Chapter 12. Deep Computer Vision Using Convolutional Neural Networks

Although IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, it wasn't until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it's just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how our sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s. Over the last 15 years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing. However, we will focus on visual applications for now.

In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using PyTorch. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including object detection (classifying multiple objects in an image and placing bounding boxes around them) and semantic segmentation (classifying each pixel according to the class of the object it belongs to).

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights into the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 12-1, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field.

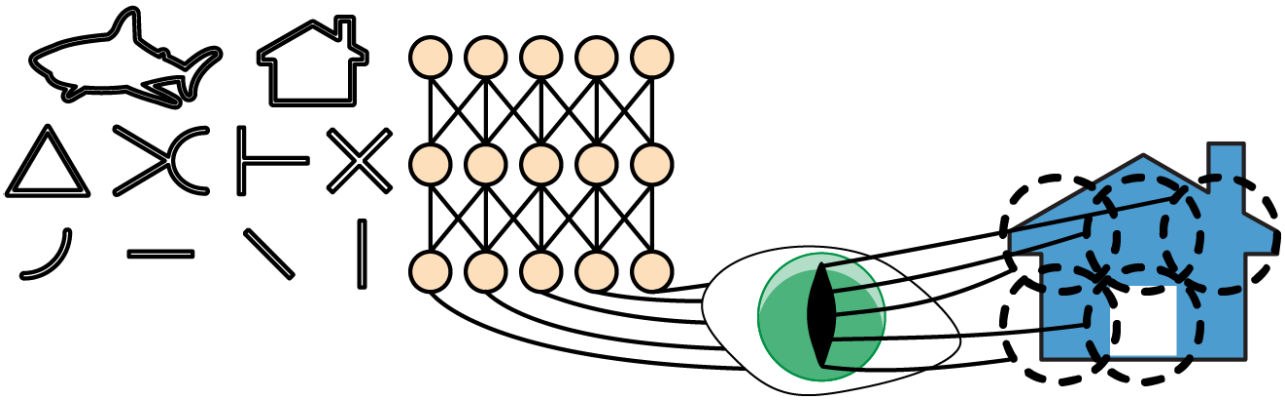


Figure 12-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 12-1, notice that each neuron is connected only to nearby neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

These studies of the visual cortex inspired the neocognitron,[4] introduced in 1980, which gradually evolved into what we now call convolutional neural networks. An important milestone was a 1998 paper[5] by Yann LeCun et al. that introduced the famous *LeNet-5* architecture, which became widely used by banks to recognize handwritten digits on checks. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.

NOTE

Why not simply use a deep neural network with fully connected layers for image recognition tasks? Unfortunately, although this works fine for small images (e.g., Fashion MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100 × 100–pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

Convolutional Layers

The most important building block of a CNN is the *convolutional layer*:[6] neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields (see Figure 12-2). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to

concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is well-suited to deal with composite objects, which are common in real-world images: this is one of the reasons why CNNs work so well for image recognition.

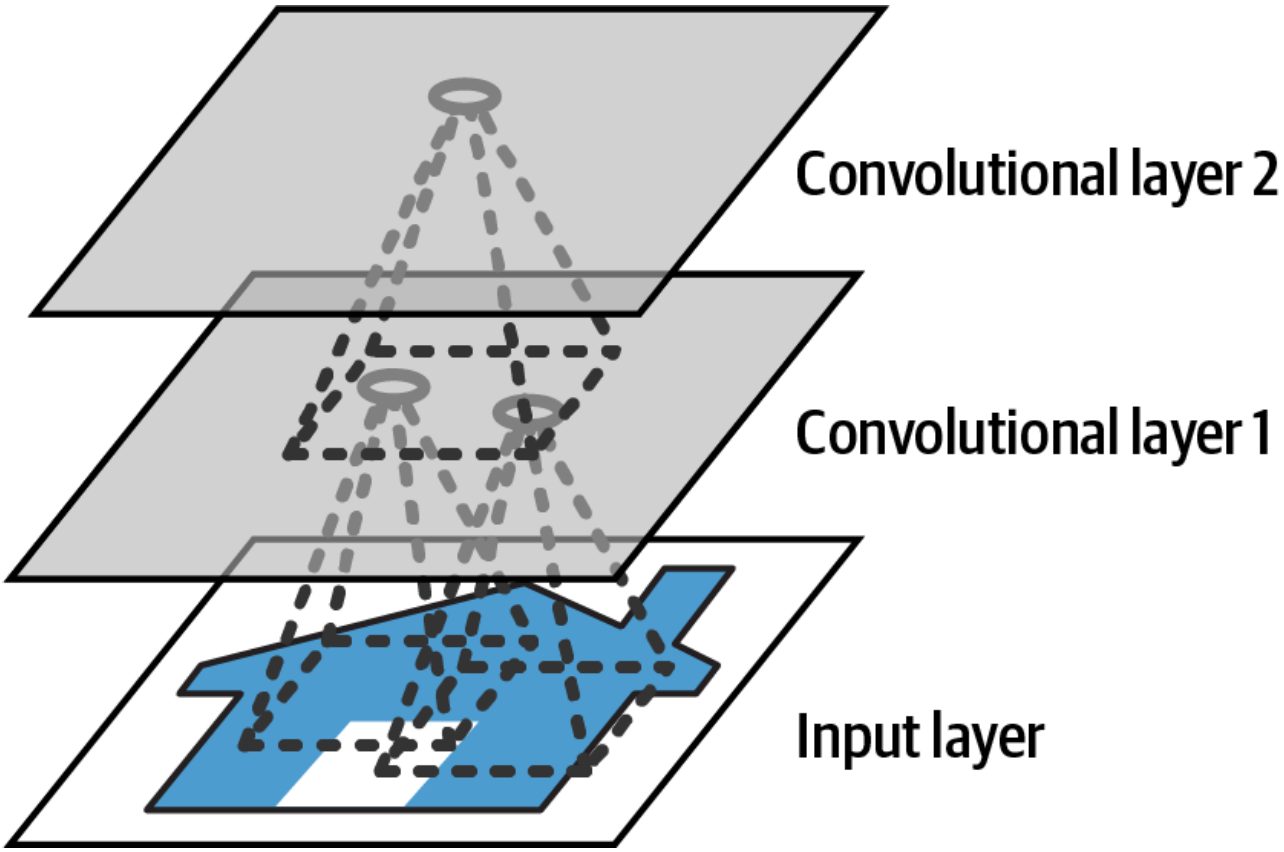


Figure 12-2. CNN layers with rectangular local receptive fields

---

NOTE

All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 12-3). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 12-4. This dramatically reduces the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the *stride*. In the diagram, a 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2. In this example the stride is the same in both directions, which is generally the case (although there are exceptions). A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and

horizontal strides.

