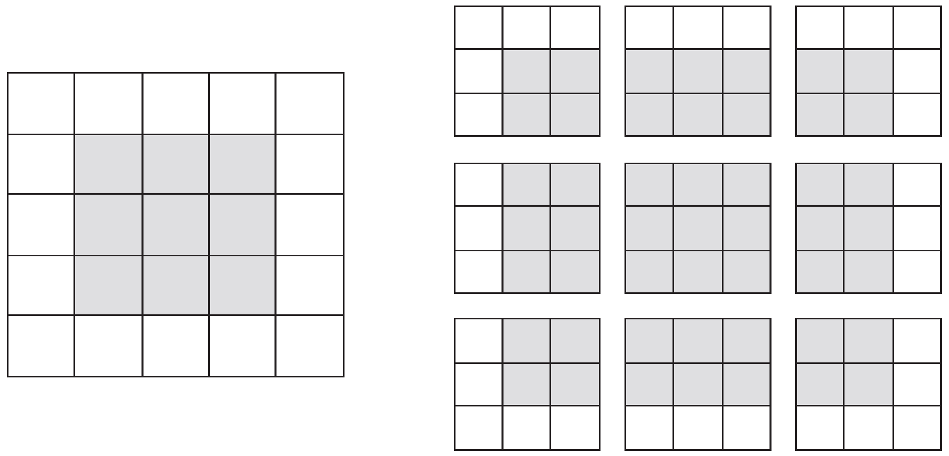


Figure 5.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

5.1.1 The convolution operation

UNDERSTANDING BORDER EFFECTS AND PADDING

- If you want to get an output feature map with the same spatial dimensions as the input, you can use *padding*.
- Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile.
- For a 3 × 3 window, you add one column on the right, one column on the left, one row at the top, and one row at the bottom. For a 5 × 5 window, you add two rows (see figure 5.6).

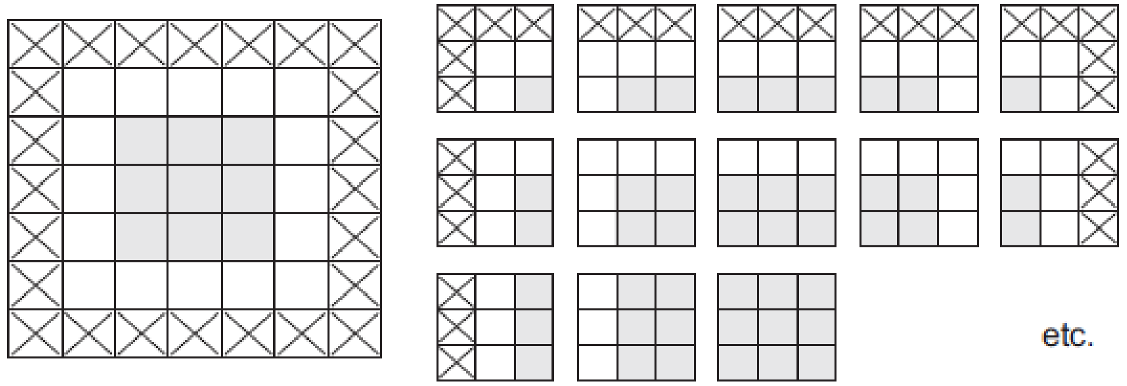


Figure 5.6 Padding a 5 × 5 input in order to be able to extract 25 3 × 3 patches

5.1.1 The convolution operation

UNDERSTANDING BORDER EFFECTS AND PADDING

- In `Conv2D` layers, padding is configurable via the `padding` argument, which takes two values:
  - `"valid"`, which means no padding (only valid window locations will be used);
  - `"same"`, which means “pad in such a way as to have an output with the same width and height as the input.”
- The `padding` argument defaults to `"valid"`.

5.1.1 The convolution operation

UNDERSTANDING CONVOLUTION STRIDES

- The other factor that can influence output size is the notion of *strides*.
- The description of convolution so far has assumed that the center tiles of the convolution windows are all contiguous.
- But the distance between two successive windows is a parameter of the convolution, called its *stride*, which defaults to 1.
- It's possible to have *strided convolutions*: convolutions with a stride higher than 1. In figure 5.7, you can see the patches extracted by a 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding).

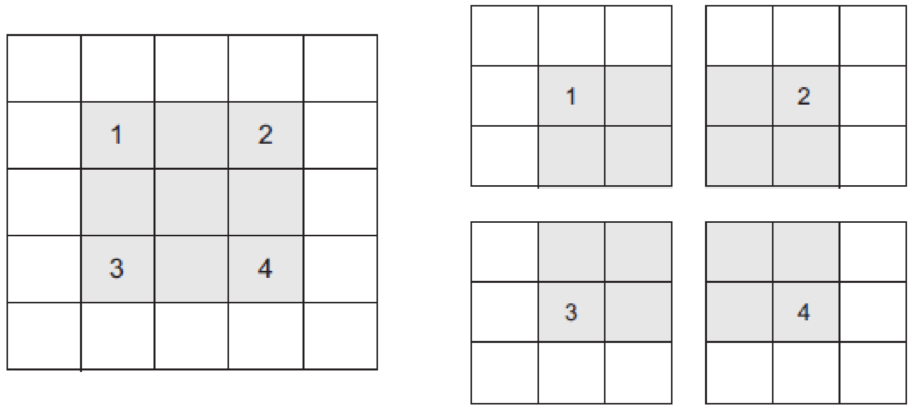


Figure 5.7 3 × 3 convolution patches with 2 × 2 strides

5.1.1 The convolution operation

UNDERSTANDING CONVOLUTION STRIDES

- Using stride 2 means the width and height of the feature map are downsampled by a factor of 2 (in addition to any changes induced by border effects).
- Strided convolutions are rarely used in practice, although they can come in handy for some types of models; it's good to be familiar with the concept.
- To downsample feature maps, instead of strides, we tend to use the *max-pooling* operation, which you saw in action in the first convnet example.

5.1.2 The max-pooling operation

- In the convnet example, you may have noticed that the size of the feature maps is halved after every `MaxPooling2D` layer.
- For instance, before the first `MaxPooling2D` layers, the feature map is 26 × 26, but the max-pooling operation halves it to 13 × 13.
- That’s the role of max pooling: to aggressively downsample feature maps, much like strided convolutions.
- Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel.
- It’s conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they’re transformed via a hardcoded `max` tensor operation.
- A big difference from convolution is that max pooling is usually done with 2 × 2 windows and stride 2, in order to downsample the feature maps by a factor of 2.

5.1.2 The max-pooling operation

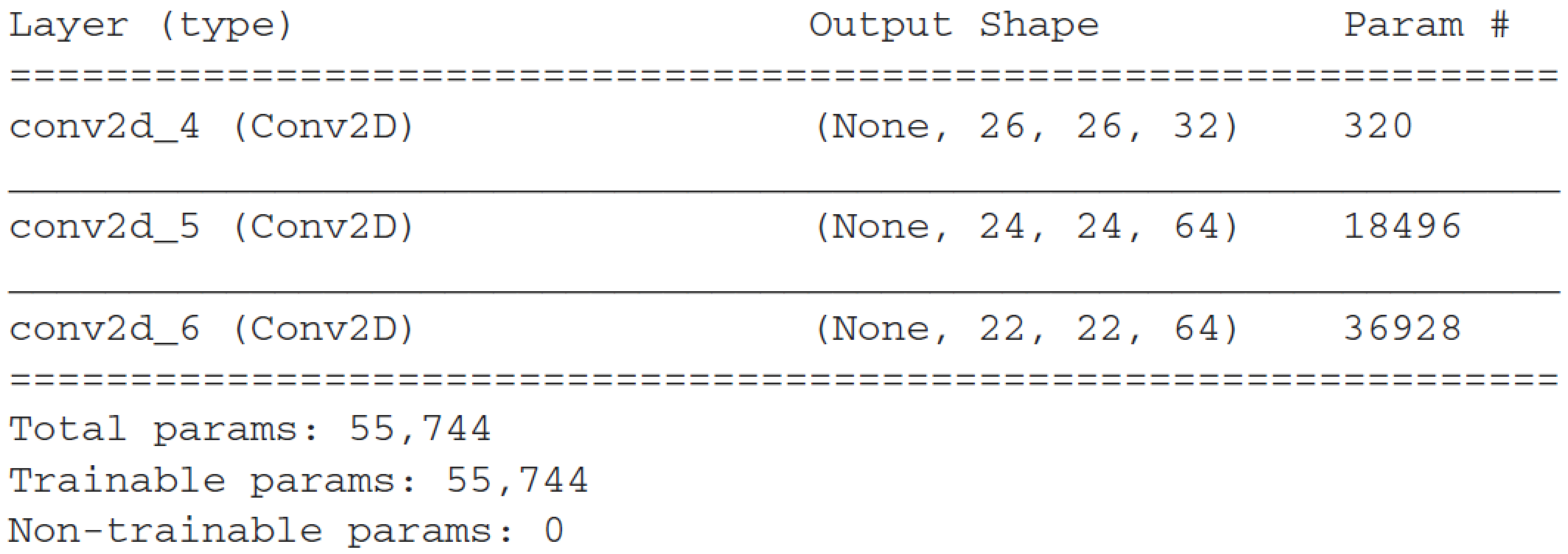
- On the other hand, convolution is typically done with 3 × 3 windows and no stride (stride 1).
- Why downsample feature maps this way? Why not remove the max-pooling layers and keep fairly large feature maps all the way up?
- Let’s look at this option. The convolutional base of the model would then look like this:

```
model_no_max_pool = models.Sequential()
model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(28, 28, 1)))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

5.1.2 The max-pooling operation

Here’s a summary of the model:

```
>>> model_no_max_pool.summary()
```



```
Layer (type)                     Output Shape          Param #
================================================================
conv2d_4 (Conv2D)                (None, 26, 26, 32)    320
________________________________________________________________
conv2d_5 (Conv2D)                (None, 24, 24, 64)    18496
________________________________________________________________
conv2d_6 (Conv2D)                (None, 22, 22, 64)    36928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

5.1.2 The max-pooling operation

- What’s wrong with this setup? Two things:
  - It isn’t conducive to learning a spatial hierarchy of features. The 3 × 3 windows in the third layer will only contain information coming from 7 × 7 windows in the initial input. The high-level patterns learned by the convnet will still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows that are 7 × 7 pixels!). We need the features from the last convolution layer to contain information about the totality of the input.
  - The final feature map has 22 × 22 × 64 = 30,976 total coefficients per sample. This is huge. If you were to flatten it to stick a `Dense` layer of size 512 on top, that layer would have 15.8 million parameters. This is far too large for such a small model and would result in intense overfitting.

