9 Advanced deep learning for computer vision

This chapter covers

- The different branches of computer vision: image classification, image segmentation, object detection
- Modern convnet architecture patterns: residual connections, batch normalization, depthwise separable convolutions
- Techniques for visualizing and interpreting what convnets learn

The previous chapter gave you a first introduction to deep learning for computer vision via simple models (stacks of `Conv2D` and `MaxPooling2D` layers) and a simple use case (binary image classification). But there's more to computer vision than image classification! This chapter dives deeper into more diverse applications and advanced best practices.

9.1 Three essential computer vision tasks

So far, we've focused on image classification models: an image goes in, a label comes out. "This image likely contains a cat; this other one likely contains a dog." But image classification is only one of several possible applications of deep learning

in computer vision. In general, there are three essential computer vision tasks you need to know about:

- *Image classification*—Where the goal is to assign one or more labels to an image. It may be either single-label classification (an image can only be in one category, excluding the others), or multi-label classification (tagging all categories that an image belongs to, as seen in figure 9.1). For example, when you search for a keyword on the Google Photos app, behind the scenes you're querying a very large multilabel classification model—one with over 20,000 different classes, trained on millions of images.
- *Image segmentation*—Where the goal is to "segment" or "partition" an image into different areas, with each area usually representing a category (as seen in figure 9.1). For instance, when Zoom or Google Meet diplays a custom background behind you in a video call, it's using an image segmentation model to tell your face apart from what's behind it, at pixel precision.
- *Object detection*—Where the goal is to draw rectangles (called *bounding boxes*) around objects of interest in an image, and associate each rectangle with a class. A self-driving car could use an object-detection model to monitor cars, pedestrians, and signs in view of its cameras, for instance.

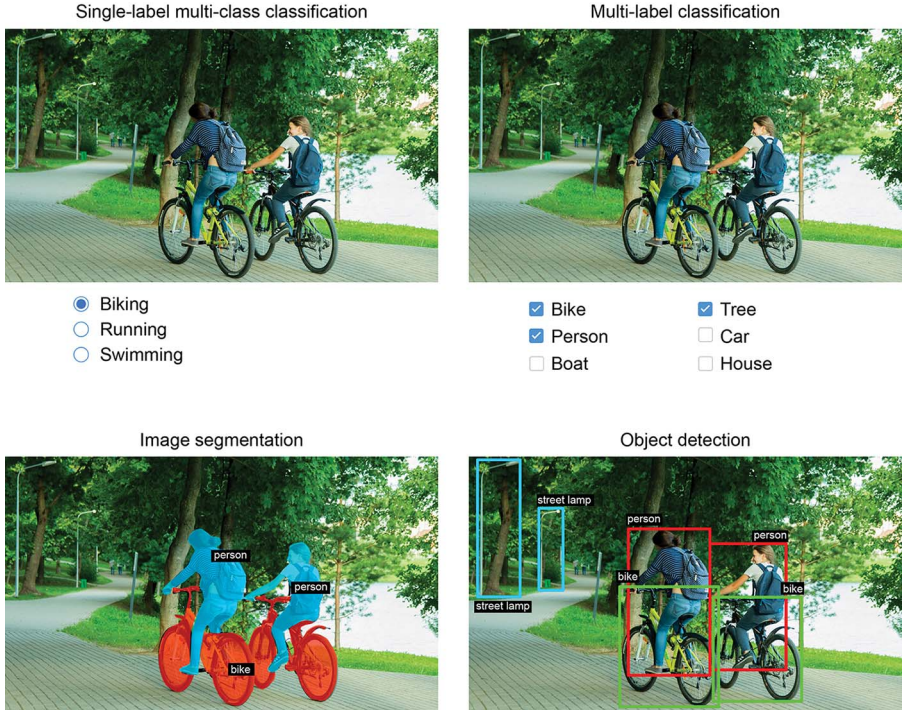


Figure 9.1 The three main computer vision tasks: classification, segmentation, detection

Deep learning for computer vision also encompasses a number of somewhat more niche tasks besides these three, such as image similarity scoring (estimating how visually similar two images are), keypoint detection (pinpointing attributes of interest in an image, such as facial features), pose estimation, 3D mesh estimation, and so on. But to start with, image classification, image segmentation, and object detection form the foundation that every machine learning engineer should be familiar with. Most computer vision applications boil down to one of these three.

You've seen image classification in action in the previous chapter. Next, let's dive into image segmentation. It's a very useful and versatile technique, and you can straightforwardly approach it with what you've already learned so far.

Note that we won't cover object detection, because it would be too specialized and too complicated for an introductory book. However, you can check out the RetinaNet example on keras.io, which shows how to build and train an object detection model from scratch in Keras in around 450 lines of code (https://keras.io/examples/vision/retinanet/).

9.2 An image segmentation example

Image segmentation with deep learning is about using a model to assign a class to each pixel in an image, thus *segmenting* the image into different zones (such as "background" and "foreground," or "road," "car," and "sidewalk"). This general category of techniques can be used to power a considerable variety of valuable applications in image and video editing, autonomous driving, robotics, medical imaging, and so on.

There are two different flavors of image segmentation that you should know about:

- *Semantic segmentation*, where each pixel is independently classified into a semantic category, like "cat." If there are two cats in the image, the corresponding pixels are all mapped to the same generic "cat" category (see figure 9.2).
- *Instance segmentation*, which seeks not only to classify image pixels by category, but also to parse out individual object instances. In an image with two cats in it, instance segmentation would treat "cat 1" and "cat 2" as two separate classes of pixels (see figure 9.2).

In this example, we'll focus on semantic segmentation: we'll be looking once again at images of cats and dogs, and this time we'll learn how to tell apart the main subject and its background.

We'll work with the Oxford-IIIT Pets dataset (www.robots.ox.ac.uk/~vgg/data/pets/), which contains 7,390 pictures of various breeds of cats and dogs, together with foreground-background segmentation masks for each picture. A *segmentation mask* is the image-segmentation equivalent of a label: it's an image the same size as the input image, with a single color channel where each integer value corresponds to the class

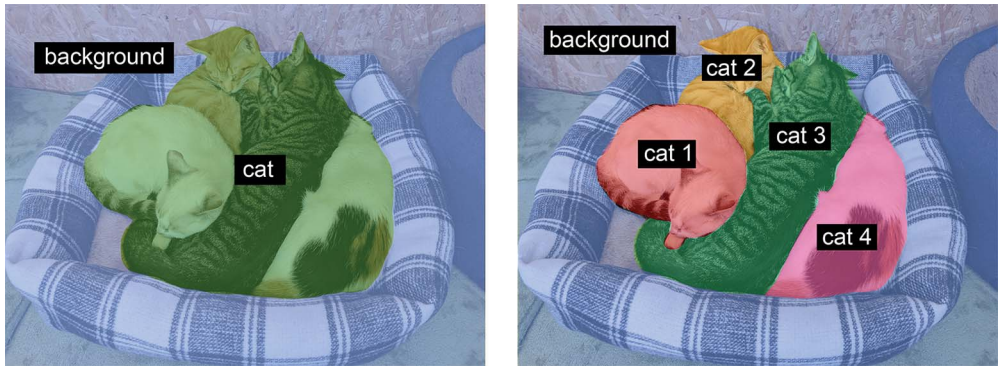


Figure 9.2 Semantic segmentation vs. instance segmentation

of the corresponding pixel in the input image. In our case, the pixels of our segmentation masks can take one of three integer values:

- 1 (foreground)
- 2 (background)
- 3 (contour)

Let's start by downloading and uncompressing our dataset, using the `wget` and `tar` shell utilities:

```
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

The input pictures are stored as JPG files in the images/ folder (such as images/Abyssinian_1.jpg), and the corresponding segmentation mask is stored as a PNG file with the same name in the annotations/trimaps/ folder (such as annotations/trimaps/Abyssinian_1.png).

Let's prepare the list of input file paths, as well as the list of the corresponding mask file paths:

```
import os

input_dir = "images/"
target_dir = "annotations/trimaps/"

input_img_paths = sorted(
    [os.path.join(input_dir, fname)
     for fname in os.listdir(input_dir)
     if fname.endswith(".jpg")])
```

```
target_paths = sorted(
    [os.path.join(target_dir, fname)
     for fname in os.listdir(target_dir)
     if fname.endswith(".png") and not fname.startswith(".")])
```

Now, what does one of these inputs and its mask look like? Let's take a quick look. Here's a sample image (see figure 9.3):

```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

Display input image number 9.

Figure 9.3 An example image

And here is its corresponding target (see figure 9.4):

The original labels are 1, 2, and 3. We subtract 1 so that the labels range from 0 to 2, and then we multiply by 127 so that the labels become 0 (black), 127 (gray), 254 (near-white).

```
def display_target(target_array):
    normalized_array = (target_array.astype("uint8") - 1) * 127
    plt.axis("off")
    plt.imshow(normalized_array[:, :, 0])

img = img_to_array(load_img(target_paths[9], color_mode="grayscale"))
display_target(img)
```

We use color_mode="grayscale" so that the image we load is treated as having a single color channel.