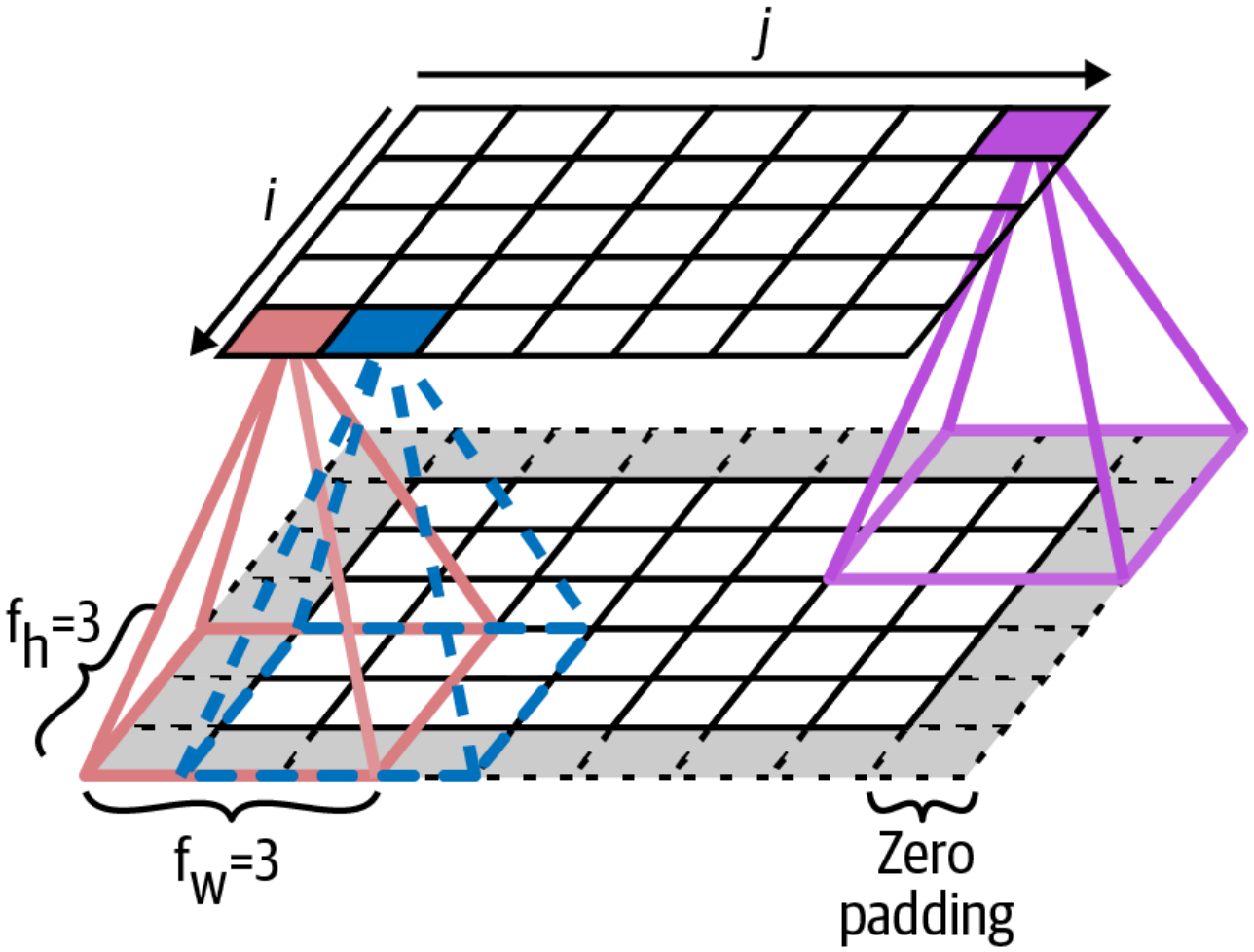


Figure 12-3. Connections between layers and zero padding

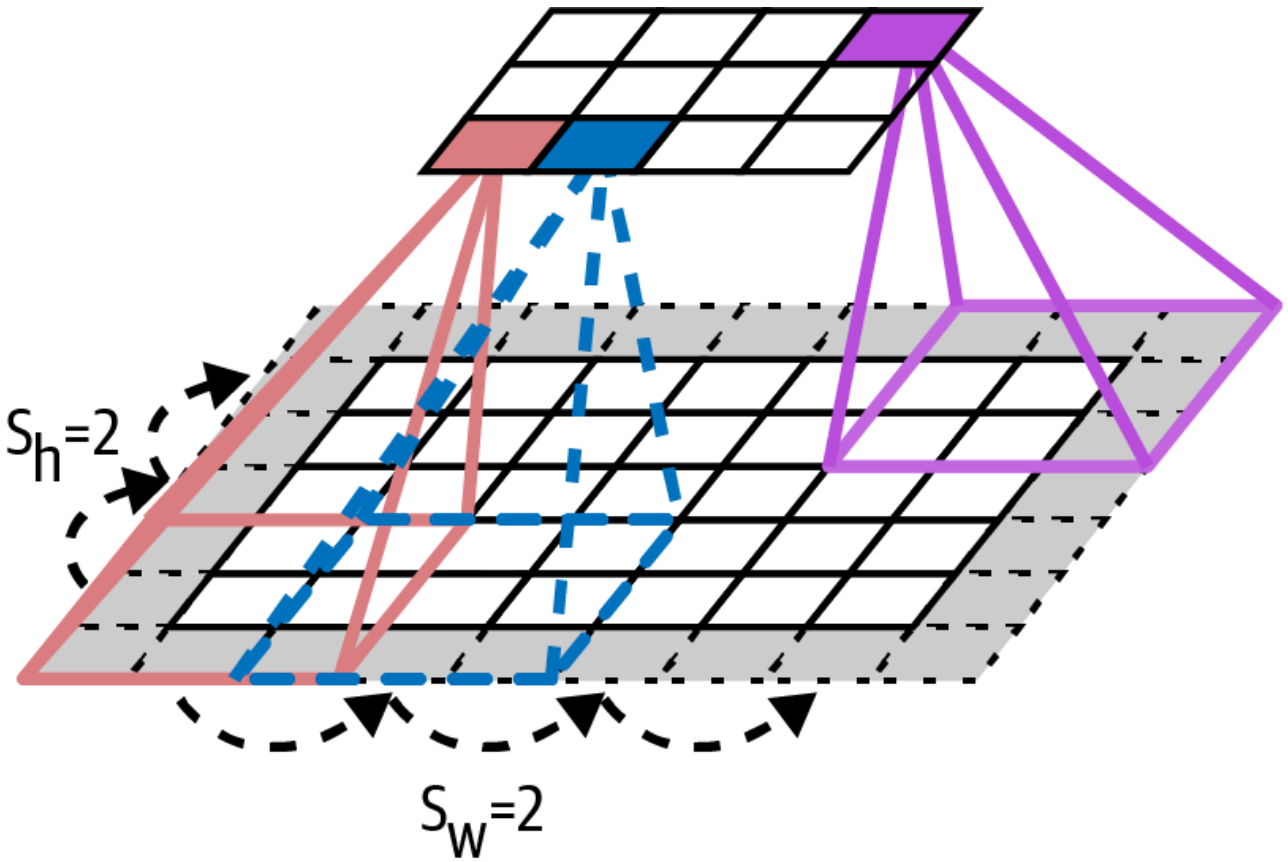


Figure 12-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 12-5 shows two possible sets of weights, called *filters* (or *convolution kernels*, or just *kernels*). The first one is represented as a black square with a vertical white line in the middle (it's a 7 × 7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will be multiplied by 0, except for the ones in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