5.1.2 The max-pooling operation

- In short, the reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).
- Note that max pooling isn't the only way you can achieve such downsampling. As you already know, you can also use strides in the prior convolution layer. And you can use average pooling instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the max.

5.1.2 The max-pooling operation

- But max pooling tends to work better than these alternative solutions. In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term *feature map*), and it's more informative to look at the *maximal presence* of different features than at their *average presence*.
- So the most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

5.1.2 The max-pooling operation

- At this point, you should understand the basics of convnets—feature maps, convolution, and max pooling—and you know how to build a small convnet to solve a toy problem such as MNIST digits classification.
- Now let’s move on to more useful, practical applications.

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.

5.2 Training a convnet from scratch on a small dataset

- 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.

5.2.1 The relevance of deep learning for small-data problems

- 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.

5.2.1 The relevance of deep learning for small-data problems

- Specifically, in the case of computer vision, many pretrained models (usually trained on the Image-Net 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
- The pictures are medium-resolution color JPEGs. Figure 5.8 shows some examples.

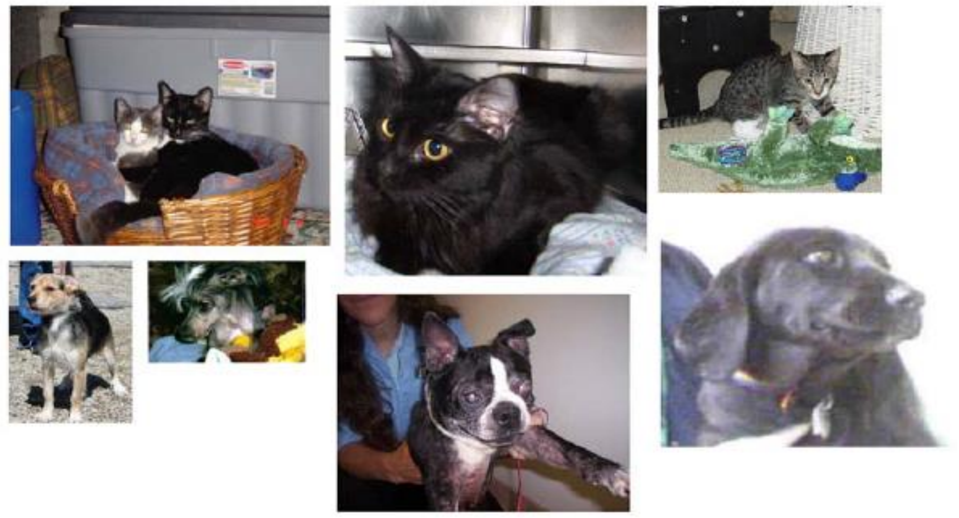


Figure 5.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples are heterogeneous in size, appearance, and so on.

5.2.2 Downloading the data

- 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.

5.2.2 Downloading the data

- Following is the code to do this.

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

```
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)
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)
```

Path to the directory where the original dataset was uncompressed

Directory where you'll store your smaller dataset

Directories for the training, validation, and test splits

5.2.2 Downloading the data

- Following is the code to do this.

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

```
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)
```

Directory with training cat pictures

Directory with training dog pictures

Directory with validation cat pictures

Directory with validation dog pictures

Directory with test cat pictures

Directory with test dog pictures

5.2.2 Downloading the data

- Following is the code to do this.

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

```
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)
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)
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)
```

Copies the first 1,000 cat images to train_cats_dir

Copies the next 500 cat images to validation_cats_dir

Copies the next 500 cat images to test_cats_dir

5.2.2 Downloading the data

- Following is the code to do this.

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

```
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)
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)
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 first 1,000 cat images to train_dogs_dir

Copies the next 500 cat images to validation_dogs_dir

Copies the next 500 cat images to test_dogs_dir

5.2.2 Downloading the data

- 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
```

5.2.2 Downloading the data

- 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.

5.2.3 Building your network

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.

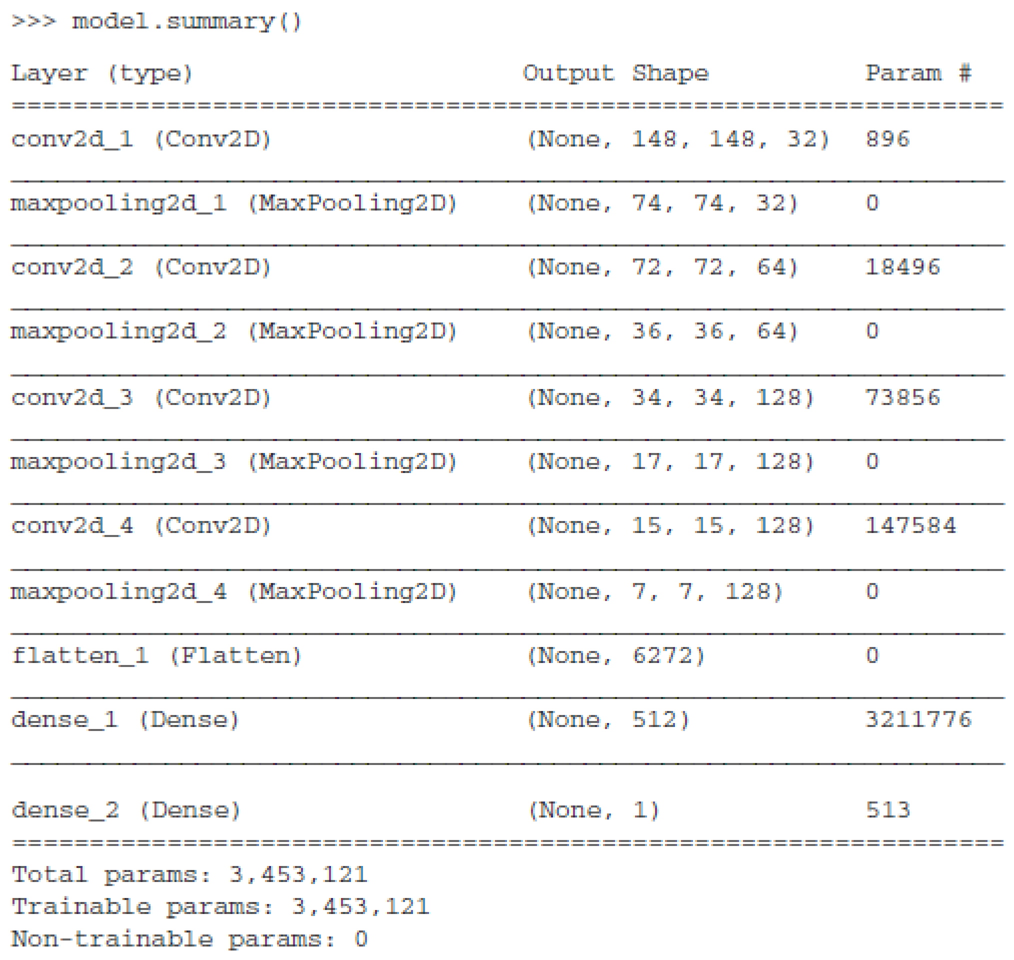
5.2.3 Building your network

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'))
```

5.2.3 Building your network

- 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
```

5.2.3 Building your network

- 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).

5.2.4 Data preprocessing

- 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.

5.2.4 Data preprocessing

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

Rescales all images by 1/255

Target directory

Resizes all images to 150 × 150

Because you use binary_crossentropy loss, you need binary labels.

5.2.4 Data preprocessing

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
```

5.2.4 Data preprocessing

- 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,)
```

5.2.4 Data preprocessing

- 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.

5.2.4 Data preprocessing

- When using `fit_generator`, you can pass a `validation_data` argument, such 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.

5.2.4 Data preprocessing

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')
```

5.2.4 Data preprocessing

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 5.4 Copying images to training, validation, and test directories