Figure 9.4 The corresponding target mask

Next, let's load our inputs and targets into two NumPy arrays, and let's split the arrays into a training and a validation set. Since the dataset is very small, we can just load everything into memory:

```
import numpy as np
import random

img_size = (200, 200)
num_imgs = len(input_img_paths)

random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_paths)

def path_to_input_image(path):
    return img_to_array(load_img(path, target_size=img_size))

def path_to_target(path):
    img = img_to_array(
        load_img(path, target_size=img_size, color_mode="grayscale"))
    img = img.astype("uint8") - 1
    return img

input_imgs = np.zeros((num_imgs,) + img_size + (3,), dtype="float32")
targets = np.zeros((num_imgs,) + img_size + (1,), dtype="uint8")
for i in range(num_imgs):
    input_imgs[i] = path_to_input_image(input_img_paths[i])
    targets[i] = path_to_target(target_paths[i])

num_val_samples = 1000
train_input_imgs = input_imgs[:-num_val_samples]
train_targets = targets[:-num_val_samples]
val_input_imgs = input_imgs[-num_val_samples:]
val_targets = targets[-num_val_samples:]
```

We resize everything to 200 × 200.

Total number of samples in the data

Shuffle the file paths (they were originally sorted by breed). We use the same seed (1337) in both statements to ensure that the input paths and target paths stay in the same order.

Subtract 1 so that our labels become 0, 1, and 2.

Load all images in the input_imgs float32 array and their masks in the targets uint8 array (same order). The inputs have three channels (RBG values) and the targets have a single channel (which contains integer labels).

Reserve 1,000 samples for validation.

Split the data into a training and a validation set.

Now it's time to define our model:

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

Here's the output of the `model.summary()` call:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 200, 200, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 200, 200, 3)       0
_________________________________________________________________
conv2d (Conv2D)              (None, 100, 100, 64)      1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 100, 100, 64)      36928
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 50, 50, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 50, 50, 128)       147584
_________________________________________________________________
```

```
conv2d_4 (Conv2D)            (None, 25, 25, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 25, 25, 256)       590080
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 25, 25, 256)       590080
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 50, 50, 256)       590080
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 50, 50, 128)       295040
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 100, 100, 128)     147584
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 100, 100, 64)      73792
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 200, 200, 64)      36928
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 200, 200, 3)       1731
=================================================================
Total params: 2,880,643
Trainable params: 2,880,643
Non-trainable params: 0
_________________________________________________________________
```

The first half of the model closely resembles the kind of convnet you'd use for image classification: a stack of `Conv2D` layers, with gradually increasing filter sizes. We downsample our images three times by a factor of two each, ending up with activations of size `(25, 25, 256)`. The purpose of this first half is to encode the images into smaller feature maps, where each spatial location (or pixel) contains information about a large spatial chunk of the original image. You can understand it as a kind of compression.

One important difference between the first half of this model and the classification models you've seen before is the way we do downsampling: in the classification convnets from the last chapter, we used `MaxPooling2D` layers to downsample feature maps. Here, we downsample by adding *strides* to every other convolution layer (if you don't remember the details of how convolution strides work, see "Understanding convolution strides" in section 8.1.1). We do this because, in the case of image segmentation, we care a lot about the *spatial location* of information in the image, since we need to produce per-pixel target masks as output of the model. When you do 2×2 max pooling, you are completely destroying location information within each pooling window: you return one scalar value per window, with zero knowledge of which of the four locations in the windows the value came from. So while max pooling layers perform well for classification tasks, they would hurt us quite a bit for a segmentation task. Meanwhile, strided convolutions do a better job at downsampling feature maps while retaining location information. Throughout this book, you'll notice that we tend to use strides instead of max pooling in any model that cares about feature location, such as the generative models in chapter 12.

The second half of the model is a stack of `Conv2DTranspose` layers. What are those? Well, the output of the first half of the model is a feature map of shape `(25, 25, 256)`,

but we want our final output to have the same shape as the target masks, `(200, 200, 3)`. Therefore, we need to apply a kind of *inverse* of the transformations we've applied so far—something that will *upsample* the feature maps instead of downsampling them. That's the purpose of the `Conv2DTranspose` layer: you can think of it as a kind of convolution layer that *learns to upsample*. If you have an input of shape `(100, 100, 64)`, and you run it through the layer `Conv2D(128, 3, strides=2, padding="same")`, you get an output of shape `(50, 50, 128)`. If you run this output through the layer `Conv2DTranspose(64, 3, strides=2, padding="same")`, you get back an output of shape `(100, 100, 64)`, the same as the original. So after compressing our inputs into feature maps of shape `(25, 25, 256)` via a stack of `Conv2D` layers, we can simply apply the corresponding sequence of `Conv2DTranspose` layers to get back to images of shape `(200, 200, 3)`.

We can now compile and fit our model:

```
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras",
                                    save_best_only=True)
]

history = model.fit(train_input_imgs, train_targets,
                    epochs=50,
                    callbacks=callbacks,
                    batch_size=64,
                    validation_data=(val_input_imgs, val_targets))
```

Let's display our training and validation loss (see figure 9.5):

```
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
```

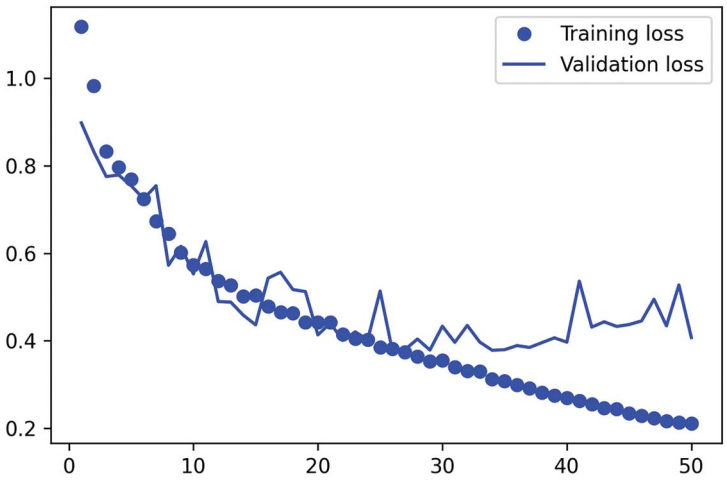


Figure 9.5 Displaying training and validation loss curves

```
val_loss = history.history["val_loss"]
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()
```

You can see that we start overfitting midway, around epoch 25. Let's reload our best performing model according to the validation loss, and demonstrate how to use it to predict a segmentation mask (see figure 9.6):

```
from tensorflow.keras.utils import array_to_img

model = keras.models.load_model("oxford_segmentation.keras")

i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

Utility to display a model's prediction

Figure 9.6 A test image and its predicted segmentation mask

There are a couple of small artifacts in our predicted mask, caused by geometric shapes in the foreground and background. Nevertheless, our model appears to work nicely.

By this point, throughout chapter 8 and the beginning of chapter 9, you've learned the basics of how to perform image classification and image segmentation: you can already accomplish a lot with what you know. However, the convnets that experienced engineers develop to solve real-world problems aren't quite as simple as those we've been using in our demonstrations so far. You're still lacking the essential mental models and thought processes that enable experts to make quick and accurate decisions about how to put together state-of-the-art models. To bridge that gap, you need to learn about *architecture patterns*. Let's dive in.

9.3 Modern convnet architecture patterns

A model's "architecture" is the sum of the choices that went into creating it: which layers to use, how to configure them, and in what arrangement to connect them. These choices define the *hypothesis space* of your model: the space of possible functions that gradient descent can search over, parameterized by the model's weights. Like feature engineering, a good hypothesis space encodes *prior knowledge* that you have about the problem at hand and its solution. For instance, using convolution layers means that you know in advance that the relevant patterns present in your input images are translation-invariant. In order to effectively learn from data, you need to make assumptions about what you're looking for.

Model architecture is often the difference between success and failure. If you make inappropriate architecture choices, your model may be stuck with suboptimal metrics, and no amount of training data will save it. Inversely, a good model architecture will accelerate learning and will enable your model to make efficient use of the training data available, reducing the need for large datasets. A good model architecture is one that *reduces the size of the search space* or otherwise *makes it easier to converge to a good point of the search space*. Just like feature engineering and data curation, model architecture is all about *making the problem simpler* for gradient descent to solve. And remember that gradient descent is a pretty stupid search process, so it needs all the help it can get.

Model architecture is more an art than a science. Experienced machine learning engineers are able to intuitively cobble together high-performing models on their first try, while beginners often struggle to create a model that trains at all. The keyword here is *intuitively*: no one can give you a clear explanation of what works and what doesn't. Experts rely on pattern-matching, an ability that they acquire through extensive practical experience. You'll develop your own intuition throughout this book. However, it's not *all* about intuition either—there isn't much in the way of actual science, but as in any engineering discipline, there are best practices.

In the following sections, we'll review a few essential convnet architecture best practices: in particular, *residual connections*, *batch normalization*, and *separable convolutions*. Once you master how to use them, you will be able to build highly effective image models. We will apply them to our cat vs. dog classification problem.

Let's start from the bird's-eye view: the modularity-hierarchy-reuse (MHR) formula for system architecture.

9.3.1 Modularity, hierarchy, and reuse

If you want to make a complex system simpler, there's a universal recipe you can apply: just structure your amorphous soup of complexity into *modules*, organize the modules into a *hierarchy*, and start *reusing* the same modules in multiple places as appropriate ("reuse" is another word for *abstraction* in this context). That's the MHR formula (modularity-hierarchy-reuse), and it underlies system architecture across pretty much every domain where the term "architecture" is used. It's at the heart of the organization of any system of meaningful complexity, whether it's a cathedral, your own body, the US Navy, or the Keras codebase (see figure 9.7).