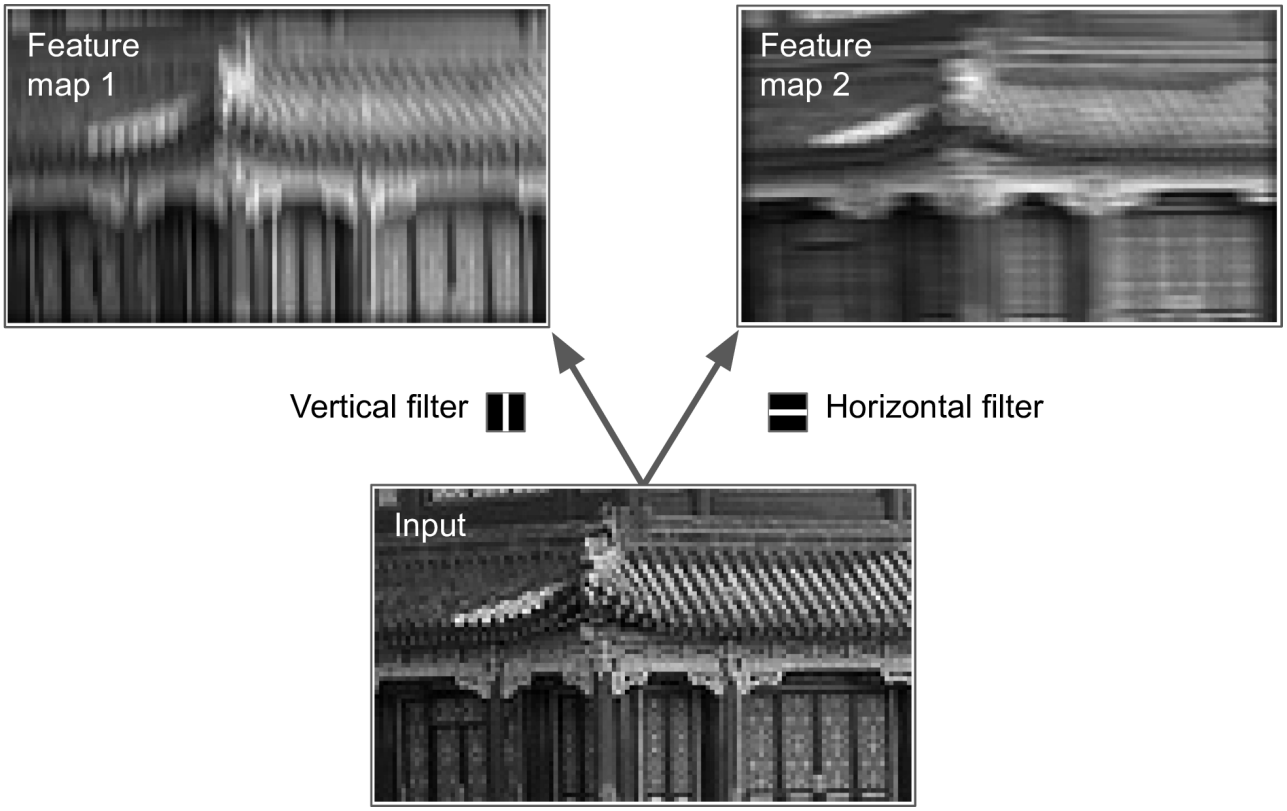


Figure 12-5. Applying two different filters to get two feature maps

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 12-5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. But don't worry, you won't have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

NOTE

In deep learning, we often build a single model that takes the raw inputs and produces the final outputs. This is called *end-to-end learning*. In contrast, classical vision systems would usually divide the system into a sequence of specialized modules.

Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and it outputs one feature map per filter, so the output is more accurately represented in 3D (see Figure 12-6). There is one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the feature maps of the previous layer. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

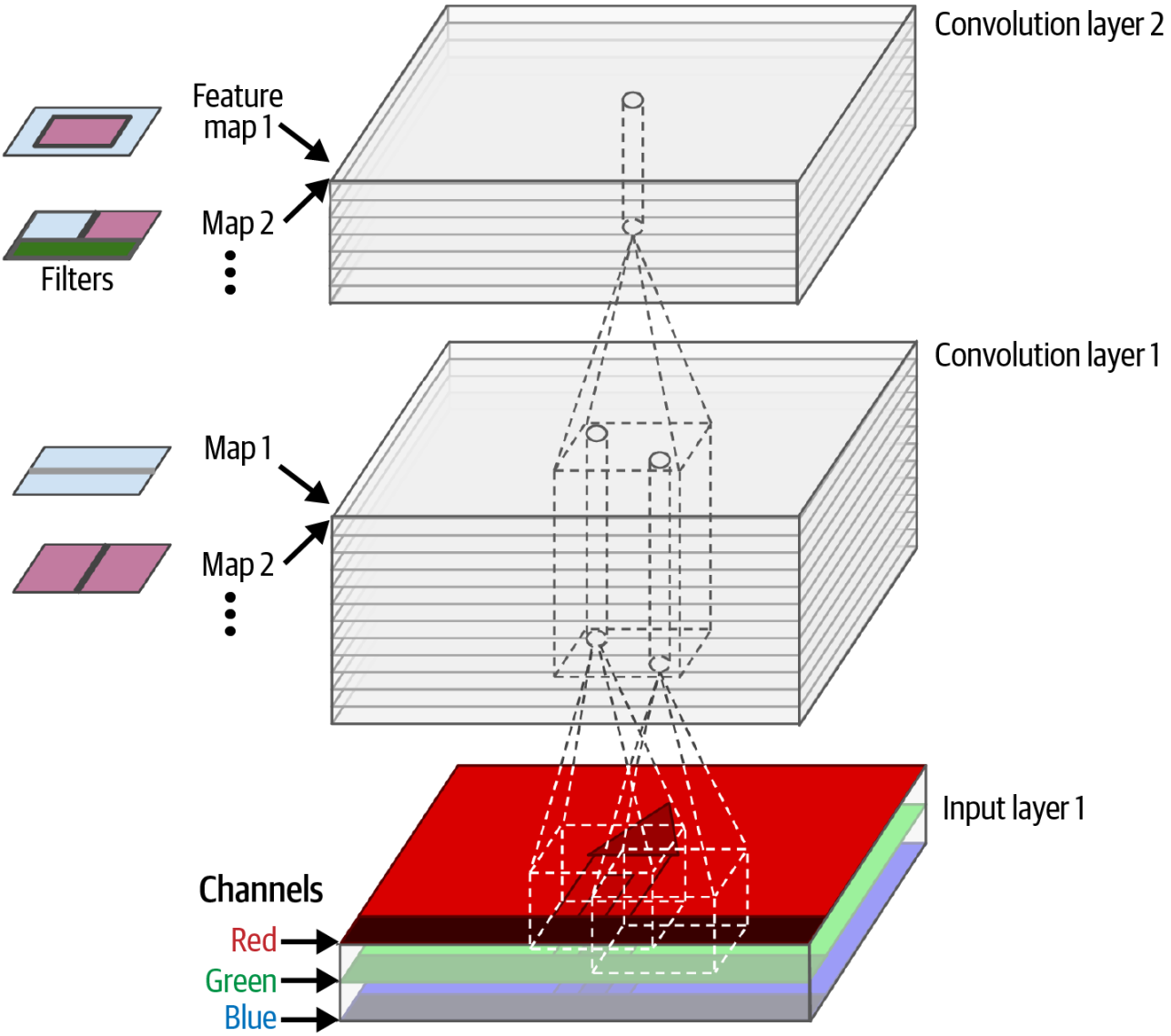


Figure 12-6. Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels; each convolutional layer outputs one feature map per filter

---

NOTE

The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. As mentioned in Chapter 8, there are typically three: red, green, and blue (RGB). Grayscale images have just one channel, but some images may have many more—for example, satellite images that capture extra light frequencies (such as infrared).

Specifically, a neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that, within a layer, all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Equation 12-1 summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer. It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term.

Equation 12-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \quad \text{with} \quad \begin{cases} i' = i \times sh + u \\ j' = j \times sw + v \end{cases}$$

In this equation:

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

Let's see how to create and use a convolutional layer using PyTorch.