```
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()
```

5.2.4 Data preprocessing

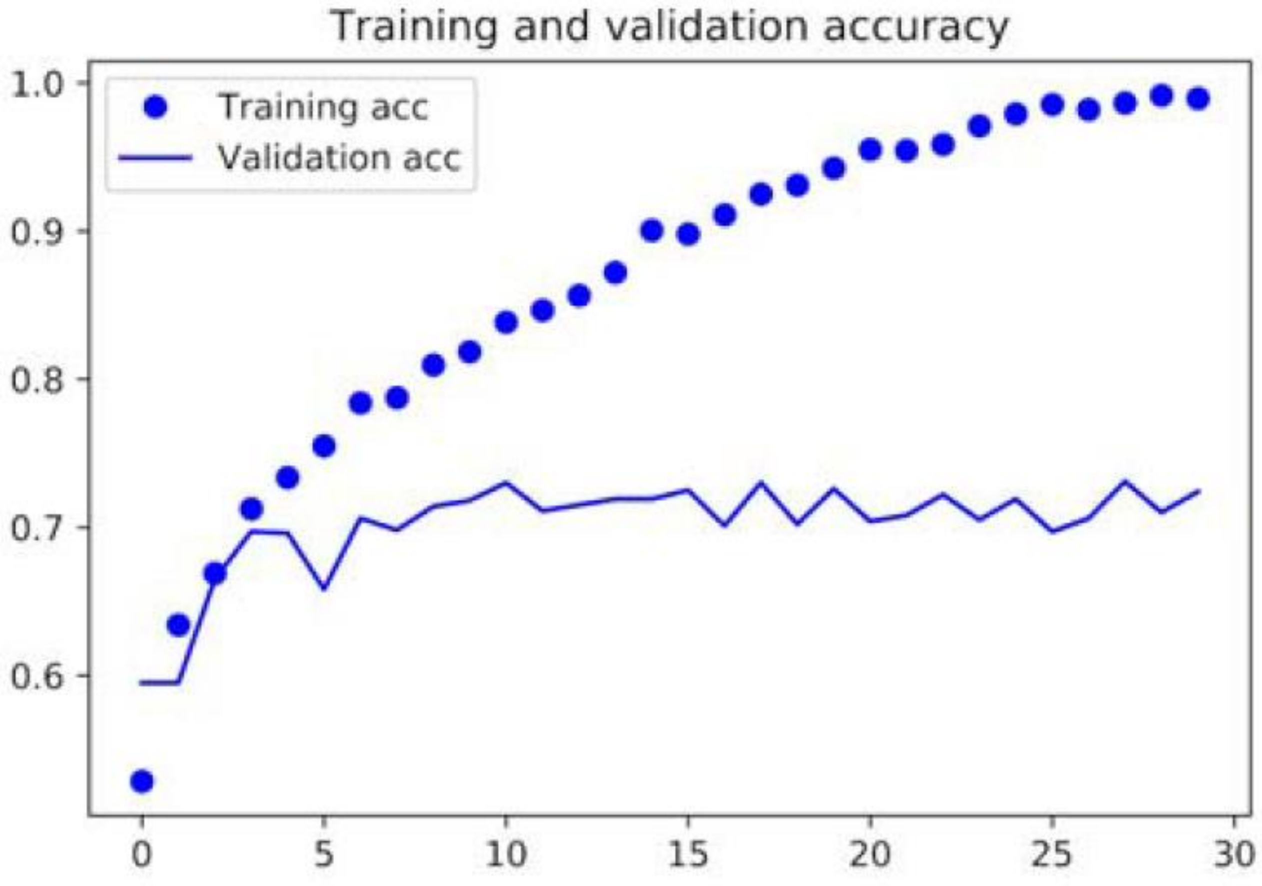


Figure 5.9 Training and validation accuracy

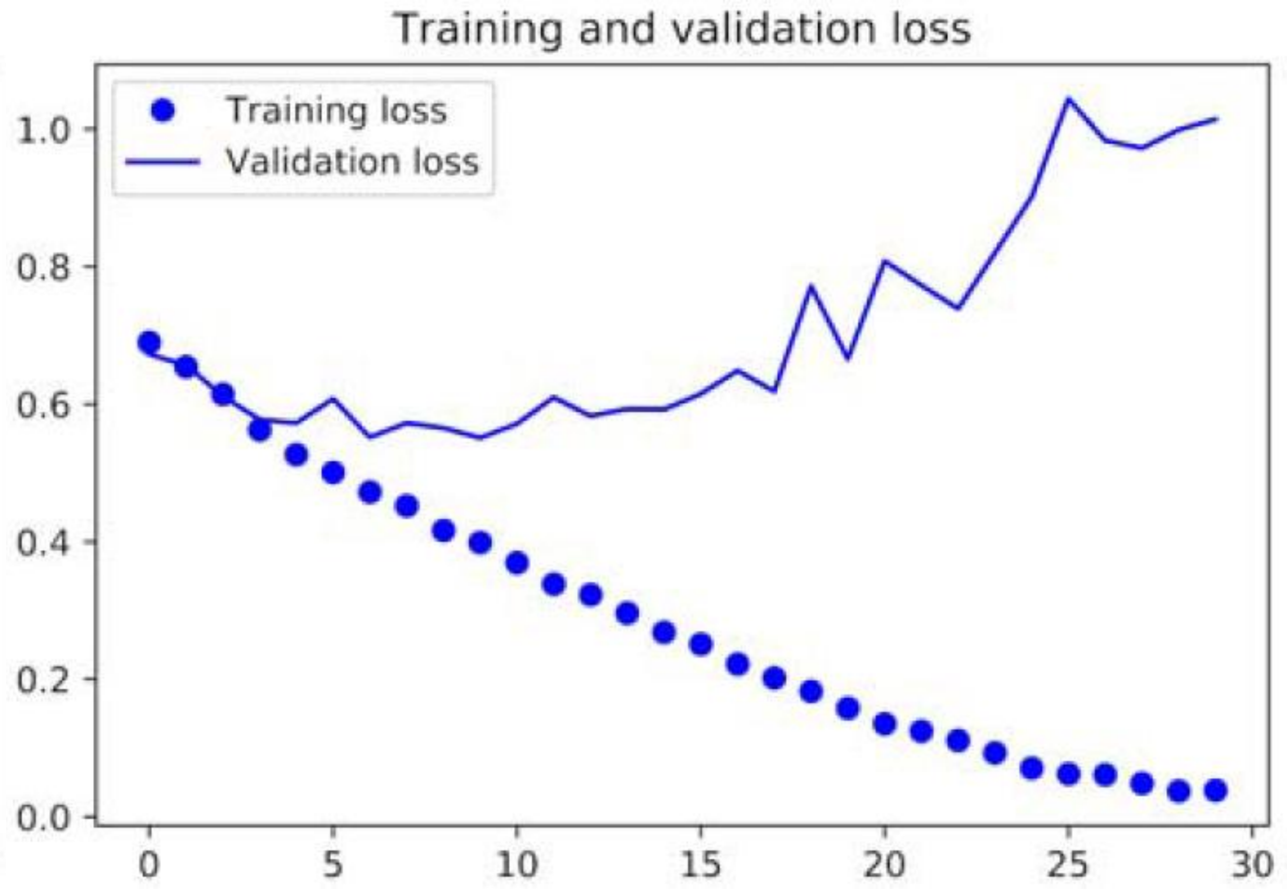


Figure 5.10 Training and validation loss

5.2.4 Data preprocessing

- 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.

5.2.5 Using data augmentation

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')
```

5.2.5 Using data augmentation

- 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.

5.2.5 Using data augmentation

- 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))
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()
```

Module with image-preprocessing utilities

Chooses one image to augment

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!

5.2.5 Using data augmentation

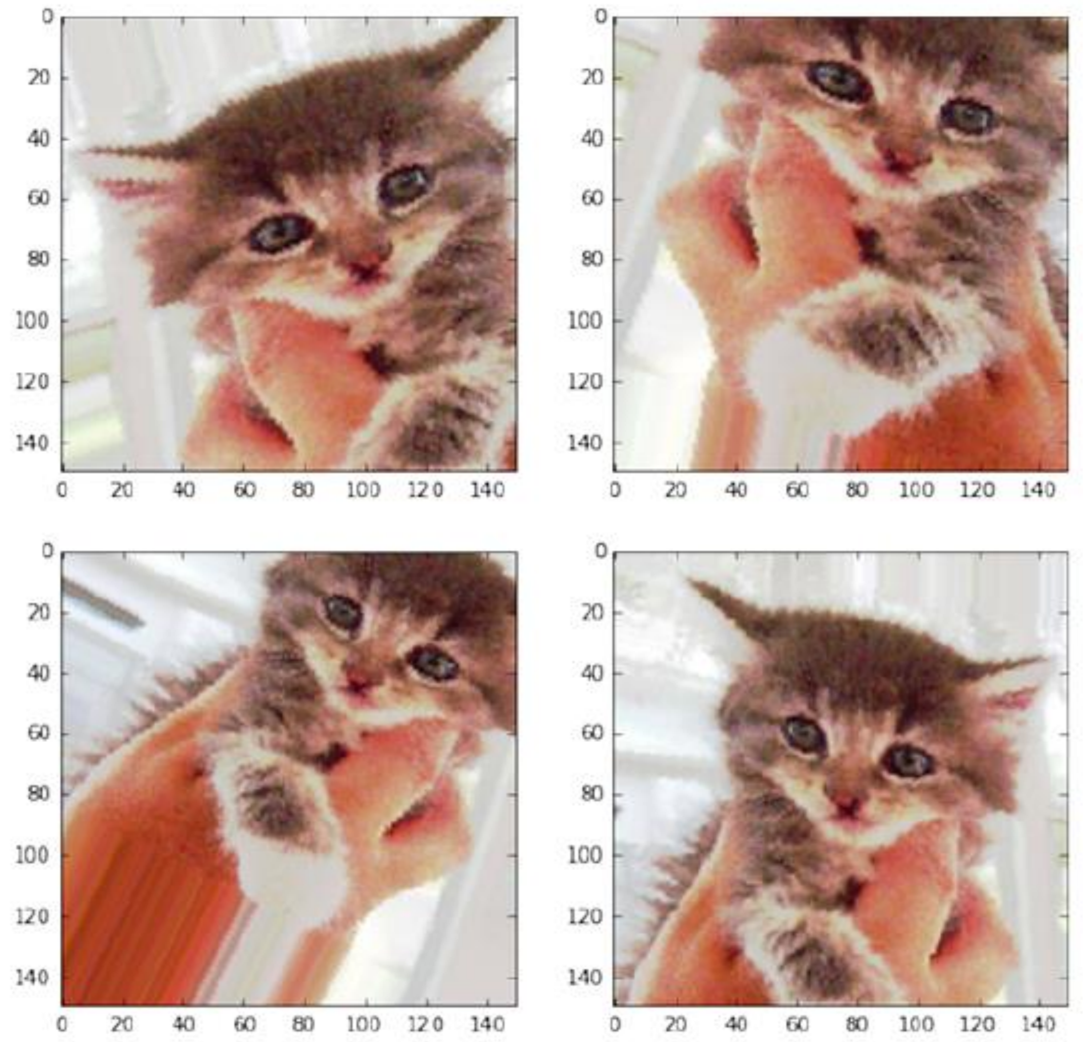


Figure 5.11 Generation of cat pictures via random data augmentation

5.2.5 Using 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.

5.2.5 Using data augmentation

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'])
```

5.2.5 Using data augmentation

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')
```

Note that the validation data shouldn't be augmented!

5.2.5 Using data augmentation

Listing 5.14 Training the convnet using data-augmentation generators