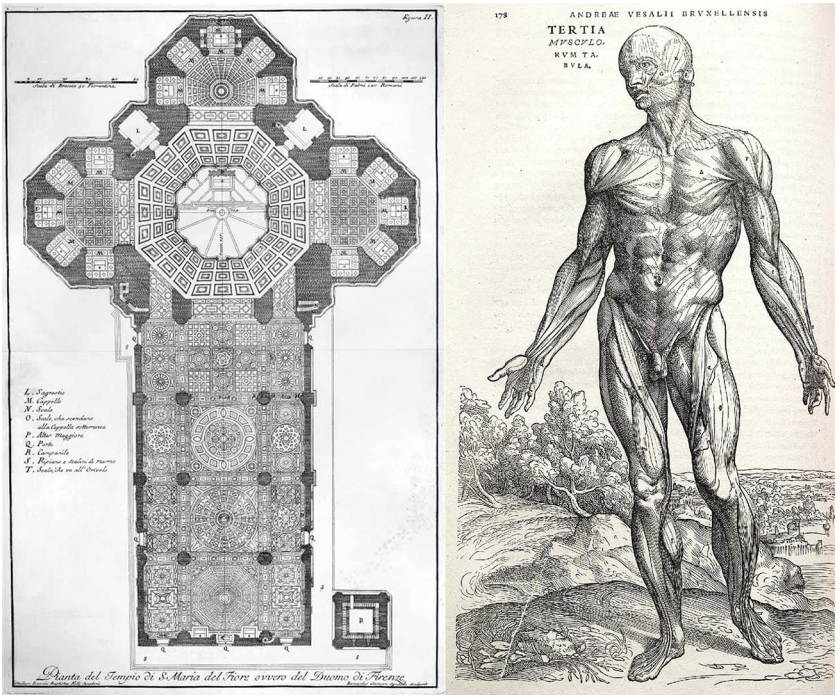


Figure 9.7 Complex systems follow a hierarchical structure and are organized into distinct modules, which are reused multiple times (such as your four limbs, which are all variants of the same blueprint, or your 20 "fingers").

If you're a software engineer, you're already keenly familiar with these principles: an effective codebase is one that is modular, hierarchical, and where you don't reimplement the same thing twice, but instead rely on reusable classes and functions. If you

factor your code by following these principles, you could say you're doing "software architecture."

Deep learning itself is simply the application of this recipe to continuous optimization via gradient descent: you take a classic optimization technique (gradient descent over a continuous function space), and you structure the search space into modules (layers), organized into a deep hierarchy (often just a stack, the simplest kind of hierarchy), where you reuse whatever you can (for instance, convolutions are all about reusing the same information in different spatial locations).

Likewise, deep learning model architecture is primarily about making clever use of modularity, hierarchy, and reuse. You'll notice that all popular convnet architectures are not only structured into layers, they're structured into repeated groups of layers (called "blocks" or "modules"). For instance, the popular VGG16 architecture we used in the previous chapter is structured into repeated "conv, conv, max pooling" blocks (see figure 9.8).

Further, most convnets often feature pyramid-like structures (*feature hierarchies*). Recall, for example, the progression in the number of convolution filters we used in the first convnet we built in the previous chapter: 32, 64, 128. The number of filters grows with layer depth, while the size of the feature maps shrinks accordingly. You'll notice the same pattern in the blocks of the VGG16 model (see figure 9.8).

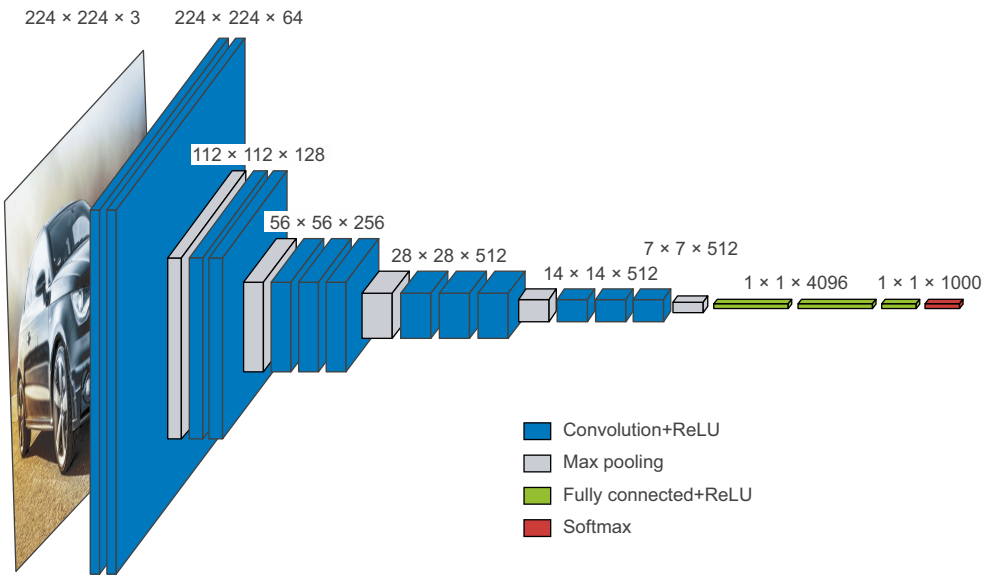


Figure 9.8 The VGG16 architecture: note the repeated layer blocks and the pyramid-like structure of the feature maps

Deeper hierarchies are intrinsically good because they encourage feature reuse, and therefore abstraction. In general, a deep stack of narrow layers performs better than a shallow stack of large layers. However, there's a limit to how deep you can stack layers, due to the problem of *vanishing gradients*. This leads us to our first essential model architecture pattern: residual connections.

On the importance of ablation studies in deep learning research

Deep learning architectures are often more *evolved* than designed—they were developed by repeatedly trying things and selecting what seemed to work. Much like in biological systems, if you take any complicated experimental deep learning setup, chances are you can remove a few modules (or replace some trained features with random ones) with no loss of performance.

This is made worse by the incentives that deep learning researchers face: by making a system more complex than necessary, they can make it appear more interesting or more novel, and thus increase their chances of getting a paper through the peer-review process. If you read lots of deep learning papers, you will notice that they're often optimized for peer review in both style and content in ways that actively hurt clarity of explanation and reliability of results. For instance, mathematics in deep learning papers is rarely used for clearly formalizing concepts or deriving non-obvious results—rather, it gets leveraged as a *signal of seriousness*, like an expensive suit on a salesman.

The goal of research shouldn't be merely to publish, but to generate reliable knowledge. Crucially, understanding *causality* in your system is the most straightforward way to generate reliable knowledge. And there's a very low-effort way to look into causality: *ablation studies*. Ablation studies consist of systematically trying to remove parts of a system—making it simpler—to identify where its performance actually comes from. If you find that X + Y + Z gives you good results, also try X, Y, Z, X + Y, X + Z, and Y + Z, and see what happens.

If you become a deep learning researcher, cut through the noise in the research process: do ablation studies for your models. Always ask, "Could there be a simpler explanation? Is this added complexity really necessary? Why?"

9.3.2 Residual connections

You probably know about the game of Telephone, also called *Chinese whispers* in the UK and *téléphone arabe* in France, where an initial message is whispered in the ear of a player, who then whispers it in the ear of the next player, and so on. The final message ends up bearing little resemblance to its original version. It's a fun metaphor for the cumulative errors that occur in sequential transmission over a noisy channel.

As it happens, backpropagation in a sequential deep learning model is pretty similar to the game of Telephone. You've got a chain of functions, like this one:

```
y = f4(f3(f2(f1(x))))
```

The name of the game is to adjust the parameters of each function in the chain based on the error recorded on the output of `f4` (the loss of the model). To adjust `f1`, you'll need to percolate error information through `f2`, `f3`, and `f4`. However, each successive function in the chain introduces some amount of noise. If your function chain is too deep, this noise starts overwhelming gradient information, and backpropagation stops working. Your model won't train at all. This is the *vanishing gradients* problem.

The fix is simple: just force each function in the chain to be nondestructive—to retain a noiseless version of the information contained in the previous input. The easiest way to implement this is to use a *residual connection*. It's dead easy: just add the input of a layer or block of layers back to its output (see figure 9.9). The residual connection acts as an *information shortcut* around destructive or noisy blocks (such as blocks that contain `relu` activations or dropout layers), enabling error gradient information from early layers to propagate noiselessly through a deep network. This technique was introduced in 2015 with the ResNet family of models (developed by He et al. at Microsoft).[1]

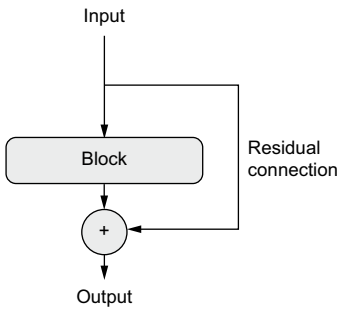


Figure 9.9 A residual connection around a processing block

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

Save a pointer to the original input. This is called the residual.

This computation block can potentially be destructive or noisy, and that's fine.