Implementing Convolutional Layers with PyTorch

First, let's load a couple of sample images using Scikit-Learn's `load_sample_images()` function. The first image represents the tower of buddhist incense in China, while the second one represents a beautiful *Dahlia pinnata* flower. These images are represented as a Python list of NumPy unsigned byte arrays, so let's stack these images into a single NumPy array, then convert it to a 32-bit float tensor, and rescale the pixel values from 0–255 to 0–1:

```
import numpy as np
import torch
from sklearn.datasets import load_sample_images

sample_images = np.stack(load_sample_images()["images"])
sample_images = torch.tensor(sample_images, dtype=torch.float32) / 255
```

Let's look at this tensor's shape:

```
>>> sample_images.shape
torch.Size([2, 427, 640, 3])
```

We have two images, both are 427 pixels high and 640 pixels wide, and they have three color channels: red, green, and blue. As we saw in Chapter 10, PyTorch expects the channel dimension to be just *before* the height and width dimensions, not after, so we need to permute the dimensions using the `permute()` method:

```
>>> sample_images_permuted = sample_images.permute(0, 3, 1, 2)
>>> sample_images_permuted.shape
torch.Size([2, 3, 427, 640])
```

Let's also use TorchVision's `CenterCrop` class to center-crop the images:

```
>>> import torchvision
>>> import torchvision.transforms.v2 as T
>>> cropped_images = T.CenterCrop((70, 120))(sample_images_permuted)
>>> cropped_images.shape
torch.Size([2, 3, 70, 120])
```

Now let's create a 2D convolutional layer and feed it these cropped images to see what comes out. For this, PyTorch provides the `nn.Conv2d` layer. Under the hood, this layer relies on the `torch.nn.functional.conv2d()` function. Let's create a convolutional layer with 32 filters, each of size 7 × 7 (using `kernel_size=7`, which is equivalent to using `kernel_size=(7, 7)`), and apply this layer to our small batch of two images:

```
import torch.nn as nn

torch.manual_seed(42)
conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7)
fmaps = conv_layer(cropped_images)
```

NOTE

When we talk about a 2D convolutional layer, "2D" refers to the number of *spatial* dimensions (height and width), but as you can see, the layer takes 4D inputs: as we saw, the two additional dimensions are the batch size (first dimension) and the channels (second dimension).

Now let's look at the output's shape:

```
>>> fmaps.shape
torch.Size([2, 32, 64, 114])
```

The output shape is similar to the input shape, with two main differences. First, there are 32 channels instead of 3. This is because we set `out_channels=32`, so we get 32 output feature maps: instead of the intensity of red, green, and blue at each location, we now have the intensity of each feature at each location. Second, the height and width have both shrunk by 6 pixels. This is due to the fact that the `nn.Conv2d` layer does not use any zero-padding by default, which means that we lose a few pixels on the sides of the output feature maps, depending on the size of the filters. In this case, since the kernel size is 7, we lose 6 pixels horizontally and 6 pixels vertically (i.e., 3 pixels on each side).

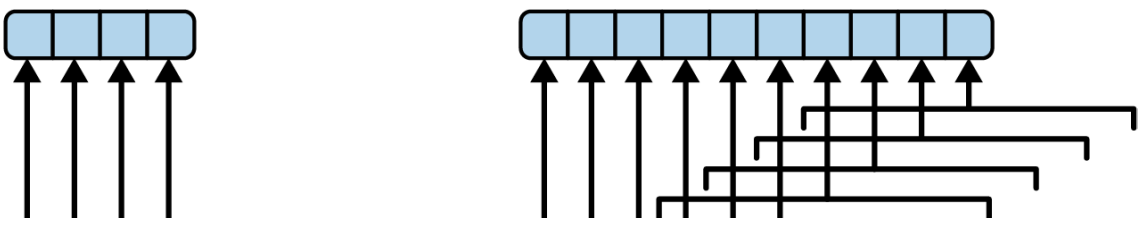
WARNING

By default, the `padding` hyperparameter is set to 0, which means that padding is turned off. Oddly, this is also called *valid padding* since every neuron's receptive field lies strictly within *valid* positions inside the input (it does not go out of bounds). You can actually set `padding="valid"`, which is equivalent to `padding=0`. It's not a PyTorch naming quirk: everyone uses this confusing nomenclature.

If instead we set `padding="same"`, then the inputs are padded with enough zeros on all sides to ensure that the output feature maps end up with the *same* size as the inputs (hence the name of this option):

```
>>> conv_layer = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7,
...                        padding="same")
...
>>> fmaps = conv_layer(cropped_images)
>>> fmaps.shape
torch.Size([2, 32, 70, 120])
```

These two padding options are illustrated in Figure 12-7. For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.



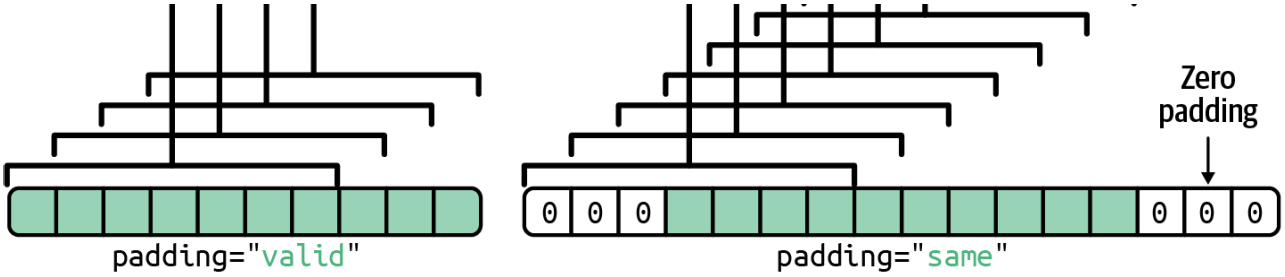


Figure 12-7. Two different padding options, with `stride=1` and `kernel_size=7`

If the stride is greater than 1 (in any direction), then the output size will be much smaller than the input size. For example, assuming the input size is 70 × 120, then if you set `stride=2` (or equivalently `stride=(2, 2)`), `padding=3`, and `kernel_size=7`, then the output feature maps will be 35 × 60: halved both vertically and horizontally. You could set a very large padding value to make the output size identical to the input size, but that's almost certainly a bad idea since it would drown your image in a sea of zeros (for this reason, PyTorch raises an exception if you set `padding="same"` along with a `stride` greater than 1). Figure 12-8 illustrates `stride=2`, with `kernel_size=7` and `padding` set to 0 or 3.

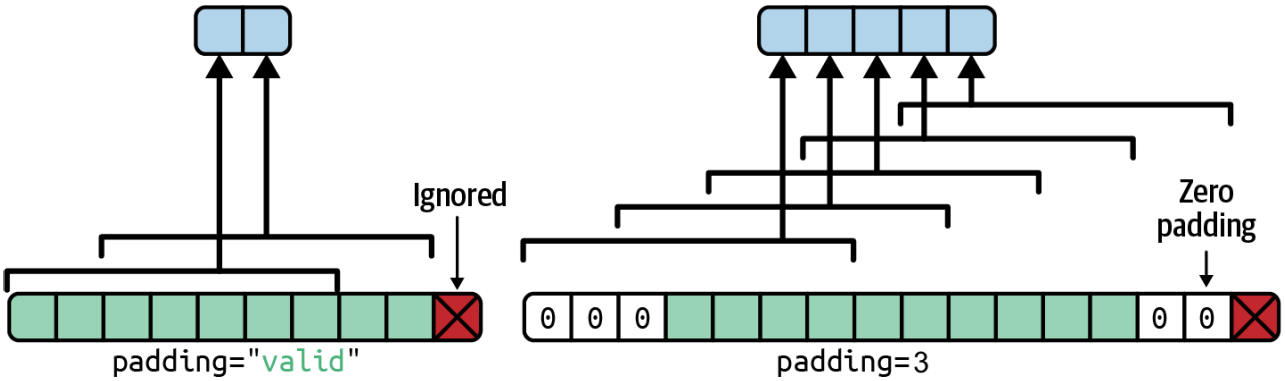


Figure 12-8. Two different padding options, with `stride=2` and `kernel_size=7`: the output size is much smaller