```
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)
```

5.2.5 Using data augmentation

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.

5.2.5 Using data augmentation

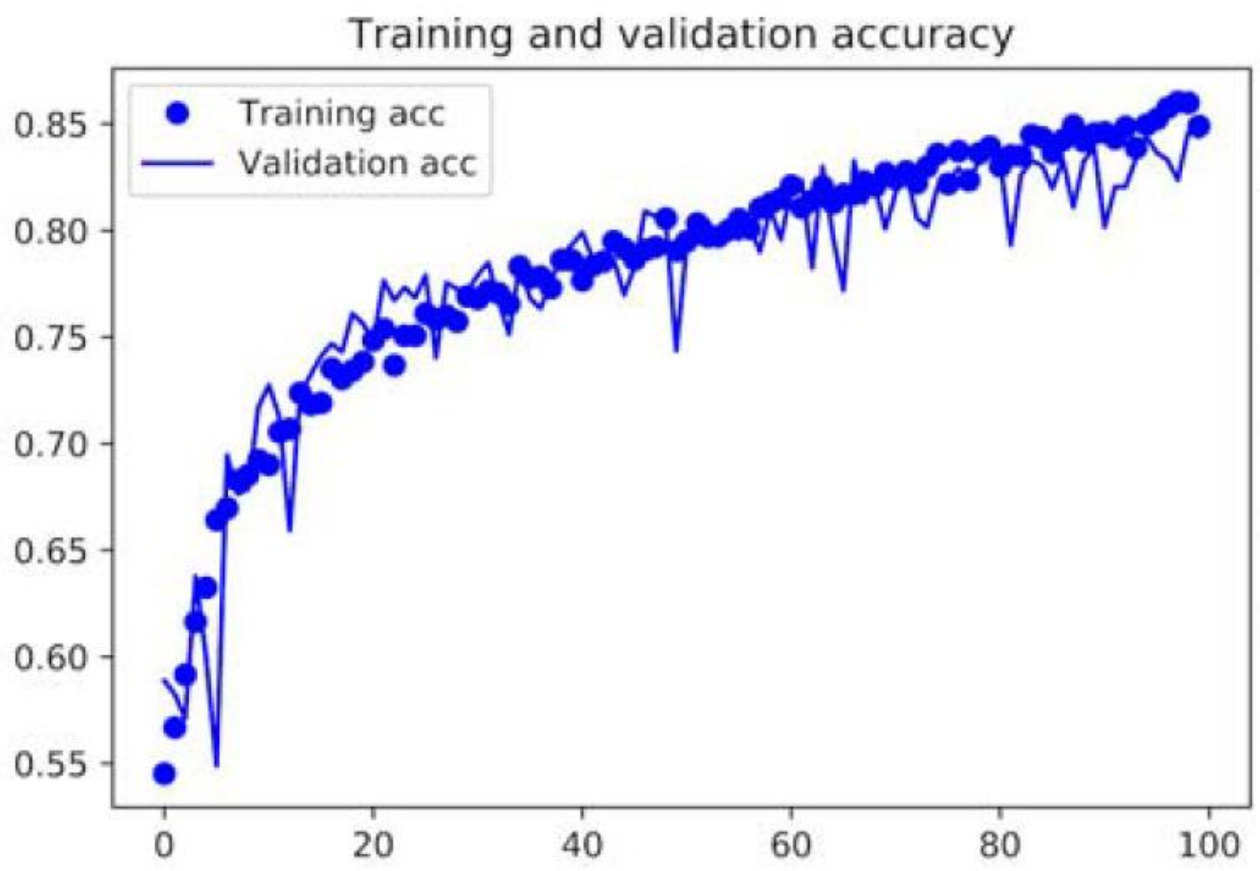


Figure 5.12 Training and validation accuracy with data augmentation

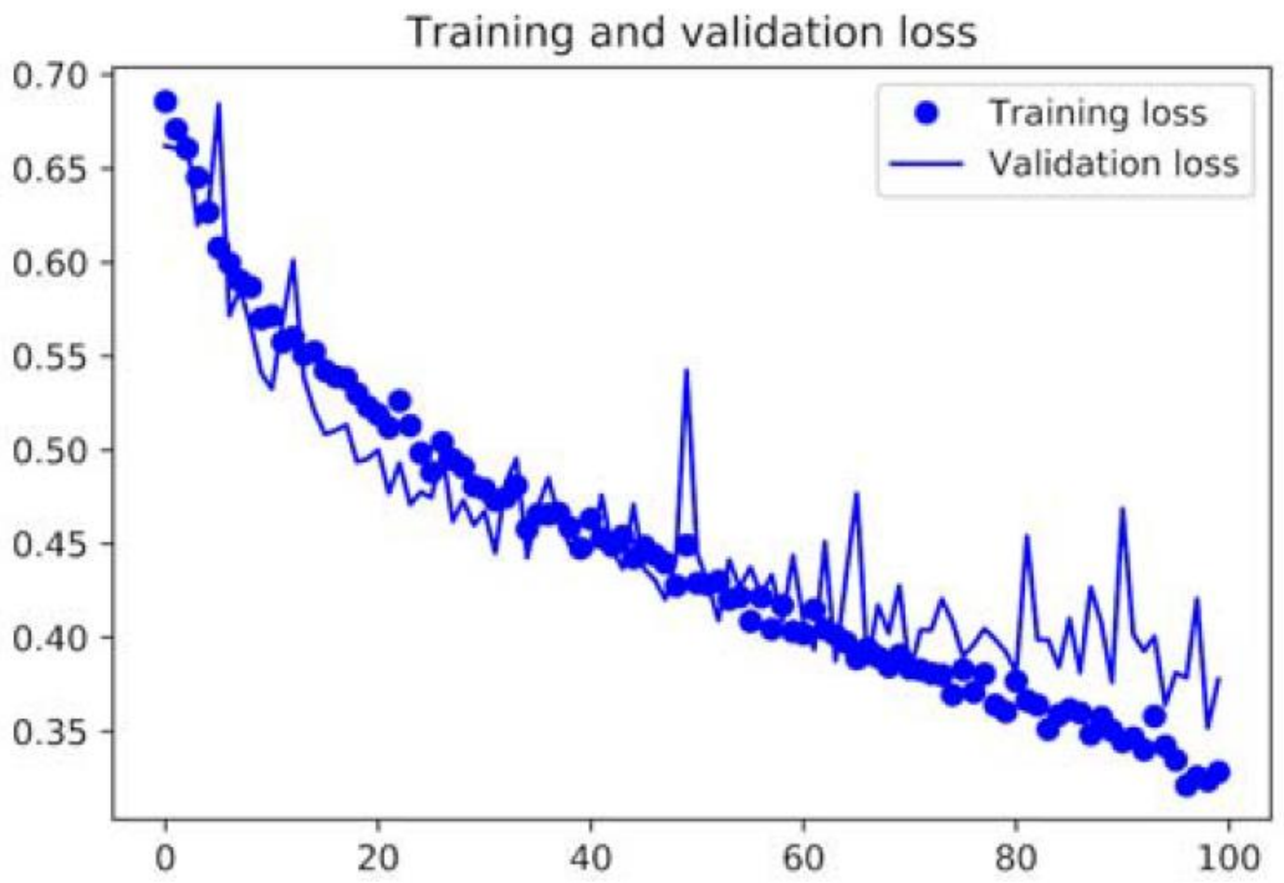


Figure 5.13 Training and validation loss with data augmentation

5.2.5 Using 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.

5.3 Using a pretrained convnet

- A common and highly effective approach to deep learning on small image datasets is to use a pretrained network. A *pretrained network* is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task.
- If this original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer-vision problems, even though these new problems may involve completely different classes than those of the original task.
- For instance, you might train a network on ImageNet (where classes are mostly animals and everyday objects) and then repurpose this trained network for something as remote as identifying furniture items in images.

5.3 Using a pretrained convnet

- Such portability of learned features across different problems is a key advantage of deep learning compared to many older, shallow-learning approaches, and it makes deep learning very effective for small-data problems.
- In this case, let’s consider a large convnet trained on the ImageNet dataset (1.4 million labeled images and 1,000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect to perform well on the dogs-versus-cats classification problem.
- You’ll use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014; it’s a simple and widely used convnet architecture for ImageNet.

5.3 Using a pretrained convnet

- Although it’s an older model, far from the current state of the art and somewhat heavier than many other recent models, I chose it because its architecture is similar to what you’re already familiar with and is easy to understand without introducing any new concepts.
- This may be your first encounter with one of these cutesy model names—VGG, ResNet, Inception, Inception-ResNet, Xception, and so on; you’ll get used to them, because they will come up frequently if you keep doing deep learning for computer vision.
- There are two ways to use a pretrained network: *feature extraction* and *fine-tuning*. We’ll cover both of them. Let’s start with feature extraction.

5.3.1 Feature extraction

- Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- As you saw previously, convnets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely connected classifier.
- The first part is called the *convolutional base* of the model. In the case of convnets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output (see figure 5.14).

5.3.1 Feature extraction

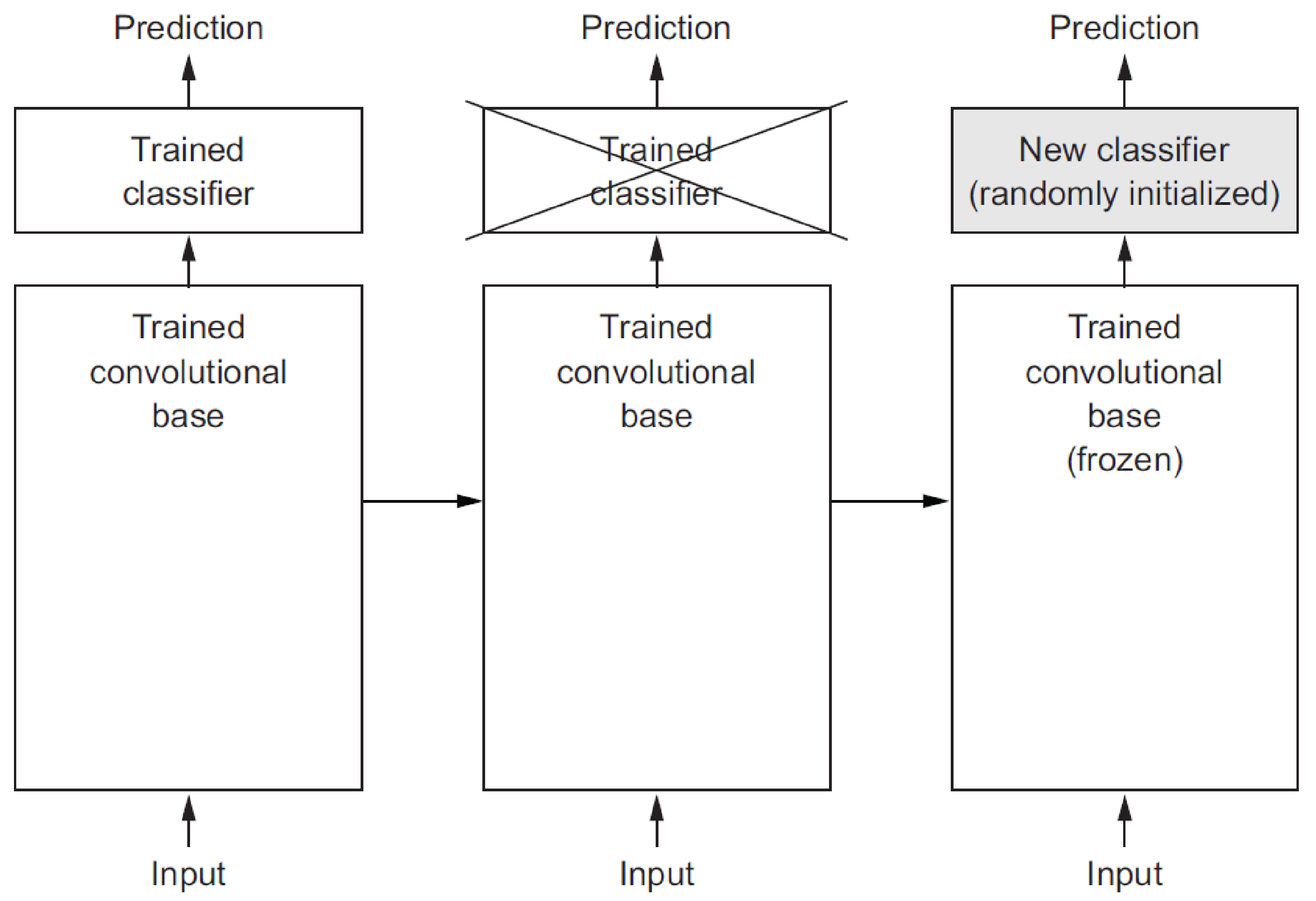

Prediction
Prediction
Prediction
Trained classifier
~~Trained classifier~~
New classifier (randomly initialized)
Trained convolutional base
Trained convolutional base
Trained convolutional base (frozen)
Input
Input
Input

Figure 5.14 Swapping classifiers while keeping the same convolutional base

5.3.1 Feature extraction

- Why only reuse the convolutional base? Could you reuse the densely connected classifier as well? In general, doing so should be avoided.
- The reason is that the representations learned by the convolutional base are likely to be more generic and therefore more reusable: the feature maps of a convnet are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer-vision problem at hand.
- But the representations learned by the classifier will necessarily be specific to the set of classes on which the model was trained—they will only contain information about the presence probability of this or that class in the entire picture.

5.3.1 Feature extraction

- Additionally, representations found in densely connected layers no longer contain any information about *where* objects are located in the input image: these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps.
- For problems where object location matters, densely connected features are largely useless.
- Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model.
- Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), whereas layers that are higher up extract more-abstract concepts (such as “cat ear” or “dog eye”).

5.3.1 Feature extraction

- So if your new dataset differs a lot from the dataset on which the original model was trained, you may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.
- In this case, because the ImageNet class set contains multiple dog and cat classes, it’s likely to be beneficial to reuse the information contained in the densely connected layers of the original model.
- But we’ll choose not to, in order to cover the more general case where the class set of the new problem doesn’t overlap the class set of the original model.
- Let’s put this in practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from cat and dog images, and then train a dogs-versus-cats classifier on top of these features.

5.3.1 Feature extraction

- The VGG16 model, among others, comes prepackaged with Keras. You can import it from the `keras.applications` module.
- Here's the list of image-classification models (all pretrained on the ImageNet dataset) that are available as part of `keras.applications`:
  - Xception
  - Inception V3
  - ResNet50
  - VGG16
  - VGG19
  - MobileNet

5.3.1 Feature extraction

- Let’s instantiate the VGG16 model.

Listing 5.16 Instantiating the VGG16 convolutional base

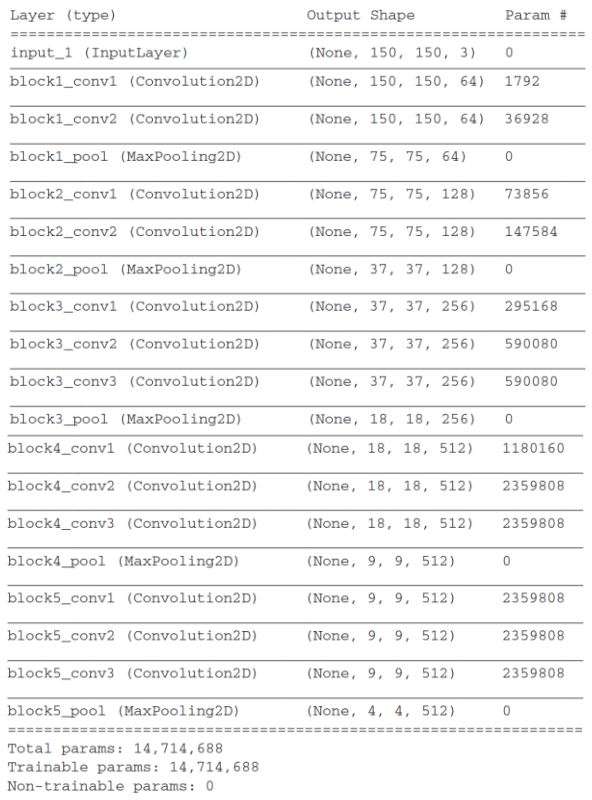
```
from keras.applications import VGG16
conv_base = VGG16(weights='imagenet’,
                  include_top=False,
                  input_shape=(150, 150, 3))
```

- You pass three arguments to the constructor:
  - `weights` specifies the weight checkpoint from which to initialize the model.
  - `include_top` refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because you intend to use your own densely connected classifier (with only two classes: `cat` and `dog`), you don’t need to include it.
  - `input_shape` is the shape of the image tensors that you’ll feed to the network. This argument is purely optional: if you don’t pass it, the network will be able to process inputs of any size.

5.3.1 Feature extraction

- Here’s the detail of the architecture of the VGG16 convolutional base. It’s similar to the simple convnets you’re already familiar with:

```
>>> conv_base.summary()
```



```
Layer (type)                     Output Shape          Param #
================================================================
input_1 (InputLayer)             (None, 150, 150, 3)   0
________________________________________________________________
block1_conv1 (Convolution2D)     (None, 150, 150, 64)  1792
________________________________________________________________
block1_conv2 (Convolution2D)     (None, 150, 150, 64)  36928
________________________________________________________________
block1_pool (MaxPooling2D)       (None, 75, 75, 64)    0
________________________________________________________________
block2_conv1 (Convolution2D)     (None, 75, 75, 128)   73856
________________________________________________________________
block2_conv2 (Convolution2D)     (None, 75, 75, 128)   147584
________________________________________________________________
block2_pool (MaxPooling2D)       (None, 37, 37, 128)   0
________________________________________________________________
block3_conv1 (Convolution2D)     (None, 37, 37, 256)   295168
________________________________________________________________
block3_conv2 (Convolution2D)     (None, 37, 37, 256)   590080
________________________________________________________________
block3_conv3 (Convolution2D)     (None, 37, 37, 256)   590080
________________________________________________________________
block3_pool (MaxPooling2D)       (None, 18, 18, 256)   0
block4_conv1 (Convolution2D)     (None, 18, 18, 512)   1180160
________________________________________________________________
block4_conv2 (Convolution2D)     (None, 18, 18, 512)   2359808
________________________________________________________________
block4_conv3 (Convolution2D)     (None, 18, 18, 512)   2359808
________________________________________________________________
block4_pool (MaxPooling2D)       (None, 9, 9, 512)     0
________________________________________________________________
block5_conv1 (Convolution2D)     (None, 9, 9, 512)     2359808
________________________________________________________________
block5_conv2 (Convolution2D)     (None, 9, 9, 512)     2359808
________________________________________________________________
block5_conv3 (Convolution2D)     (None, 9, 9, 512)     2359808
________________________________________________________________
block5_pool (MaxPooling2D)       (None, 4, 4, 512)     0
================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

5.3.1 Feature extraction

- The final feature map has shape `(4, 4, 512)`. That’s the feature on top of which you’ll stick a densely connected classifier.
- At this point, there are two ways you could proceed:

  1. Running the convolutional base over your dataset, recording its output to a Numpy array on disk, and then using this data as input to a standalone, densely connected classifier similar to those you saw in part 1 of this book. This solution is fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. But for the same reason, this technique won’t allow you to use data augmentation.

5.3.1 Feature extraction

- The final feature map has shape `(4, 4, 512)`. That's the feature on top of which you'll stick a densely connected classifier.
- At this point, there are two ways you could proceed:

  2. Extending the model you have (`conv_base`) by adding `Dense` layers on top, and running the whole thing end to end on the input data. This will allow you to use data augmentation, because every input image goes through the convolutional base every time it's seen by the model. But for the same reason, this technique is far more expensive than the first.

- We'll cover both techniques. Let's walk through the code required to set up the first one: recording the output of `conv_base` on your data and using these outputs as inputs to a new model.

5.3.1 Feature extraction

FAST FEATURE EXTRACTION WITHOUT DATA AUGMENTATION

- You'll start by running instances of the previously introduced `ImageDataGenerator` to extract images as Numpy arrays as well as their labels. You'll extract features from these images by calling the `predict` method of the `conv_base` model.