Add the original input to the layer's output: the final output will thus always preserve full information about the original input.

[1] Kaiming He et al., "Deep Residual Learning for Image Recognition," Conference on Computer Vision and Pattern Recognition (2015), https://arxiv.org/abs/1512.03385.

Note that adding the input back to the output of a block implies that the output should have the same shape as the input. However, this is not the case if your block includes convolutional layers with an increased number of filters, or a max pooling layer. In such cases, use a 1 × 1 `Conv2D` layer with no activation to linearly project the residual to the desired output shape (see listing 9.2). You'd typically use `padding="same"` in the convolution layers in your target block so as to avoid spatial downsampling due to padding, and you'd use strides in the residual projection to match any downsampling caused by a max pooling layer (see listing 9.3).

Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

Listing 9.3 Case where the target block includes a max pooling layer

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

To make these ideas more concrete, here's an example of a simple convnet structured into a series of blocks, each made of two convolution layers and one optional max pooling layer, with a residual connection around each block:

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
```

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

```
        if pooling:
            x = layers.MaxPooling2D(2, padding="same")(x)
            residual = layers.Conv2D(filters, 1, strides=2)(residual)
        elif filters != residual.shape[-1]:
            residual = layers.Conv2D(filters, 1)(residual)
        x = layers.add([x, residual])
        return x

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)
x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
model.summary()
```

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

This is the model summary we get:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0
_________________________________________________________________
rescaling (Rescaling)        (None, 32, 32, 3)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        128
_________________________________________________________________
add (Add)                    (None, 16, 16, 32)        0
                                                       max_pooling2d[0][0]
                                                       conv2d_2[0][0]
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D) (None, 8, 8, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 64)          2112
_________________________________________________________________
add_1 (Add)                  (None, 8, 8, 64)          0
                                                       max_pooling2d_1[0][0]
                                                       conv2d_5[0][0]
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
```

```
conv2d_8 (Conv2D)                (None, 8, 8, 128)     8320        add_1[0][0]
__________________________________________________________________________________________________
add_2 (Add)                      (None, 8, 8, 128)     0           conv2d_7[0][0]
                                                                   conv2d_8[0][0]
__________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 128)           0           add_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                    (None, 1)             129         global_average_pooling2d[0][0]
==================================================================================================
Total params: 297,697
Trainable params: 297,697
Non-trainable params: 0
__________________________________________________________________________________________________
```

With residual connections, you can build networks of arbitrary depth, without having to worry about vanishing gradients.

Now let's move on to the next essential convnet architecture pattern: *batch normalization*.

9.3.3 Batch normalization

Normalization is a broad category of methods that seek to make different samples seen by a machine learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is one you've already seen several times in this book: centering the data on zero by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit variance:

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Previous examples in this book normalized data before feeding it into models. But data normalization may be of interest after every transformation operated by the network: even if the data entering a `Dense` or `Conv2D` network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming out. Could normalizing intermediate activations help?

Batch normalization does just that. It's a type of layer (`BatchNormalization` in Keras) introduced in 2015 by Ioffe and Szegedy;[2] it can adaptively normalize data even as the mean and variance change over time during training. During training, it uses the mean and variance of the current batch of data to normalize samples, and during inference (when a big enough batch of representative data may not be available), it uses an exponential moving average of the batch-wise mean and variance of the data seen during training.

[2] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proceedings of the 32nd International Conference on Machine Learning* (2015), https://arxiv.org/abs/1502.03167.

Although the original paper stated that batch normalization operates by "reducing internal covariate shift," no one really knows for sure why batch normalization helps. There are various hypotheses, but no certitudes. You'll find that this is true of many things in deep learning—deep learning is not an exact science, but a set of ever-changing, empirically derived engineering best practices, woven together by unreliable narratives. You will sometimes feel like the book you have in hand tells you *how* to do something but doesn't quite satisfactorily say *why* it works: that's because we know the how but we don't know the why. Whenever a reliable explanation is available, I make sure to mention it. Batch normalization isn't one of those cases.

In practice, the main effect of batch normalization appears to be that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks. Some very deep networks can only be trained if they include multiple `BatchNormalization` layers. For instance, batch normalization is used liberally in many of the advanced convnet architectures that come packaged with Keras, such as ResNet50, EfficientNet, and Xception.

The `BatchNormalization` layer can be used after any layer—`Dense`, `Conv2D`, etc.:

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
```

Because the output of the Conv2D layer gets normalized, the layer doesn't need its own bias vector.

> **NOTE** Both `Dense` and `Conv2D` involve a *bias vector*, a learned variable whose purpose is to make the layer *affine* rather than purely linear. For instance, `Conv2D` returns, schematically, `y = conv(x, kernel) + bias`, and `Dense` returns `y = dot(x, kernel) + bias`. Because the normalization step will take care of centering the layer's output on zero, the bias vector is no longer needed when using `BatchNormalization`, and the layer can be created without it via the option `use_bias=False`. This makes the layer slightly leaner.

Importantly, I would generally recommend placing the previous layer's activation *after* the batch normalization layer (although this is still a subject of debate). So instead of doing what is shown in listing 9.4, you would do what's shown in listing 9.5.

Listing 9.4 How not to use batch normalization

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

Listing 9.5 How to use batch normalization: the activation comes last