Now let's look at the layer's parameters (which were noted as $w_{u,\,v,\,k',\,k}$ and b_k in Equation 12-1). Just like a `nn.Linear` layer, a `nn.Conv2d` layer holds all the layer's parameters, including the kernels and biases, which are accessible via the `weight` and `bias` attributes:

```
>>> conv_layer.weight.shape
torch.Size([32, 3, 7, 7])
>>> conv_layer.bias.shape
torch.Size([32])
```

The `weight` tensor is 4D, and its shape is [*output_channels*, *input_channels*, *kernel_height*, *kernel_width*]. The `bias` tensor is 1D, with shape [*output_channels*]. The number of output channels is equal to the number of output feature maps, which is also equal to the number of filters. Most importantly, note that the height and width of the input images do not appear in the kernel's shape: this is because all the neurons in the output feature maps share the same weights, as explained earlier. This means that you can feed images of any size to this layer, as long as they are at least as large as the kernels, and if they have the right number of channels (three in this case).

It's important to add an activation function after each convolutional layer. This is for the same reason as for `nn.Linear` layers: a convolutional layer performs a linear operation, so if you stacked multiple convolutional layers without any activation functions, they would all be equivalent to a single convolutional layer, and they wouldn't be able to learn anything really complex.

Both the `weight` and `bias` parameters are initialized randomly, using a uniform distribution similar to the one used by the `nn.Linear` layer, between and , where k is the fan_{in}. In `nn.Conv2d`, $k = f_h \times f_w \times f_{n'}$, where f_h and f_w are the height and width of the kernel, and $f_{n'}$ is the number of input channels. As we saw in Chapter 11, you will generally want to reinitialize the weights depending on the activation function you use. For example, you should apply He initialization whenever you use the ReLU activation function. As for the biases, they can just be reinitialized to zero.

As you can see, convolutional layers have quite a few hyperparameters: the number of filters (`out_channels`), the kernel size, the type of padding, the strides, and the activation function. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later in this chapter to give you some idea of which hyperparameter values work best in practice.

Now, let's look at the second common building block of CNNs: the *pooling layer*.

Pooling Layers

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights or biases; all it does is aggregate the inputs using an aggregation function such as the max or mean. Figure 12-9 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2 × 2 *pooling kernel*,[7] with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in Figure 12-9, the input values are 1, 5, 3, and 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

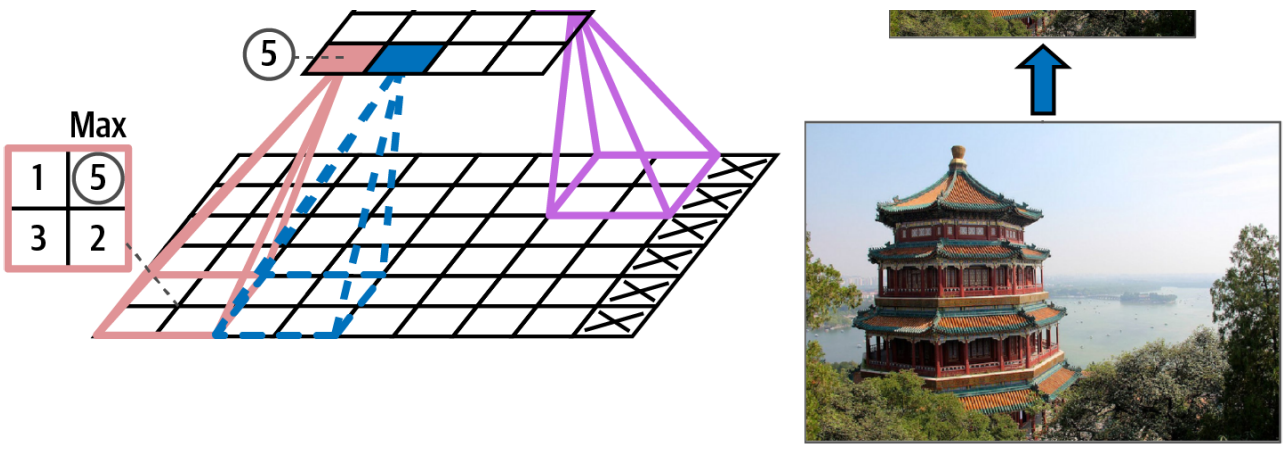


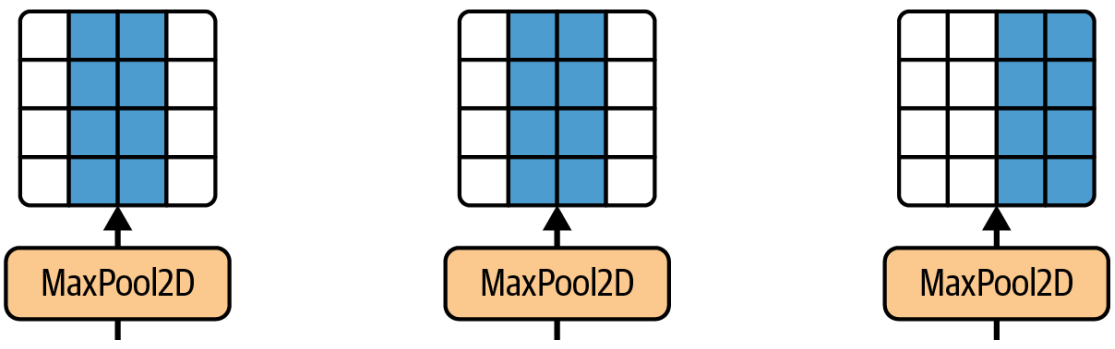
Figure 12-9. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

NOTE

A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) is the same as the input depth.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of *invariance* to small translations, as shown in Figure 12-10. Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2 × 2 kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 50% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

However, max pooling has some downsides too. It's obviously very destructive: even with a tiny 2 × 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), thereby dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation (the task of classifying each pixel in an image according to the object that pixel belongs to, which we'll explore later in this chapter): obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.