Listing 5.17 Extracting features using the pretrained convolutional base

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20
```

5.3.1 Feature extraction

```
def extract_features(directory, sample_count):
        features = np.zeros(shape=(sample_count, 4, 4, 512))
        labels = np.zeros(shape=(sample_count))
        generator = datagen.flow_from_directory(
                directory,
                target_size=(150, 150),
                batch_size=batch_size,
                class_mode='binary’)
        i = 0
        for inputs_batch, labels_batch in generator:
                features_batch = conv_base.predict(inputs_batch)
                features[i * batch_size : (i + 1) * batch_size] = features_batch
                labels[i * batch_size : (i + 1) * batch_size] = labels_batch
                i += 1
                if i * batch_size >= sample_count:
                        break
        return features, labels
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

5.3.1 Feature extraction

- The extracted features are currently of shape `(samples, 4, 4, 512)`. You'll feed them to a densely connected classifier, so first you must flatten them to `(samples, 8192)`:

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

- At this point, you can define your densely connected classifier (note the use of dropout for regularization) and train it on the data and labels that you just recorded.

5.3.1 Feature extraction

Listing 5.18 Defining and training the densely connected classifier

```
from keras import models
from keras import layers
from keras import optimizers
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
             loss='binary_crossentropy’,
             metrics=['acc'])
history = model.fit(train_features, train_labels,
             epochs=30,
             batch_size=20,
             validation_data=(validation_features, validation_labels))
```

5.3.1 Feature extraction

- Training is very fast, because you only have to deal with two `Dense` layers—an epoch takes less than one second even on CPU.
- Let’s look at the loss and accuracy curves during training (see figures 5.15 and 5.16).

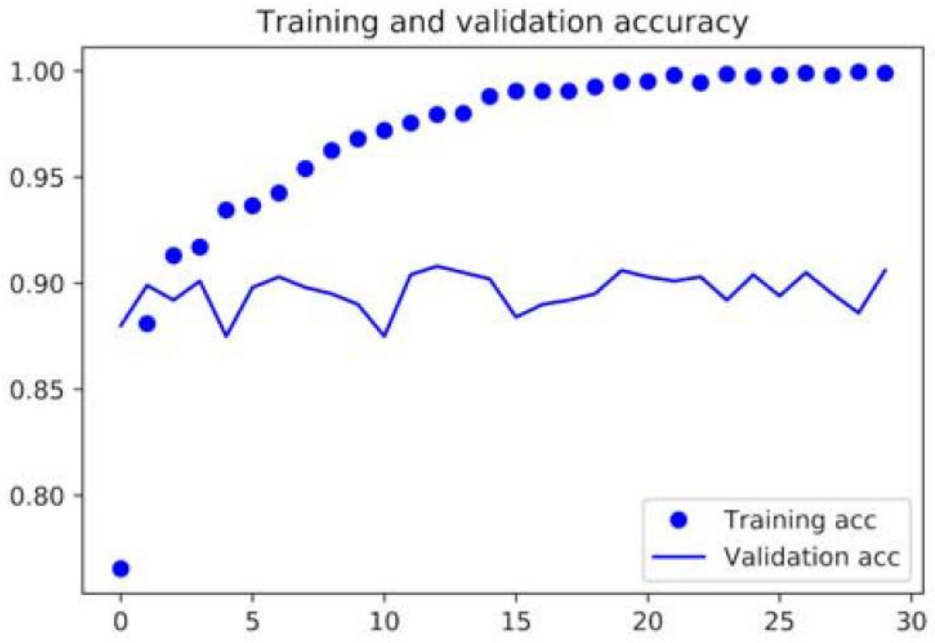


Figure 5.15 Training and validation accuracy for simple feature extraction

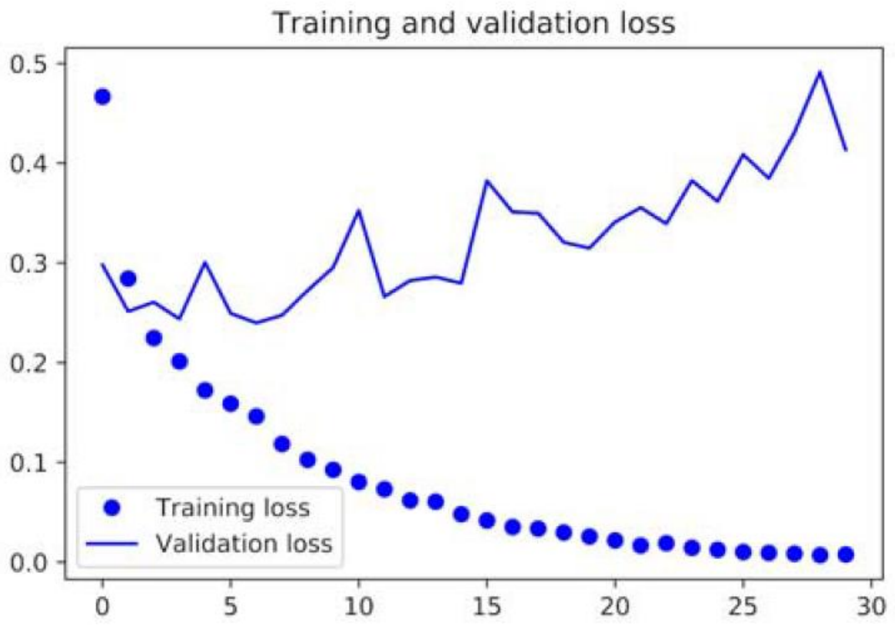


Figure 5.16 Training and validation loss for simple feature extraction

5.3.1 *Feature extraction*

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

```
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()
```

5.3.1 Feature extraction

- You reach a validation accuracy of about 90%—much better than you achieved in the previous section with the small model trained from scratch.
- But the plots also indicate that you’re overfitting almost from the start—despite using dropout with a fairly large rate.
- That’s because this technique doesn’t use data augmentation, which is essential for preventing overfitting with small image datasets.

5.3.1 Feature extraction

FEATURE EXTRACTION WITH DATA AUGMENTATION

- Now, let’s review the second technique I mentioned for doing feature extraction, which is much slower and more expensive, but which allows you to use data augmentation during training: extending the `conv_base` model and running it end to end on the inputs.

NOTE This technique is so expensive that you should only attempt it if you have access to a GPU—it’s absolutely intractable on CPU. If you can’t run your code on GPU, then the previous technique is the way to go.

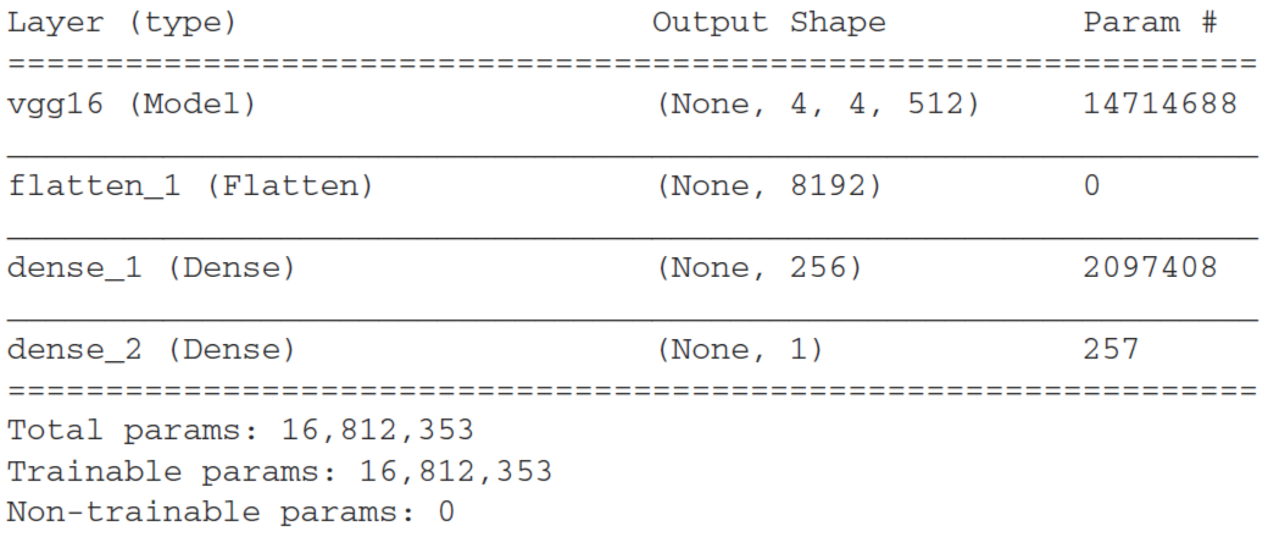
- Because models behave just like layers, you can add a model (like `conv_base`) to a `Sequential` model just like you would add a layer.

5.3.1 Feature extraction

Listing 5.20 Adding a densely connected classifier on top of the convolutional base

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
>>> model.summary()
```



```
Layer (type)                     Output Shape          Param #
================================================================
vgg16 (Model)                    (None, 4, 4, 512)     14714688
________________________________________________________________
flatten_1 (Flatten)              (None, 8192)          0
________________________________________________________________
dense_1 (Dense)                  (None, 256)           2097408
________________________________________________________________
dense_2 (Dense)                  (None, 1)             257
================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

5.3.1 Feature extraction

- As you can see, the convolutional base of VGG16 has 14,714,688 parameters, which is very large. The classifier you’re adding on top has 2 million parameters.
- Before you compile and train the model, it’s very important to freeze the convolutional base. *Freezing* a layer or set of layers means preventing their weights from being updated during training.
- If you don’t do this, then the representations that were previously learned by the convolutional base will be modified during training.
- Because the `Dense` layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned.

