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).

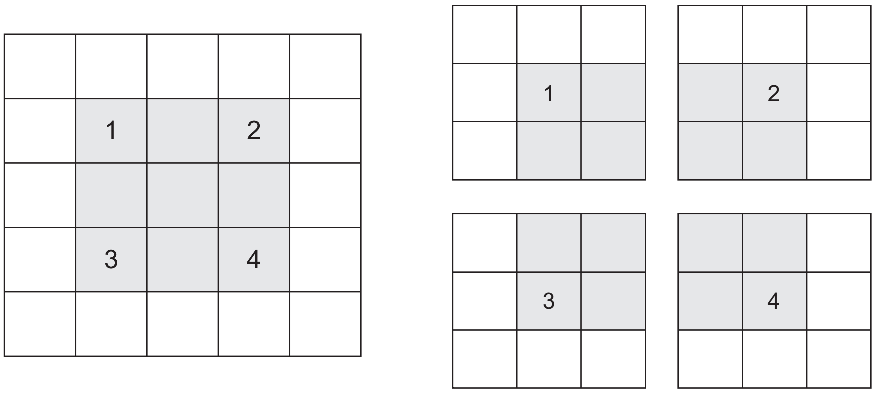

Figure 5.7 3 × 3 convolution patches with 2 × 2 strides

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. Let's look at it in more depth.

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. 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'))
```

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
```

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.

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. 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.

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. 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.

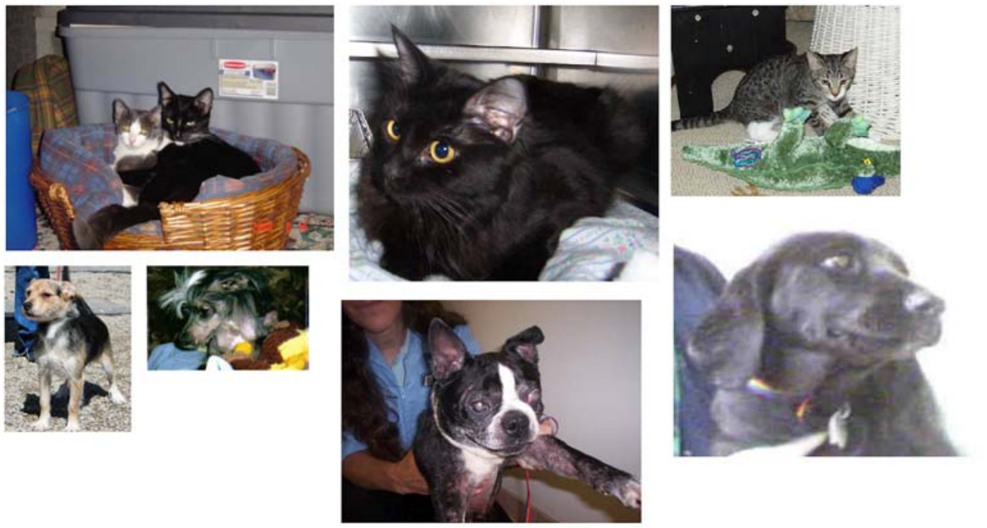

Figure 5.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples are heterogeneous in size, appearance, and so on.

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()
```