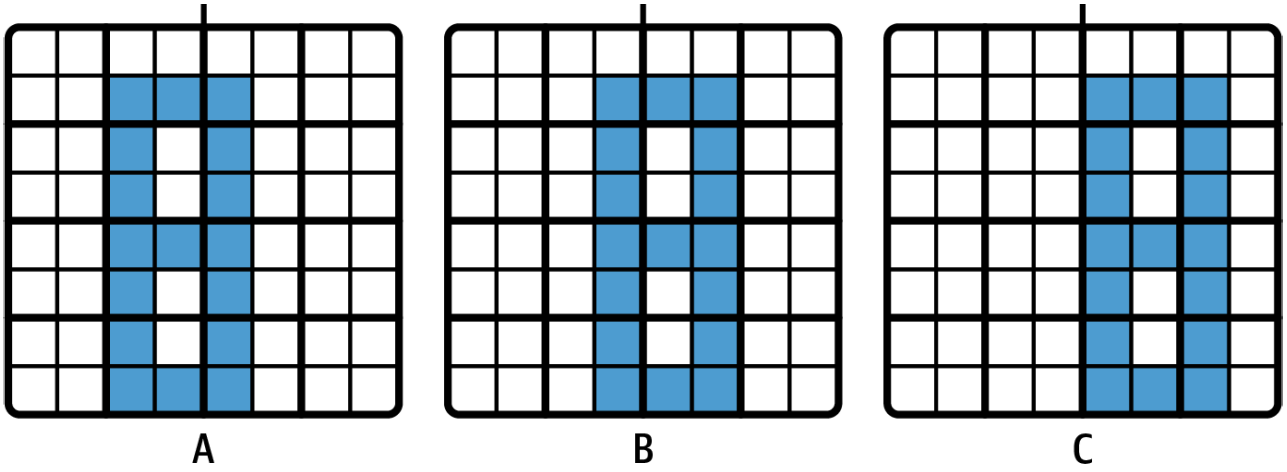


Figure 12-10. Invariance to small translations

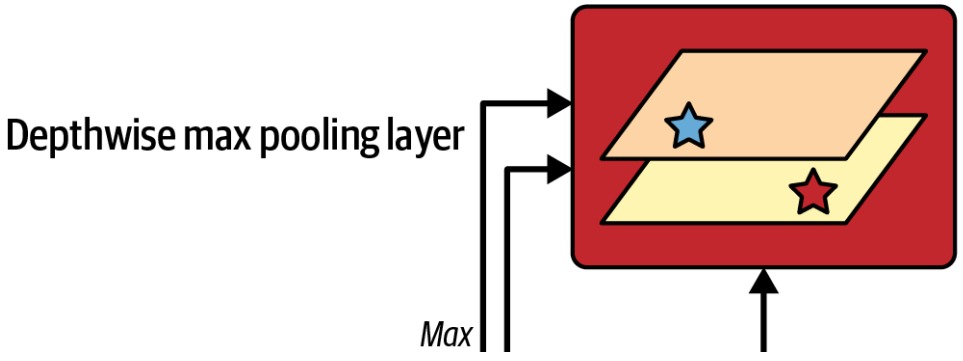
Implementing Pooling Layers with PyTorch

The following code creates a `nn.MaxPool2d` layer, using a 2 × 2 kernel. The strides default to the kernel size, so this layer uses a stride of 2 (horizontally and vertically). By default, it uses `padding=0` (i.e., "valid" padding):

```
max_pool = nn.MaxPool2d(kernel_size=2)
```

To create an *average pooling layer*, just use `nn.AvgPool2d`, instead of `nn.MaxPool2d`. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can also be performed along the depth dimension instead of the spatial dimensions, although it's not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits; see Figure 12-11), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything: thickness, brightness, skew, color, and so on.



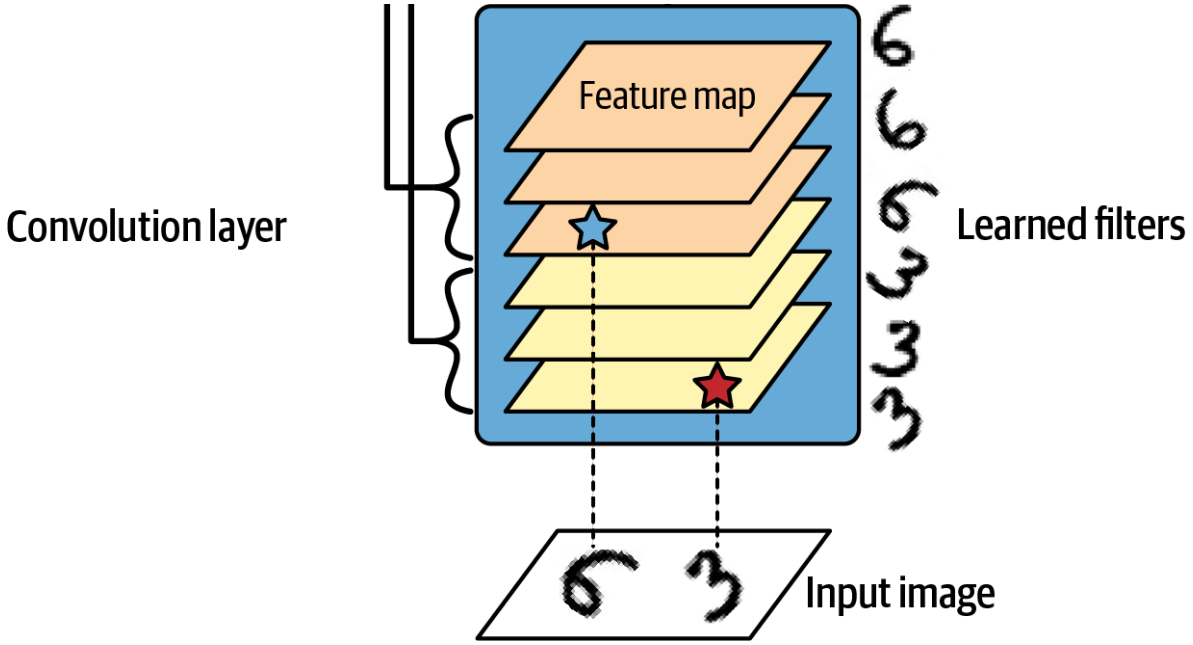


Figure 12-11. Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case)

PyTorch does not include a depthwise max pooling layer, but we can implement a custom module based on the `torch.nn.functional.max_pool1d()` function:

```
import torch.nn.functional as F

class DepthPool(torch.nn.Module):
    def __init__(self, kernel_size, stride=None, padding=0):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding

    def forward(self, inputs):
        batch, channels, height, width = inputs.shape
        Z = inputs.view(batch, channels, height * width)  # merge spatial dims
        Z = Z.permute(0, 2, 1)  # switch spatial and channels dims
        Z = F.max_pool1d(Z, kernel_size=self.kernel_size, stride=self.stride,
                         padding=self.padding)  # compute max pool
        Z = Z.permute(0, 2, 1)  # switch back spatial and channels dims
        return Z.view(batch, -1, height, width)  # unmerge spatial dims
```

For example, suppose the input batch contains two 70 × 120 images, each with 32 channels (i.e., the inputs have a shape of `[2, 32, 70, 120]`), and we use `kernel_size=4`, and the default `stride` (equal to `kernel_size`) and `padding=0`:

- The `forward()` method starts by merging the spatial dimensions, which gives us a tensor of shape `[2, 32, 8400]` (since 70 × 120 = 8,400).
- It then permutes the last two dimensions, so we get a shape of `[2, 8400, 32]`.
- Next, it uses the `max_pool1d()` function to compute the max pool along the last dimension, which corresponds to our original 32 channels. Since `kernel_size` and `stride` are both equal to 4, and we don't use any padding, the size of the last dimension gets divided by 4, so the resulting shape is `[2, 8400, 8]`.

- The function then permutes the last two dimensions again, giving us a shape of `[2, 8, 8400]`.
- Lastly, it separates the spatial dimensions to get the final shape of `[2, 8, 50, 100]`. You can verify that the output is exactly what we were after.

One last type of pooling layer that you will often see in modern architectures is the *global average pooling layer*. It works very differently: all it does is compute the mean of each entire feature map. Therefore it outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful just before the output layer, as you will see later in this chapter.