```
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
```

Note the lack of activation here.

We place the activation after the BatchNormalization layer.

The intuitive reason for this approach is that batch normalization will center your inputs on zero, while your `relu` activation uses zero as a pivot for keeping or dropping activated channels: doing normalization before the activation maximizes the utilization of the `relu`. That said, this ordering best practice is not exactly critical, so if you do convolution, then activation, and then batch normalization, your model will still train, and you won't necessarily see worse results.

> **On batch normalization and fine-tuning**
>
> Batch normalization has many quirks. One of the main ones relates to fine-tuning: when fine-tuning a model that includes `BatchNormalization` layers, I recommend leaving these layers frozen (set their `trainable` attribute to `False`). Otherwise they will keep updating their internal mean and variance, which can interfere with the very small updates applied to the surrounding `Conv2D` layers.

Now let's take a look at the last architecture pattern in our series: depthwise separable convolutions.

9.3.4 Depthwise separable convolutions

What if I told you that there's a layer you can use as a drop-in replacement for `Conv2D` that will make your model smaller (fewer trainable weight parameters) and leaner (fewer floating-point operations) and cause it to perform a few percentage points better on its task? That is precisely what the *depthwise separable convolution* layer does (`SeparableConv2D` in Keras). This layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution (a 1 × 1 convolution), as shown in figure 9.10.

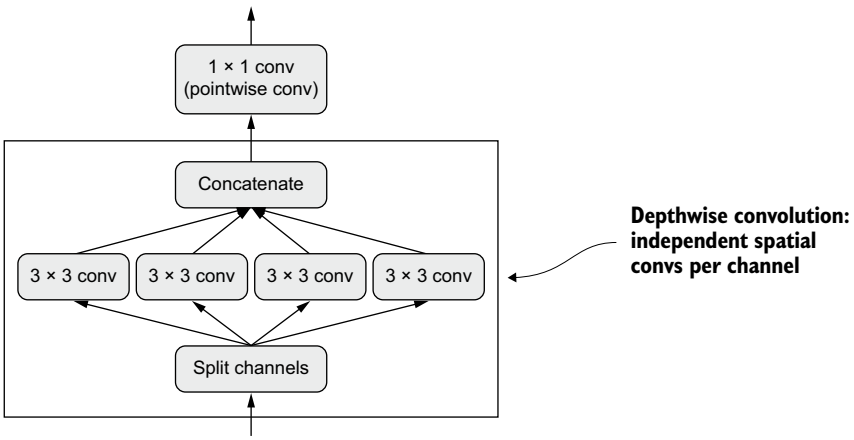


Figure 9.10 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

This is equivalent to separating the learning of spatial features and the learning of channel-wise features. In much the same way that convolution relies on the assumption that the patterns in images are not tied to specific locations, depthwise separable convolution relies on the assumption that *spatial locations* in intermediate activations are *highly correlated*, but *different channels* are *highly independent*. Because this assumption is generally true for the image representations learned by deep neural networks, it serves as a useful prior that helps the model make more efficient use of its training data. A model with stronger priors about the structure of the information it will have to process is a better model—as long as the priors are accurate.

Depthwise separable convolution requires significantly fewer parameters and involves fewer computations compared to regular convolution, while having comparable representational power. It results in smaller models that converge faster and are less prone to overfitting. These advantages become especially important when you're training small models from scratch on limited data.

When it comes to larger-scale models, depthwise separable convolutions are the basis of the Xception architecture, a high-performing convnet that comes packaged with Keras. You can read more about the theoretical grounding for depthwise separable convolutions and Xception in the paper "Xception: Deep Learning with Depthwise Separable Convolutions."[3]

The co-evolution of hardware, software, and algorithms

Consider a regular convolution operation with a 3 × 3 window, 64 input channels, and 64 output channels. It uses 3*3*64*64 = 36,864 trainable parameters, and when you apply it to an image, it runs a number of floating-point operations that is proportional to this parameter count. Meanwhile, consider an equivalent depthwise separable convolution: it only involves 3*3*64 + 64*64 = 4,672 trainable parameters, and proportionally fewer floating-point operations. This efficiency improvement only increases as the number of filters or the size of the convolution windows gets larger.

As a result, you would expect depthwise separable convolutions to be dramatically faster, right? Hold on. This would be true if you were writing simple CUDA or C implementations of these algorithms—in fact, you do see a meaningful speedup when running on CPU, where the underlying implementation is parallelized C. But in practice, you're probably using a GPU, and what you're executing on it is far from a "simple" CUDA implementation: it's a *cuDNN kernel*, a piece of code that has been extraordinarily optimized, down to each machine instruction. It certainly makes sense to spend a lot of effort optimizing this code, since cuDNN convolutions on NVIDIA hardware are responsible for many exaFLOPS of computation every day. But a side effect of this extreme micro-optimization is that alternative approaches have little chance to compete on performance—even approaches that have significant intrinsic advantages, like depthwise separable convolutions.

[3] François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," Conference on Computer Vision and Pattern Recognition (2017), https://arxiv.org/abs/1610.02357.

Despite repeated requests to NVIDIA, depthwise separable convolutions have not benefited from nearly the same level of software and hardware optimization as regular convolutions, and as a result they remain only about as fast as regular convolutions, even though they're using quadratically fewer parameters and floating-point operations. Note, though, that using depthwise separable convolutions remains a good idea even if it does not result in a speedup: their lower parameter count means that you are less at risk of overfitting, and their assumption that channels should be uncorrelated leads to faster model convergence and more robust representations.

What is a slight inconvenience in this case can become an impassable wall in other situations: because the entire hardware and software ecosystem of deep learning has been micro-optimized for a very specific set of algorithms (in particular, convnets trained via backpropagation), there's an extremely high cost to steering away from the beaten path. If you were to experiment with alternative algorithms, such as gradient-free optimization or spiking neural networks, the first few parallel C++ or CUDA implementations you'd come up with would be orders of magnitude slower than a good old convnet, no matter how clever and efficient your ideas were. Convincing other researchers to adopt your method would be a tough sell, even if it were just plain better.

You could say that modern deep learning is the product of a co-evolution process between hardware, software, and algorithms: the availability of NVIDIA GPUs and CUDA led to the early success of backpropagation-trained convnets, which led NVIDIA to optimize its hardware and software for these algorithms, which in turn led to consolidation of the research community behind these methods. At this point, figuring out a different path would require a multi-year re-engineering of the entire ecosystem.

9.3.5 Putting it together: A mini Xception-like model

As a reminder, here are the convnet architecture principles you've learned so far:

- Your model should be organized into repeated *blocks* of layers, usually made of multiple convolution layers and a max pooling layer.
- The number of filters in your layers should increase as the size of the spatial feature maps decreases.
- Deep and narrow is better than broad and shallow.
- Introducing residual connections around blocks of layers helps you train deeper networks.
- It can be beneficial to introduce batch normalization layers after your convolution layers.
- It can be beneficial to replace `Conv2D` layers with `SeparableConv2D` layers, which are more parameter-efficient.

Let's bring these ideas together into a single model. Its architecture will resemble a smaller version of Xception, and we'll apply it to the dogs vs. cats task from the last chapter. For data loading and model training, we'll simply reuse the setup we used in section 8.2.5, but we'll replace the model definition with the following convnet:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)

x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x)

for size in [32, 64, 128, 256, 512]:
    residual = x

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

    residual = layers.Conv2D(
        size, 1, strides=2, padding="same", use_bias=False)(residual)
    x = layers.add([x, residual])

x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

We use the same data augmentation configuration as before.

Don't forget input rescaling!

In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.

Like in the original model, we add a dropout layer for regularization.

We apply a series of convolutional blocks with increasing feature depth. Each block consists of two batch-normalized depthwise separable convolution layers and a max pooling layer, with a residual connection around the entire block.

Note that the assumption that underlies separable convolution, "feature channels are largely independent," does not hold for RGB images! Red, green, and blue color channels are actually highly correlated in natural images. As such, the first layer in our model is a regular Conv2D layer. We'll start using SeparableConv2D afterwards.

This convnet has a trainable parameter count of 721,857, slightly lower than the 991,041 trainable parameters of the original model, but still in the same ballpark. Figure 9.11 shows its training and validation curves.

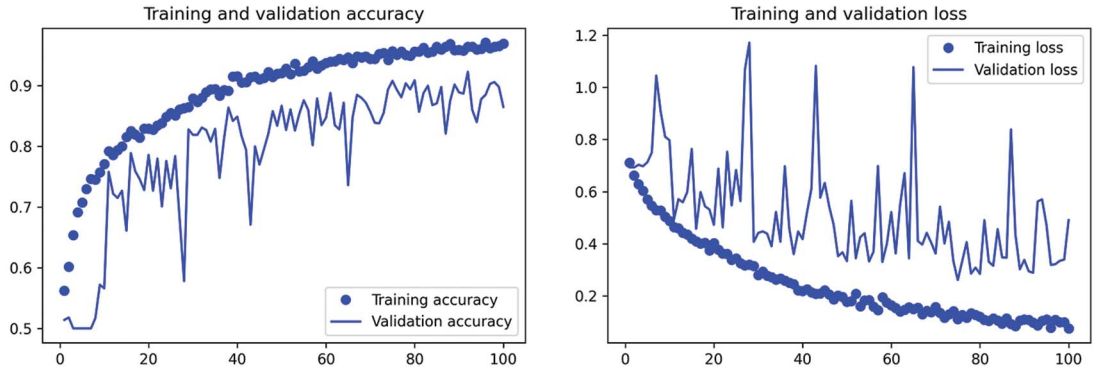


Figure 9.11 Training and validation metrics with an Xception-like architecture

You'll find that our new model achieves a test accuracy of 90.8%, compared to 83.5% for the naive model in the last chapter. As you can see, following architecture best practices does have an immediate, sizable impact on model performance!

At this point, if you want to further improve performance, you should start systematically tuning the hyperparameters of your architecture—a topic we'll cover in detail in chapter 13. We haven't gone through this step here, so the configuration of the preceding model is purely based on the best practices we discussed, plus, when it comes to gauging model size, a small amount of intuition.

Note that these architecture best practices are relevant to computer vision in general, not just image classification. For example, Xception is used as the standard convolutional base in DeepLabV3, a popular state-of-the-art image segmentation solution.[4]

This concludes our introduction to essential convnet architecture best practices. With these principles in hand, you'll be able to develop higher-performing models across a wide range of computer vision tasks. You're now well on your way to becoming a proficient computer vision practitioner. To further deepen your expertise, there's one last important topic we need to cover: interpreting how a model arrives at its predictions.

9.4 Interpreting what convnets learn

A fundamental problem when building a computer vision application is that of *interpretability*: *why* did your classifier think a particular image contained a fridge, when all you can see is a truck? This is especially relevant to use cases where deep learning is used to complement human expertise, such as in medical imaging use cases. We will end this chapter by getting you familiar with a range of different techniques for visualizing what convnets learn and understanding the decisions they make.

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

- *Visualizing intermediate convnet outputs (intermediate activations)*—Useful for understanding how successive convnet layers transform their input, and for getting a first idea of the meaning of individual convnet filters
- *Visualizing convnet filters*—Useful for understanding precisely what visual pattern or concept each filter in a convnet is receptive to

[4] Liang-Chieh Chen et al., "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation," ECCV (2018), https://arxiv.org/abs/1802.02611.

- *Visualizing heatmaps of class activation in an image*—Useful for understanding which parts of an image were identified as belonging to a given class, thus allowing you to localize objects in images

For the first method—activation visualization—we'll use the small convnet that we trained from scratch on the dogs-versus-cats classification problem in section 8.2. For the next two methods, we'll use a pretrained Xception model.

9.4.1 Visualizing intermediate activations

Visualizing intermediate activations consists of displaying the values returned by various convolution and pooling layers in a model, given a certain input (the output of a layer is often called its *activation*, the output of the activation function). This gives a view into how an input is decomposed into the different filters learned by the network. We want to visualize feature maps with three dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel as a 2D image. Let's start by loading the model that you saved in section 8.2:

```
>>> from tensorflow import keras
>>> model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
>>> model.summary()
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 180, 180, 3)       0
_________________________________________________________________
rescaling_1 (Rescaling)      (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 178, 178, 32)      896
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 89, 89, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 87, 87, 64)        18496
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 43, 43, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 41, 41, 128)       73856
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 18, 18, 256)       295168
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 9, 9, 256)         0
_________________________________________________________________
```

```
conv2d_9 (Conv2D)            (None, 7, 7, 256)         590080
_________________________________________________________________
flatten_1 (Flatten)          (None, 12544)             0
_________________________________________________________________
dropout (Dropout)            (None, 12544)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 12545
=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
_________________________________________________________________
```

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

Listing 9.6 Preprocessing a single image

```
from tensorflow import keras
import numpy as np

img_path = keras.utils.get_file(
    fname="cat.jpg",
    origin="https://img-datasets.s3.amazonaws.com/cat.jpg")

def get_img_array(img_path, target_size):
    img = keras.utils.load_img(
        img_path, target_size=target_size)
    array = keras.utils.img_to_array(img)
    array = np.expand_dims(array, axis=0)
    return array

img_tensor = get_img_array(img_path, target_size=(180, 180))
```

Download a test image.

Open the image file and resize it.

Turn the image into a float32 NumPy array of shape (180, 180, 3).

Add a dimension to transform the array into a "batch" of a single sample. Its shape is now (1, 180, 180, 3).

Let's display the picture (see figure 9.12).

Listing 9.7 Displaying the test picture