5.3.1 Feature extraction

- In Keras, you freeze a network by setting its `trainable` attribute to `False`:

```
>>> print('This is the number of trainable weights '
'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

- With this setup, only the weights from the two `Dense` layers that you added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector).
- Note that in order for these changes to take effect, you must first compile the model. If you ever modify weight trainability after compilation, you should then recompile the model, or these changes will be ignored.

5.3.1 Feature extraction

- Now you can start training your model, with the same data-augmentation configuration that you used in the previous example.

Listing 5.21 Training the model end to end with a frozen convolutional base

```
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
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,
      fill_mode='nearest')
```

5.3.1 Feature extraction

```
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,
      target_size=(150, 150),
      batch_size=20,
      class_mode='binary')
model.compile(loss='binary_crossentropy’,
      optimizer=optimizers.RMSprop(lr=2e-5),
      metrics=['acc'])
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50)
```

5.3.1 Feature extraction

- Let’s plot the results again (see figures 5.17 and 5.18). As you can see, you reach a validation accuracy of about 96%.
- This is much better than you achieved with the small convnet trained from scratch.

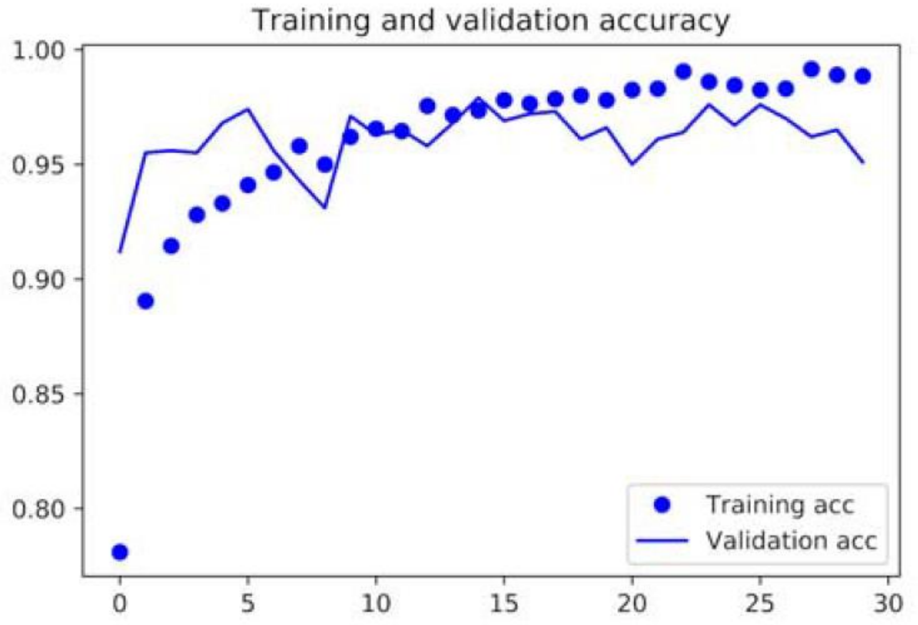


Figure 5.17 Training and validation accuracy for feature extraction with data augmentation

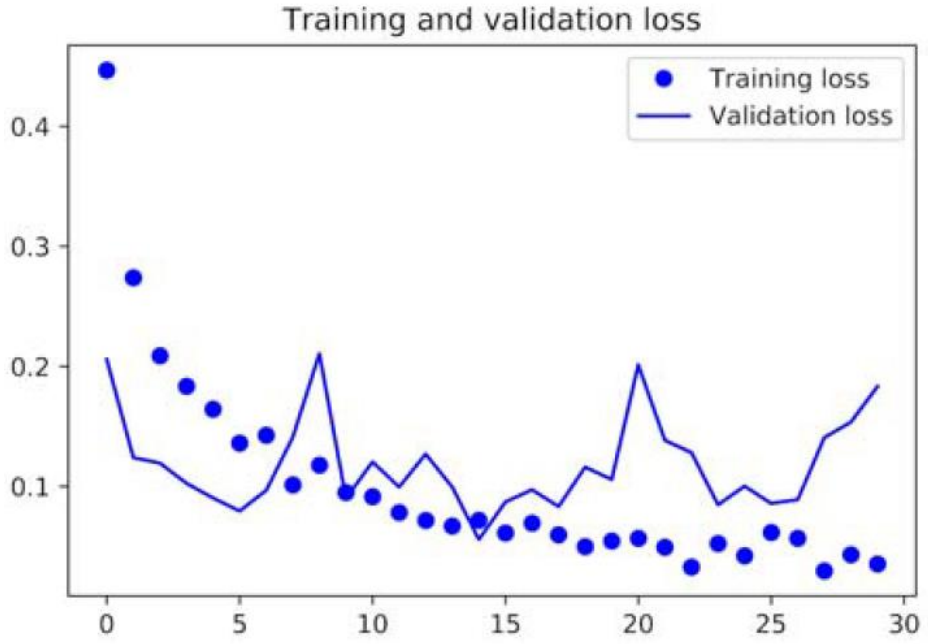


Figure 5.18 Training and validation loss for feature extraction with data augmentation

5.3.2 Fine-tuning

- Another widely used technique for model reuse, complementary to feature extraction, is *fine-tuning* (see figure 5.19).
- Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers.
- This is called *fine-tuning* because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.

5.3.2 Fine-tuning

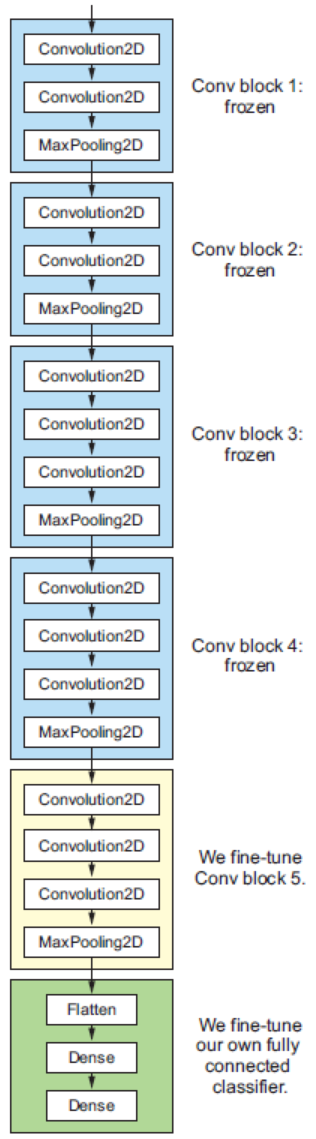


Figure 5.19 Fine-tuning the last convolutional block of the VGG16 network

5.3.2 Fine-tuning

- I stated earlier that it’s necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top.
- For the same reason, it’s only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained.
- If the classifier isn’t already trained, then the error signal propagating through the network during training will be too large, and the representations previously learned by the layers being fine-tuned will be destroyed.
- Thus the steps for fine-tuning a network are as follow:
  1. Add your custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part you added.
  4. Unfreeze some layers in the base network.
  5. Jointly train both these layers and the part you added.

5.3.2 Fine-tuning

- You already completed the first three steps when doing feature extraction. Let’s proceed with step 4: you’ll unfreeze your `conv_base` and then freeze individual layers inside it.
- You’ll fine-tune the last three convolutional layers, which means all layers up to `block4_pool` should be frozen, and the layers `block5_conv1`, `block5_conv2`, and `block5_conv3` should be trainable.

5.3.2 Fine-tuning

- Why not fine-tune more layers? Why not fine-tune the entire convolutional base? You could. But you need to consider the following:
  - Earlier layers in the convolutional base encode more-generic, reusable features, whereas layers higher up encode more-specialized features. It's more useful to fine-tune the more specialized features, because these are the ones that need to be repurposed on your new problem. There would be fast-decreasing returns in fine-tuning lower layers.
  - The more parameters you're training, the more you're at risk of overfitting. The convolutional base has 15 million parameters, so it would be risky to attempt to train it on your small dataset.

5.3.2 Fine-tuning

- Thus, in this situation, it’s a good strategy to fine-tune only the top two or three layers in the convolutional base. Let’s set this up, starting from where you left off in the previous example.

Listing 5.22 Freezing all layers up to a specific one

```
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
	if layer.name == 'block5_conv1’:
		set_trainable = True
	if set_trainable:
		layer.trainable = True
	else:
		layer.trainable = False
```

5.3.2 Fine-tuning

- Now you can begin fine-tuning the network. You’ll do this with the RMSProp optimizer, using a very low learning rate.
- The reason for using a low learning rate is that you want to limit the magnitude of the modifications you make to the representations of the three layers you’re fine-tuning. Updates that are too large may harm these representations.

Listing 5.23 Fine-tuning the model

```
model.compile(loss='binary_crossentropy’,
      optimizer=optimizers.RMSprop(lr=1e-5),
      metrics=['acc'])
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

5.3.2 Fine-tuning

- Let’s plot the results using the same plotting code as before (see figures 5.20 and 5.21).