To create such a layer, one option is to use a regular `nn.AvgPool2d` layer and set its kernel size to the same size as the inputs. However, this is not very convenient since it requires knowing the exact dimensions of the inputs ahead of time. A simpler solution is to use the `nn.AdaptiveAvgPool2d` layer, which lets you specify the desired spatial dimensions of the output: it automatically adapts the kernel size (with an equal stride) to get the desired result, adding a bit of padding if needed. If we set the output size to 1, we get a global average pooling layer:

```
global_avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
output = global_avg_pool(cropped_images)
```

Alternatively, you could just use the `torch.mean()` function to get the same output:

```
output = cropped_images.mean(dim=(2, 3), keepdim=True)
```

Now you know all the building blocks to create convolutional neural networks. Let's see how to assemble them.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see Figure 12-12). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

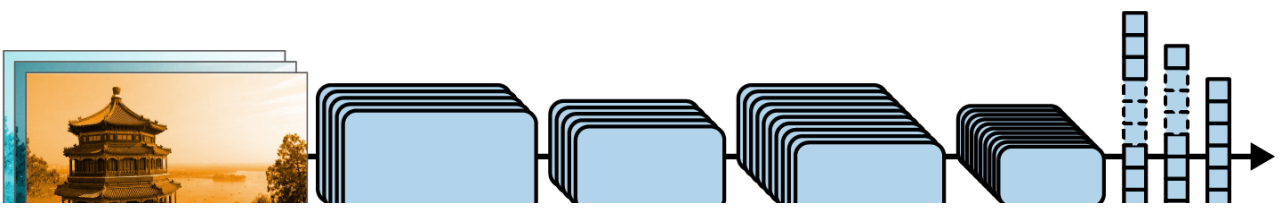


Figure 12-12. Typical CNN architecture

TIP

Instead of using a convolutional layer with a 5 × 5 kernel, it is generally preferable to stack two layers with 3 × 3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5 × 5 or 7 × 7), usually with a stride of 2 or more. This reduces the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how you can implement a basic CNN to tackle the Fashion MNIST dataset (introduced in Chapter 9):

```
from functools import partial

DefaultConv2d = partial(nn.Conv2d, kernel_size=3, padding="same")
model = nn.Sequential(
    DefaultConv2d(in_channels=1, out_channels=64, kernel_size=7), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=64, out_channels=128), nn.ReLU(),
    DefaultConv2d(in_channels=128, out_channels=128), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    DefaultConv2d(in_channels=128, out_channels=256), nn.ReLU(),
    DefaultConv2d(in_channels=256, out_channels=256), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),
    nn.Linear(in_features=2304, out_features=128), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=128, out_features=64), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(in_features=64, out_features=10),
).to(device)
```

Let's go through this code:

- We use the `functools.partial()` function (introduced in Chapter 11) to define `DefaultConv2d`, which acts just like `nn.Conv2d` but with different default arguments: a small kernel size of 3, and `"same"` padding. This avoids having to repeat these arguments throughout the model.
- Next, we create the `nn.Sequential` model. Its first layer is a `DefaultConv2d` with 64 fairly large filters (7 × 7). It uses the default stride of 1 because the input images are not very large. It also uses `in_channels=1` because the Fashion MNIST images have a single color channel (i.e., grayscale). Each convolutional layer is followed by the ReLU activation

function.

- We then add a max pooling layer with a kernel size of 2, so it divides each spatial dimension by a factor of 2 (rounded down if needed).
- Then we repeat the same structure twice: two convolutional layers followed by a max pooling layer. For larger images, we could repeat this structure several more times. The number of repetitions is a hyperparameter you can tune.
- Note that the number of filters doubles as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256). It makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.
- Next is the fully connected network, composed of two hidden dense layers (`nn.Linear`) with the ReLU activation function, plus a dense output layer. Since it's a classification task with 10 classes, the output layer has 10 units. As we did in Chapter 10, we leave out the softmax activation function, so the model will output logits rather than probabilities, and we must use the `nn.CrossEntropyLoss` to train the model. Note that we must flatten the inputs just before the first dense layer, since it expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

TIP

The first `nn.Linear` layer has 2,304 input features: where did this number come from? Well the Fashion MNIST images are 28 × 28 pixels, but the pooling layers shrink them to 14 × 14, then 7 × 7, and finally 3 × 3. Just before the first `nn.Linear` layer, there are 256 feature maps, so we end up with 256 × 3 × 3 = 2,304 input features. Figuring out the number of features can sometimes be a bit difficult, but one trick is to set `in_features` to some arbitrary value (say, 999), and let training crash. The correct number of features appears in the error message: "RuntimeError: mat1 and mat2 shapes cannot be multiplied (32x2304 and 999x128)". Another option is to use `nn.LazyLinear` instead of `nn.Linear`: it's just like the `nn.Linear` layer, except it only creates the weights matrix the first time it gets called: it can then automatically set the number of input features to the correct value. Other layers—such as convolutional layers and batch norm layers—also have lazy variants.

If you train this model on the Fashion MNIST training set, it should reach about 92% accuracy on the test set (you can use the `train()` and `evaluate_tm()` functions we defined in Chapter 10). It's not state of the art, but it is pretty good, and better than what we achieved with dense networks in Chapter 9.

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition, the error rate for image classification fell from over 26% to less than 2.3% in just 6 years. More precisely, this was the *top-*

five error rate, which is the ratio of test images for which the system's five most confident predictions did *not* include the correct answer. The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds!). Looking at the evolution of the winning entries is a good way to understand how CNNs work, and how research in deep learning progresses.

We will first look at the classical LeNet-5 architecture (1998), then several winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), ResNet (2015), and SENet (2017). We will also discuss a few more architectures, including VGGNet, Xception, ResNeXt, DenseNet, MobileNet, CSPNet, EfficientNet, and ConvNeXt (and we will discuss vision transformers in Chapter 16).

LeNet-5

The LeNet-5 architecture[8] is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in Table 12-1.

Table 12-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

As you can see, this looks pretty similar to our Fashion MNIST model: a stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh, and softmax instead of RBF (introduced in Chapter 2). There were several other minor differences that don't really matter much, but in case you are interested, they are listed in this chapter's notebook at *https://homl.info/colab-p*. Yann LeCun's website also features great demos of LeNet-5 classifying digits.

AlexNet

The AlexNet CNN architecture[9] won the 2012 ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best competitor achieved only 26%! AlexNet was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is simi-

developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. Table 12-2 presents this architecture.

Table 12-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6 × 6	3 × 3	2	`valid`	–
C7	Convolution	256	13 × 13	3 × 3	1	`same`	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	`same`	ReLU
S4	Max pooling	256	13 × 13	3 × 3	2	`valid`	–
C3	Convolution	256	27 × 27	5 × 5	1	`same`	ReLU
S2	Max pooling	96	27 × 27	3 × 3	2	`valid`	–
C1	Convolution	96	55 × 55	11 × 11	4	`valid`	ReLU
In	Input	3 (RGB)	227 × 227	–	–	–	–

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout (introduced in Chapter 11) with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they performed data augmentation by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

DATA AUGMENTATION

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. This reduces overfitting, making this a regularization technique. The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not. Simply adding white noise will not help; the modifications should be learnable (white noise is not).

For example, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 12-13). To do this, you can use tools available in `torchvision.transforms.v2` (e.g., `RandomCrop`, `RandomRotation`, etc.). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. You can similarly use transforms to tweak the colors, and contrasts to simulate many different lighting conditions. In general, you can also flip the pictures horizontally (except for text and other asymmetrical objects). By combining these transformations (using `Compose`), you can greatly increase your training set size.