```
import matplotlib.pyplot as plt
plt.axis("off")
plt.imshow(img_tensor[0].astype("uint8"))
plt.show()
```

In order to extract the feature maps we want to look at, we'll create a Keras model that takes batches of images as input, and that outputs the activations of all convolution and pooling layers.

Figure 9.12 The test cat picture

Listing 9.8 Instantiating a model that returns layer activations

```
from tensorflow.keras import layers

layer_outputs = []
layer_names = []
for layer in model.layers:
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)):
        layer_outputs.append(layer.output)
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=model.input, outputs=layer_outputs)
```

Extract the outputs of all Conv2D and MaxPooling2D layers and put them in a list.

Save the layer names for later.

Create a model that will return these outputs, given the model input.

When fed an image input, this model returns the values of the layer activations in the original model, as a list. This is the first time you've encountered a multi-output model in this book in practice since you learned about them in chapter 7; until now, the models you've seen have had exactly one input and one output. This one has one input and nine outputs: one output per layer activation.

Listing 9.9 Using the model to compute layer activations

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

Return a list of nine NumPy arrays: one array per layer activation.

For instance, this is the activation of the first convolution layer for the cat image input:

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 178, 178, 32)
```

It's a 178 × 178 feature map with 32 channels. Let's try plotting the fifth channel of the activation of the first layer of the original model (see figure 9.13).

Listing 9.10 Visualizing the fifth channel

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 5], cmap="viridis")
```

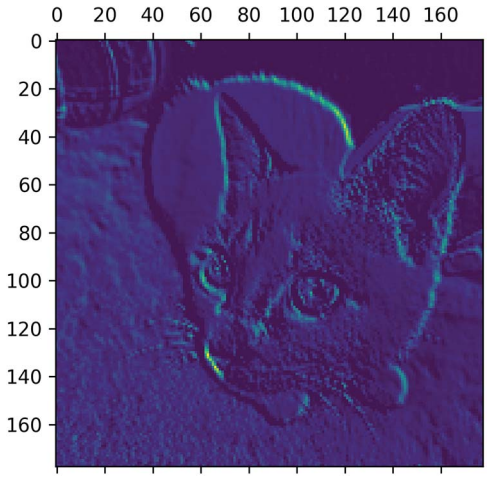


Figure 9.13 Fifth channel of the activation of the first layer on the test cat picture

This channel appears to encode a diagonal edge detector—but note that your own channels may vary, because the specific filters learned by convolution layers aren't deterministic.

Now, let's plot a complete visualization of all the activations in the network (see figure 9.14). We'll extract and plot every channel in each of the layer activations, and we'll stack the results in one big grid, with channels stacked side by side.

Listing 9.11 Visualizing every channel in every intermediate activation

```
images_per_row = 16
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]
    size = layer_activation.shape[1]
    n_cols = n_features // images_per_row
    display_grid = np.zeros(((size + 1) * n_cols - 1,
                             images_per_row * (size + 1) - 1))
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_index = col * images_per_row + row
            channel_image = layer_activation[0, :, :, channel_index].copy()
```

Iterate over the activations (and the names of the corresponding layers).

The layer activation has shape (l, size, size, n_features).

Prepare an empty grid for displaying all the channels in this activation.

This is a single channel (or feature).

Normalize channel values within the [0, 255] range. All-zero channels are kept at zero.

```
            if channel_image.sum() != 0:
                channel_image -= channel_image.mean()
                channel_image /= channel_image.std()
                channel_image *= 64
                channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype("uint8")
            display_grid[
                col * (size + 1): (col + 1) * size + col,
                row * (size + 1) : (row + 1) * size + row] = channel_image
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.axis("off")
    plt.imshow(display_grid, aspect="auto", cmap="viridis")
```

Place the channel matrix in the empty grid we prepared.

Display the grid for the layer.

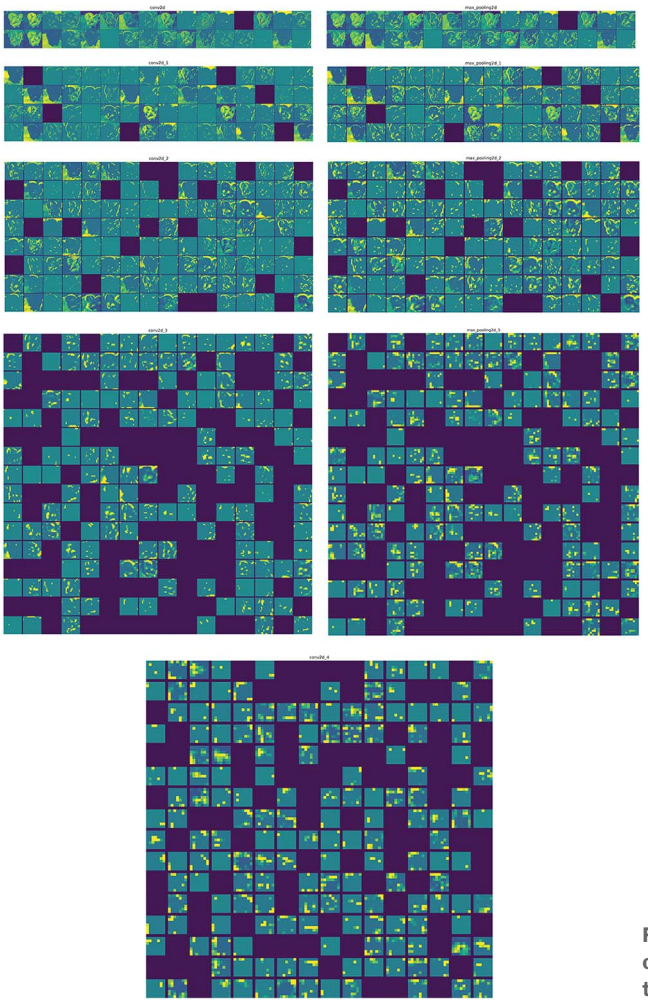


Figure 9.14 Every channel of every layer activation on the test cat picture

There are a few things to note here:

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

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

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (bicycle, tree) but can't remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from memory, chances are you couldn't get it even remotely right, even though you've seen thousands of bicycles in your lifetime (see, for example, figure 9.15). Try it right now: this effect is absolutely real. Your brain has learned to completely abstract its visual input—to transform it

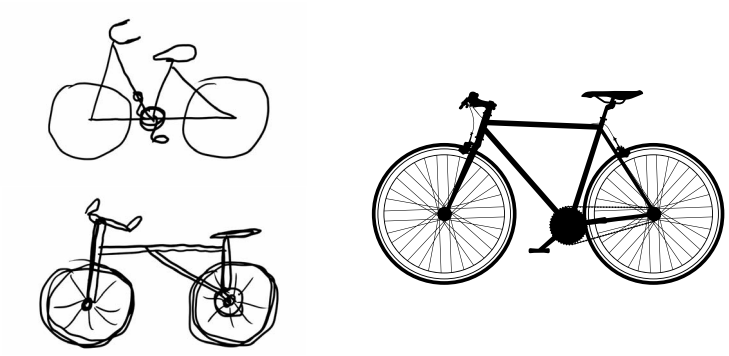


Figure 9.15 Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.

into high-level visual concepts while filtering out irrelevant visual details—making it tremendously difficult to remember how things around you look.

9.4.2 Visualizing convnet filters

Another easy way to inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to. This can be done with *gradient ascent in input space*: applying *gradient descent* to the value of the input image of a convnet so as to *maximize* the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.

Let's try this with the filters of the Xception model, pretrained on ImageNet. The process is simple: we'll build a loss function that maximizes the value of a given filter in a given convolution layer, and then we'll use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. This will be our second example of a low-level gradient descent loop leveraging the `GradientTape` object (the first one was in chapter 2).

First, let's instantiate the Xception model, loaded with weights pretrained on the ImageNet dataset.

Listing 9.12 Instantiating the Xception convolutional base

```
model = keras.applications.xception.Xception(
    weights="imagenet",
    include_top=False)
```

The classification layers are irrelevant for this use case, so we don't include the top stage of the model.

We're interested in the convolutional layers of the model—the `Conv2D` and `SeparableConv2D` layers. We'll need to know their names so we can retrieve their outputs. Let's print their names, in order of depth.

Listing 9.13 Printing the names of all convolutional layers in Xception

```
for layer in model.layers:
    if isinstance(layer, (keras.layers.Conv2D, keras.layers.SeparableConv2D)):
        print(layer.name)
```

You'll notice that the `SeparableConv2D` layers here are all named something like `block6_sepconv1`, `block7_sepconv2`, etc. Xception is structured into blocks, each containing several convolutional layers.

Now, let's create a second model that returns the output of a specific layer—a *feature extractor* model. Because our model is a Functional API model, it is inspectable: we can query the `output` of one of its layers and reuse it in a new model. No need to copy the entire Xception code.

Listing 9.14 Creating a feature extractor model

```
layer_name = "block3_sepconv1"
layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output)
```

You could replace this with the name of any layer in the Xception convolutional base.

This is the layer object we're interested in.

We use model.input and layer.output to create a model that, given an input image, returns the output of our target layer.

To use this model, simply call it on some input data (note that Xception requires inputs to be preprocessed via the `keras.applications.xception.preprocess_input` function).

Listing 9.15 Using the feature extractor