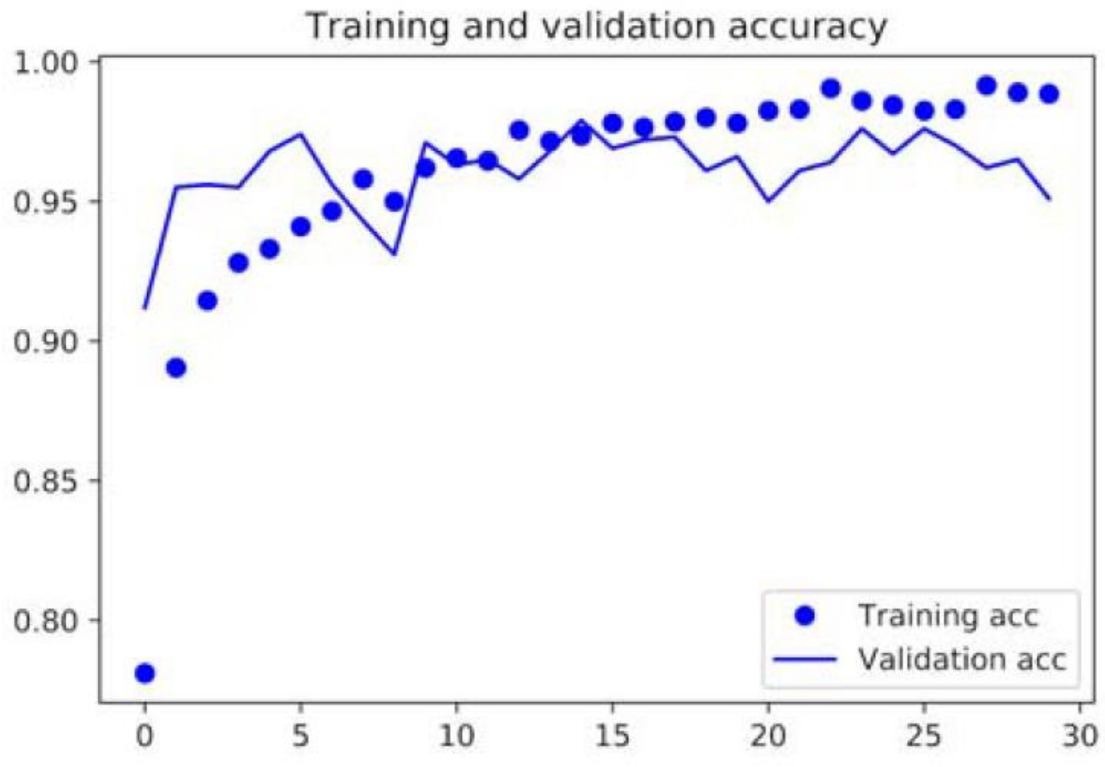


Figure 5.20 Training and validation accuracy for fine-tuning

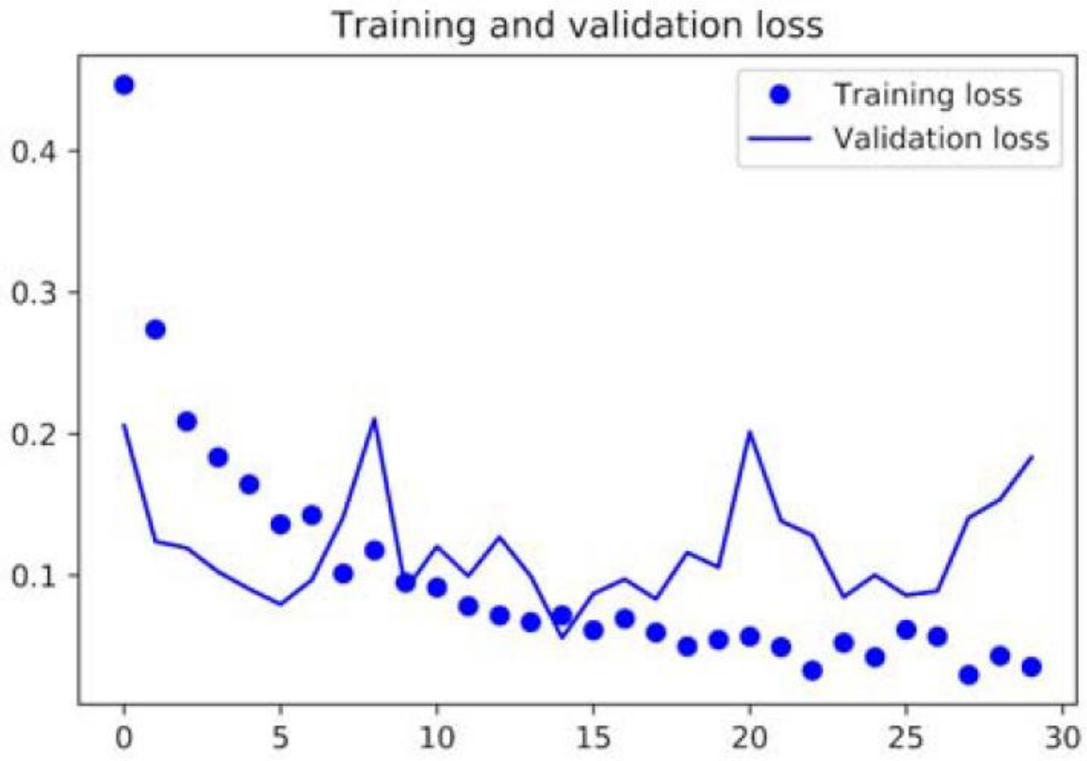


Figure 5.21 Training and validation loss for fine-tuning

5.3.2 Fine-tuning

- These curves look noisy. To make them more readable, you can smooth them by replacing every loss and accuracy with exponential moving averages of these quantities. Here’s a trivial utility function to do this (see figures 5.22 and 5.23).

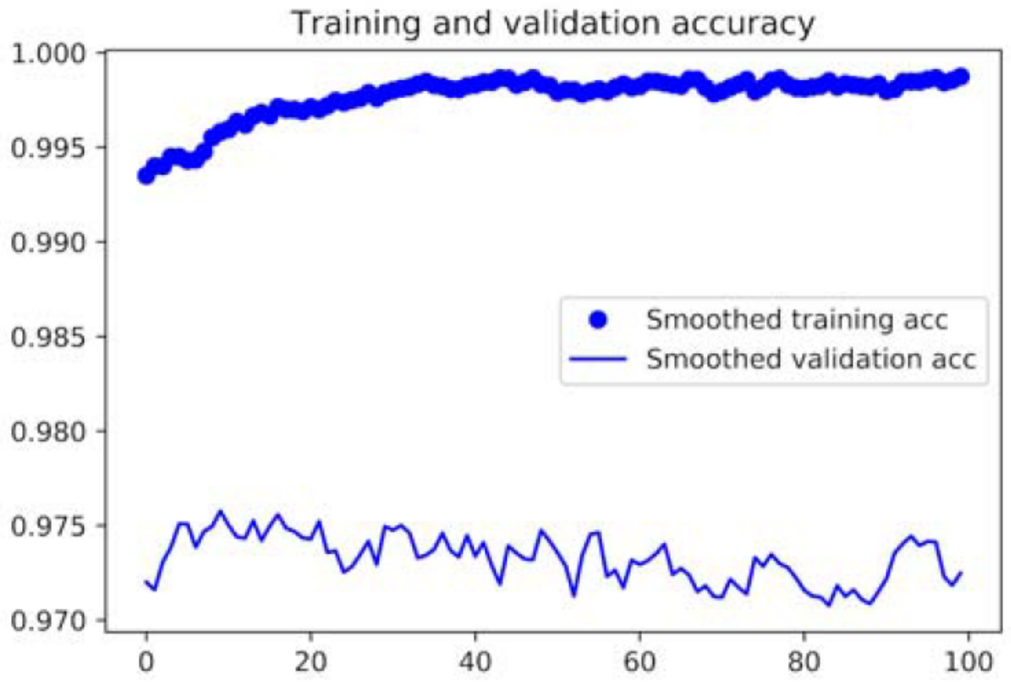


Figure 5.22 Smoothed curves for training and validation accuracy for fine-tuning

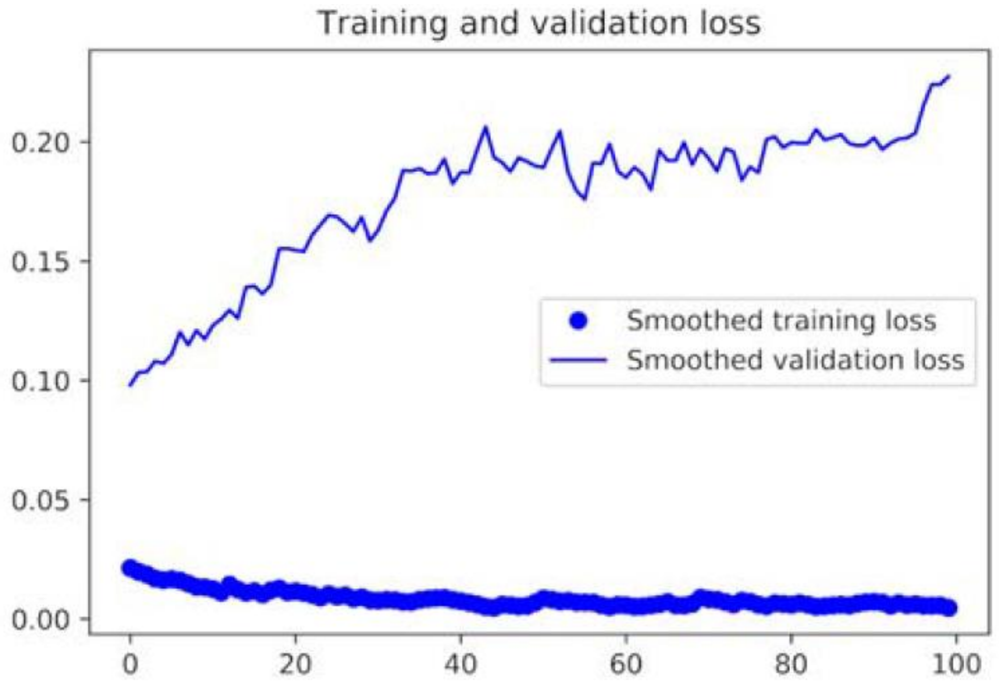


Figure 5.23 Smoothed curves for training and validation loss for fine-tuning

Listing 5.24 Smoothing the plots

```
def smooth_curve(points, factor=0.8):
      smoothed_points = []
      for point in points:
            if smoothed_points:
                  previous = smoothed_points[-1]
                  smoothed_points.append(previous * factor + point * (1 - factor))
            else:
                  smoothed_points.append(point)
      return smoothed_points
plt.plot(epochs,
      smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
      smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs,
      smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
      smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

5.3.2 Fine-tuning

- The validation accuracy curve look much cleaner. You’re seeing a nice 1% absolute improvement in accuracy, from about 96% to above 97%.
- Note that the loss curve doesn’t show any real improvement (in fact, it’s deteriorating).
- You may wonder, how could accuracy stay stable or improve if the loss isn’t decreasing? The answer is simple: what you display is an average of pointwise loss values; but what matters for accuracy is the distribution of the loss values, not their average, because accuracy is the result of a binary thresholding of the class probability predicted by the model.
- The model may still be improving even if this isn’t reflected in the average loss.

5.3.2 Fine-tuning

- You can now finally evaluate this model on the test data:

```
test_generator = test_datagen.flow_from_directory(
test_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')
test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

- Here you get a test accuracy of 97%. In the original Kaggle competition around this dataset, this would have been one of the top results. But using modern deep-learning techniques, you managed to reach this result using only a small fraction of the training data available (about 10%).
- There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

5.3.3 Wrapping up

- Here’s what you should take away from the exercises in the past two sections:
  - Convnets are the best type of machine-learning models for computer-vision tasks. It’s possible to train one from scratch even on a very small dataset, with decent results.
  - On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when you’re working with image data.
  - It’s easy to reuse an existing convnet on a new dataset via feature extraction. This is a valuable technique for working with small image datasets.
  - As a complement to feature extraction, you can use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.
- Now you have a solid set of tools for dealing with image-classification problems—in particular with small datasets.