```
activation = feature_extractor(
    keras.applications.xception.preprocess_input(img_tensor)
)
```

Let's use our feature extractor model to define a function that returns a scalar value quantifying how much a given input image "activates" a given filter in the layer. This is the "loss function" that we'll maximize during the gradient ascent process:

```
import tensorflow as tf

def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return tf.reduce_mean(filter_activation)
```

The loss function takes an image tensor and the index of the filter we are considering (an integer).

Return the mean of the activation values for the filter.

Note that we avoid border artifacts by only involving non-border pixels in the loss; we discard the first two pixels along the sides of the activation.

The difference between `model.predict(x)` and `model(x)`

In the previous chapter, we used `predict(x)` for feature extraction. Here, we're using `model(x)`. What gives?

Both `y = model.predict(x)` and `y = model(x)` (where `x` is an array of input data) mean "run the model on `x` and retrieve the output `y`." Yet they aren't exactly the same thing.

`predict()` loops over the data in batches (in fact, you can specify the batch size via `predict(x, batch_size=64)`), and it extracts the NumPy value of the outputs. It's schematically equivalent to this:

```
def predict(x):
    y_batches = []
    for x_batch in get_batches(x):
```

(continued)

```
        y_batch = model(x).numpy()
        y_batches.append(y_batch)
    return np.concatenate(y_batches)
```

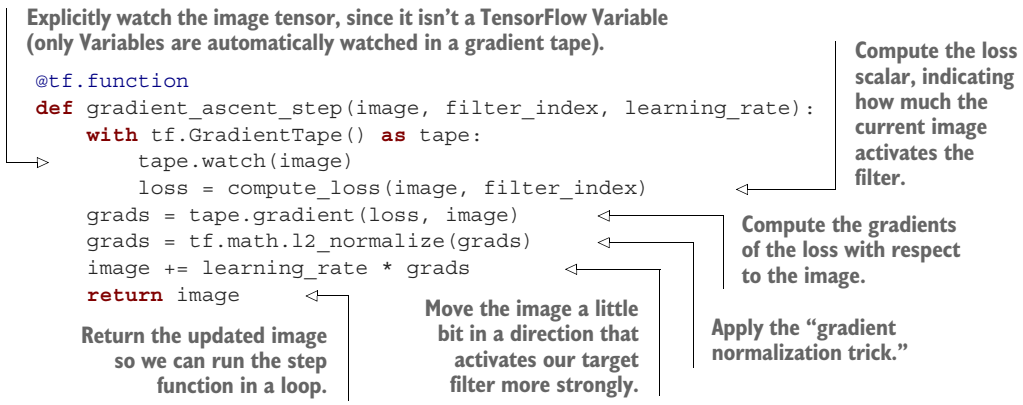
This means that `predict()` calls can scale to very large arrays. Meanwhile, `model(x)` happens in-memory and doesn't scale. On the other hand, `predict()` is not differentiable: you cannot retrieve its gradient if you call it in a `GradientTape` scope.

You should use `model(x)` when you need to retrieve the gradients of the model call, and you should use `predict()` if you just need the output value. In other words, always use `predict()` unless you're in the middle of writing a low-level gradient descent loop (as we are now).

Let's set up the gradient ascent step function, using the `GradientTape`. Note that we'll use a `@tf.function` decorator to speed it up.

A non-obvious trick to help the gradient descent process go smoothly is to normalize the gradient tensor by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor). This ensures that the magnitude of the updates done to the input image is always within the same range.

Listing 9.16 Loss maximization via stochastic gradient ascent



Now we have all the pieces. Let's put them together into a Python function that takes as input a layer name and a filter index, and returns a tensor representing the pattern that maximizes the activation of the specified filter.

Listing 9.17 Function to generate filter visualizations

```
img_width = 200
img_height = 200
```

```
def generate_filter_pattern(filter_index):
    iterations = 30
    learning_rate = 10.
    image = tf.random.uniform(
        minval=0.4,
        maxval=0.6,
        shape=(1, img_width, img_height, 3))
    for i in range(iterations):
        image = gradient_ascent_step(image, filter_index, learning_rate)
    return image[0].numpy()
```

Amplitude of a single step

Number of gradient ascent steps to apply

Initialize an image tensor with random values (the Xception model expects input values in the [0, 1] range, so here we pick a range centered on 0.5).

Repeatedly update the values of the image tensor so as to maximize our loss function.

The resulting image tensor is a floating-point array of shape `(200, 200, 3)`, with values that may not be integers within `[0, 255]`. Hence, we need to post-process this tensor to turn it into a displayable image. We do so with the following straightforward utility function.

Listing 9.18 Utility function to convert a tensor into a valid image

```
def deprocess_image(image):
    image -= image.mean()
    image /= image.std()
    image *= 64
    image += 128
    image = np.clip(image, 0, 255).astype("uint8")
    image = image[25:-25, 25:-25, :]
    return image
```

Normalize image values within the [0, 255] range.

Center crop to avoid border artifacts.

Let's try it (see figure 9.16):

```
>>> plt.axis("off")
>>> plt.imshow(deprocess_image(generate_filter_pattern(filter_index=2)))
```

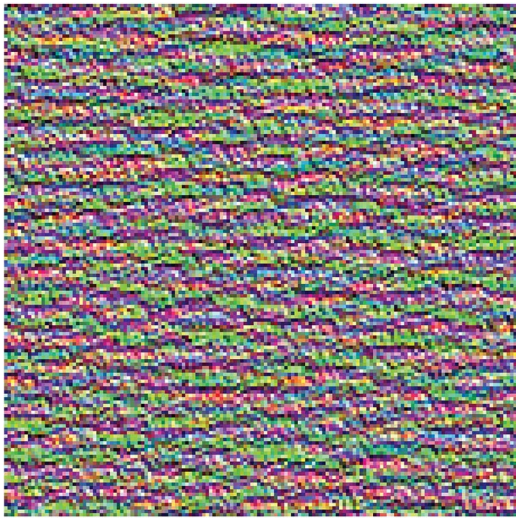


Figure 9.16 Pattern that the second channel in layer `block3_sepconv1` responds to maximally

It seems that filter 0 in layer `block3_sepconv1` is responsive to a horizontal lines pattern, somewhat water-like or fur-like.

Now the fun part: you can start visualizing every filter in the layer, and even every filter in every layer in the model.

Listing 9.19 Generating a grid of all filter response patterns in a layer

```
all_images = []
for filter_index in range(64):
    print(f"Processing filter {filter_index}")
    image = deprocess_image(
        generate_filter_pattern(filter_index)
    )
    all_images.append(image)

margin = 5
n = 8
cropped_width = img_width - 25 * 2
cropped_height = img_height - 25 * 2
width = n * cropped_width + (n - 1) * margin
height = n * cropped_height + (n - 1) * margin
stitched_filters = np.zeros((width, height, 3))

for i in range(n):
    for j in range(n):
        image = all_images[i * n + j]
        stitched_filters[
            row_start = (cropped_width + margin) * i
            row_end = (cropped_width + margin) * i + cropped_width
            column_start = (cropped_height + margin) * j
            column_end = (cropped_height + margin) * j + cropped_height

            stitched_filters[
                row_start: row_end,
                column_start: column_end, :] = image

keras.utils.save_img(
    f"filters_for_layer_{layer_name}.png", stitched_filters)
```

Generate and save visualizations for the first 64 filters in the layer.

Prepare a blank canvas for us to paste filter visualizations on.

Fill the picture with the saved filters.

Save the canvas to disk.

These filter visualizations (see figure 9.17) tell you a lot about how convnet layers see the world: each layer in a convnet learns a collection of filters such that their inputs can be expressed as a combination of the filters. This is similar to how the Fourier transform decomposes signals onto a bank of cosine functions. The filters in these convnet filter banks get increasingly complex and refined as you go deeper in the model:

- The filters from the first layers in the model encode simple directional edges and colors (or colored edges, in some cases).
- The filters from layers a bit further up the stack, such as `block4_sepconv1`, encode simple textures made from combinations of edges and colors.
- The filters in higher layers begin to resemble textures found in natural images: feathers, eyes, leaves, and so on.

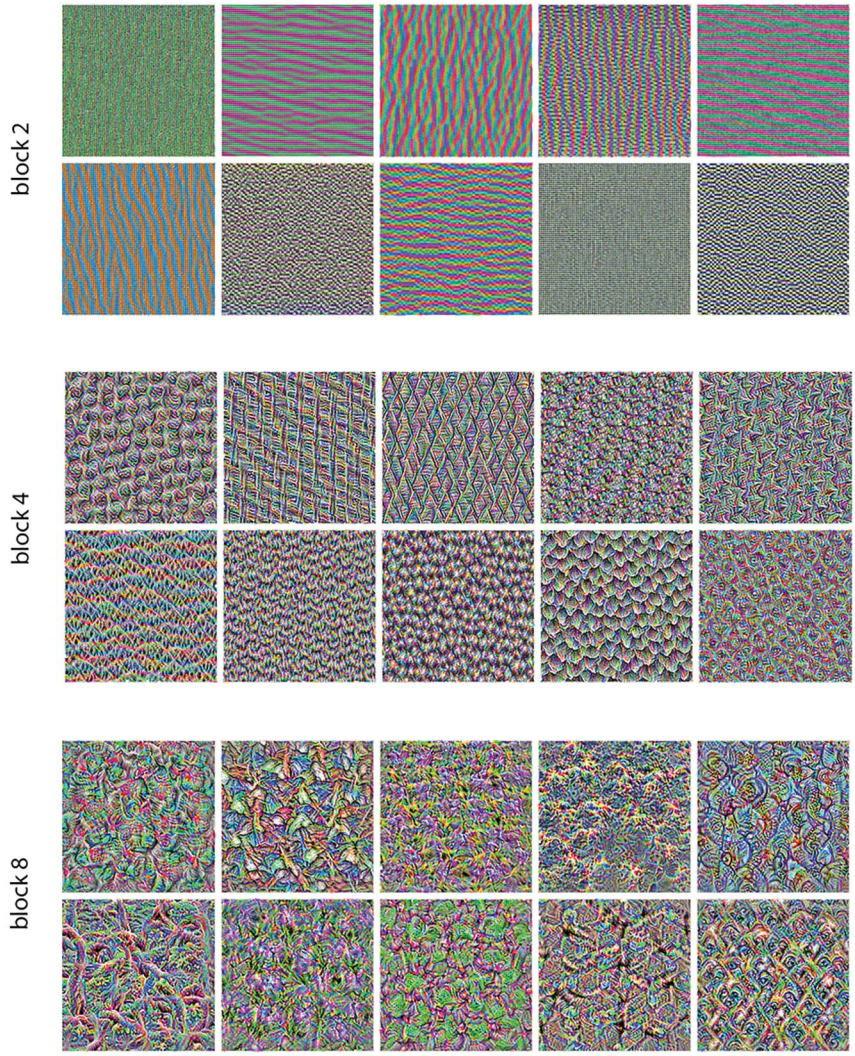


Figure 9.17 Some filter patterns for layers `block2_sepconv1`, `block4_sepconv1`, and `block8_sepconv1`

9.4.3 Visualizing heatmaps of class activation

We'll introduce one last visualization technique—one that is useful for understanding which parts of a given image led a convnet to its final classification decision. This is helpful for "debugging" the decision process of a convnet, particularly in the case of a classification mistake (a problem domain called *model interpretability*). It can also allow you to locate specific objects in an image.

This general category of techniques is called *class activation map* (CAM) visualization, and it consists of producing heatmaps of class activation over input images. A

class activation heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class under consideration. For instance, given an image fed into a dogs-versus-cats convnet, CAM visualization would allow you to generate a heatmap for the class "cat," indicating how cat-like different parts of the image are, and also a heatmap for the class "dog," indicating how dog-like parts of the image are.

The specific implementation we'll use is the one described in an article titled "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization."[5]

Grad-CAM consists of taking the output feature map of a convolution layer, given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel. Intuitively, one way to understand this trick is to imagine that you're weighting a spatial map of "how intensely the input image activates different channels" by "how important each channel is with regard to the class," resulting in a spatial map of "how intensely the input image activates the class."

Let's demonstrate this technique using the pretrained Xception model.

Listing 9.20 Loading the Xception network with pretrained weights

```
model = keras.applications.xception.Xception(weights="imagenet")
```

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

Consider the image of two African elephants shown in figure 9.18, possibly a mother and her calf, strolling on the savanna. Let's convert this image into something the Xception model can read: the model was trained on images of size 299 × 299, preprocessed according to a few rules that are packaged in the `keras.applications.xception.preprocess_input` utility function. So we need to load the image, resize it to 299 × 299, convert it to a NumPy `float32` tensor, and apply these preprocessing rules.

Listing 9.21 Preprocessing an input image for Xception

```
img_path = keras.utils.get_file(
    fname="elephant.jpg",
    origin="https://img-datasets.s3.amazonaws.com/elephant.jpg")

def get_img_array(img_path, target_size):
    img = keras.utils.load_img(img_path, target_size=target_size)
    array = keras.utils.img_to_array(img)
    array = np.expand_dims(array, axis=0)
    array = keras.applications.xception.preprocess_input(array)
    return array

img_array = get_img_array(img_path, target_size=(299, 299))
```

Download the image and store it locally under the path img_path.

Return a Python Imaging Library (PIL) image of size 299 × 299.

Return a float32 NumPy array of shape (299, 299, 3).

Add a dimension to transform the array into a batch of size (1, 299, 299, 3).

Preprocess the batch (this does channel-wise color normalization).

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

Figure 9.18 Test picture of African elephants

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

```
>>> preds = model.predict(img_array)
>>> print(keras.applications.xception.decode_predictions(preds, top=3)[0])
[("n02504458", "African_elephant", 0.8699266),
 ("n01871265", "tusker", 0.076968715),
 ("n02504013", "Indian_elephant", 0.02353728)]
```

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

- African elephant (with 87% probability)
- Tusker (with 7% probability)
- Indian elephant (with 2% probability)

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

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

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

First, we create a model that maps the input image to the activations of the last convolutional layer.

Listing 9.22 Setting up a model that returns the last convolutional output

```
last_conv_layer_name = "block14_sepconv2_act"
classifier_layer_names = [
    "avg_pool",
    "predictions",
]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

Second, we create a model that maps the activations of the last convolutional layer to the final class predictions.

Listing 9.23 Reapplying the classifier on top of the last convolutional output

```
classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
x = classifier_input
for layer_name in classifier_layer_names:
    x = model.get_layer(layer_name)(x)
classifier_model = keras.Model(classifier_input, x)
```

Then we compute the gradient of the top predicted class for our input image with respect to the activations of the last convolution layer.

Listing 9.24 Retrieving the gradients of the top predicted class

```
import tensorflow as tf

with tf.GradientTape() as tape:
    last_conv_layer_output = last_conv_layer_model(img_array)
    tape.watch(last_conv_layer_output)
    preds = classifier_model(last_conv_layer_output)
    top_pred_index = tf.argmax(preds[0])
    top_class_channel = preds[:, top_pred_index]

grads = tape.gradient(top_class_channel, last_conv_layer_output)
```

Compute activations of the last conv layer and make the tape watch it.

Retrieve the activation channel corresponding to the top predicted class.

This is the gradient of the top predicted class with regard to the output feature map of the last convolutional layer.

Now we apply pooling and importance weighting to the gradient tensor to obtain our heatmap of class activation.

Listing 9.25 Gradient pooling and channel-importance weighting

```
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2)).numpy()
last_conv_layer_output = last_conv_layer_output.numpy()[0]
for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i]
heatmap = np.mean(last_conv_layer_output, axis=-1)
```

This is a vector where each entry is the mean intensity of the gradient for a given channel. It quantifies the importance of each channel with regard to the top predicted class.

Multiply each channel in the output of the last convolutional layer by "how important this channel is."

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

For visualization purposes, we'll also normalize the heatmap between 0 and 1. The result is shown in figure 9.19.

Listing 9.26 Heatmap post-processing

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

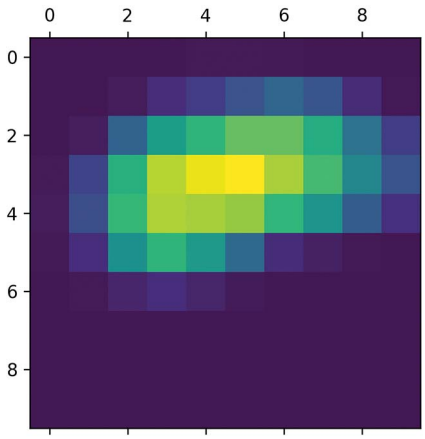


Figure 9.19 Standalone class activation heatmap

Finally, let's generate an image that superimposes the original image on the heatmap we just obtained (see figure 9.20).

Listing 9.27 Superimposing the heatmap on the original picture

```
import matplotlib.cm as cm

img = keras.utils.load_img(img_path)
img = keras.utils.img_to_array(img)
```

Load the original image.

Rescale the heatmap to the range 0–255.

```
heatmap = np.uint8(255 * heatmap)

jet = cm.get_cmap("jet")
jet_colors = jet(np.arange(256))[:, :3]
jet_heatmap = jet_colors[heatmap]
```

Use the "jet" colormap to recolorize the heatmap.

```
jet_heatmap = keras.utils.array_to_img(jet_heatmap)
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
jet_heatmap = keras.utils.img_to_array(jet_heatmap)
```

Create an image that contains the recolorized heatmap.

```
superimposed_img = jet_heatmap * 0.4 + img
superimposed_img = keras.utils.array_to_img(superimposed_img)
```

Superimpose the heatmap and the original image, with the heatmap at 40% opacity.

```
save_path = "elephant_cam.jpg"
superimposed_img.save(save_path)
```

Save the superimposed image.

Figure 9.20 African elephant class activation heatmap over the test picture

This visualization technique answers two important questions:

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

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

Summary

- There are three essential computer vision tasks you can do with deep learning: image classification, image segmentation, and object detection.
- Following modern convnet architecture best practices will help you get the most out of your models. Some of these best practices include using residual connections, batch normalization, and depthwise separable convolutions.
- The representations that convnets learn are easy to inspect—convnets are the opposite of black boxes!
- You can generate visualizations of the filters learned by your convnets, as well as heatmaps of class activity.