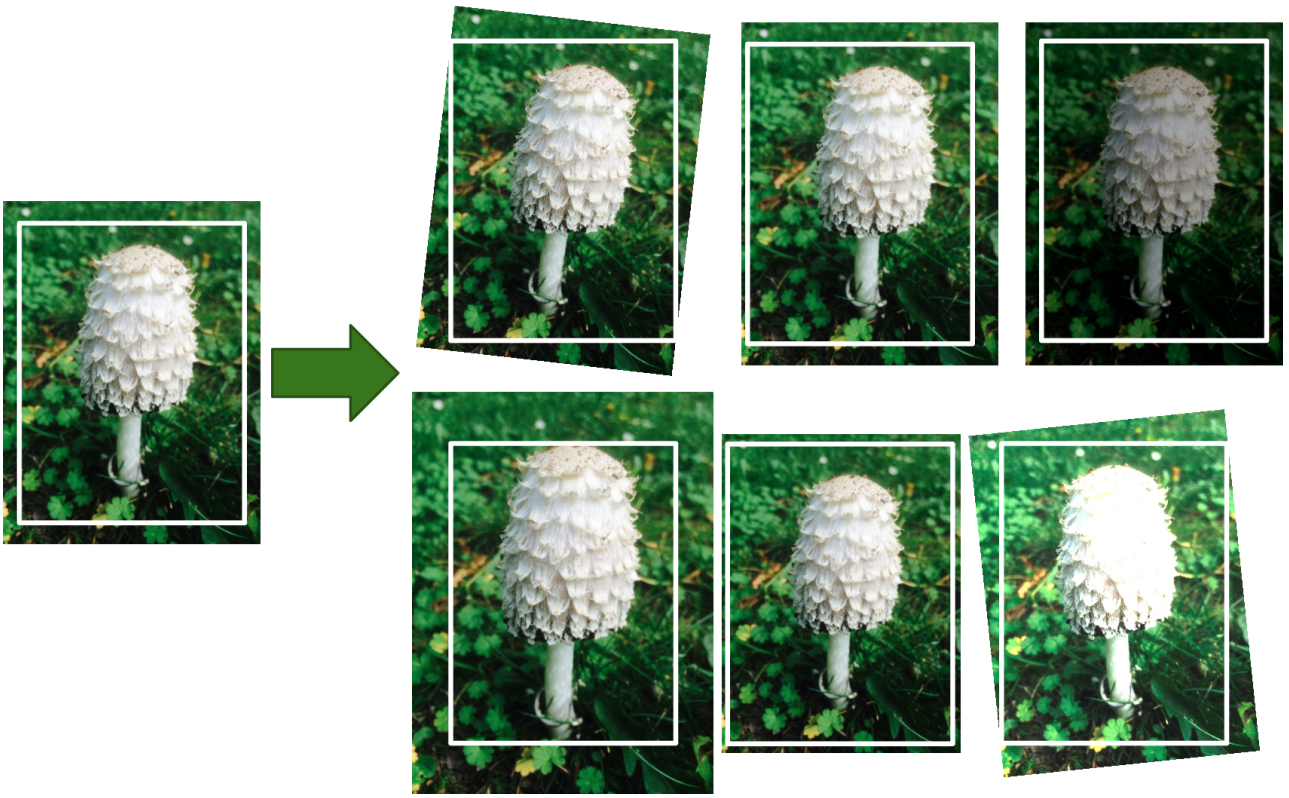


Figure 12-13. Generating new training instances from existing ones

Data augmentation is also useful when you have an unbalanced dataset: you can use it to generate more samples of the less frequent classes. This is called the *synthetic minority oversampling technique*, or SMOTE for short.

Lastly, although data augmentation is typically used only during training, one exception is *test-time augmentation* (TTA): this technique involves augmenting the test data and combining the predictions to boost accuracy. For example, if three augmented versions of an image are classified as a bus, while seven are classified as a truck, then it's probably a truck.

AlexNet also used a regularization technique called *local response normalization* (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps. Such competitive activation has been observed in biological neurons. This encourages different feature maps to specialize, pushing them apart and forcing them to explore a wider range of features, ultimately improving generalization. However, this technique was mostly superseded by simpler and more efficient regularization techniques, especially batch normalization.

A variant of AlexNet called *ZF Net*[10] was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research,[11] and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (as you'll see in Figure 12-15). This was made possible by subnetworks called *inception modules*,[12] which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 12-14 shows the architecture of an inception module. The notation "3 × 3 + 1(S)" means that the layer uses a 3 × 3 kernel, stride 1, and `"same"` padding. The input signal is first fed to four different layers in parallel. All convolutional layers use the ReLU activation function. Note that the top convolutional layers use different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and `"same"` padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concatenation layer* (i.e., it concatenates the multiple feature maps output by each of the upper four convolutional layers). It can be implemented using the `torch.cat()` function, with `dim=1`.

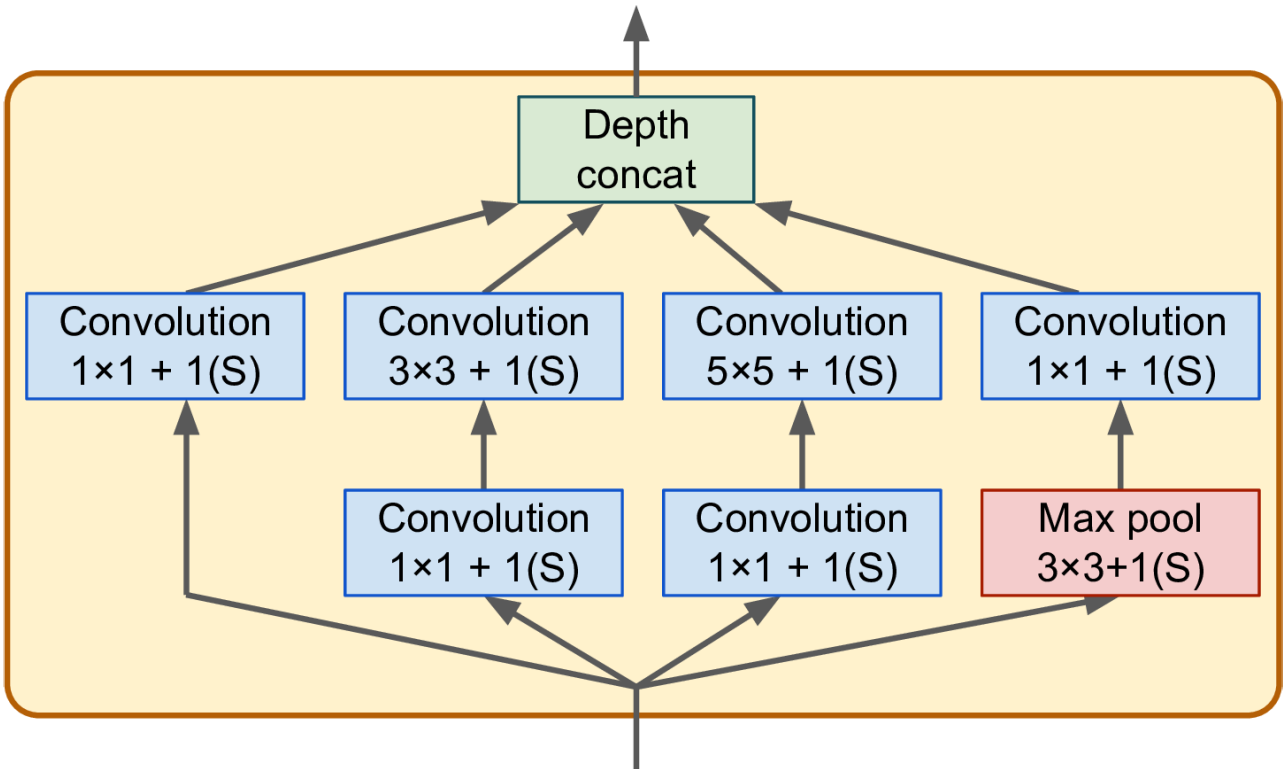


Figure 12-14. Inception module

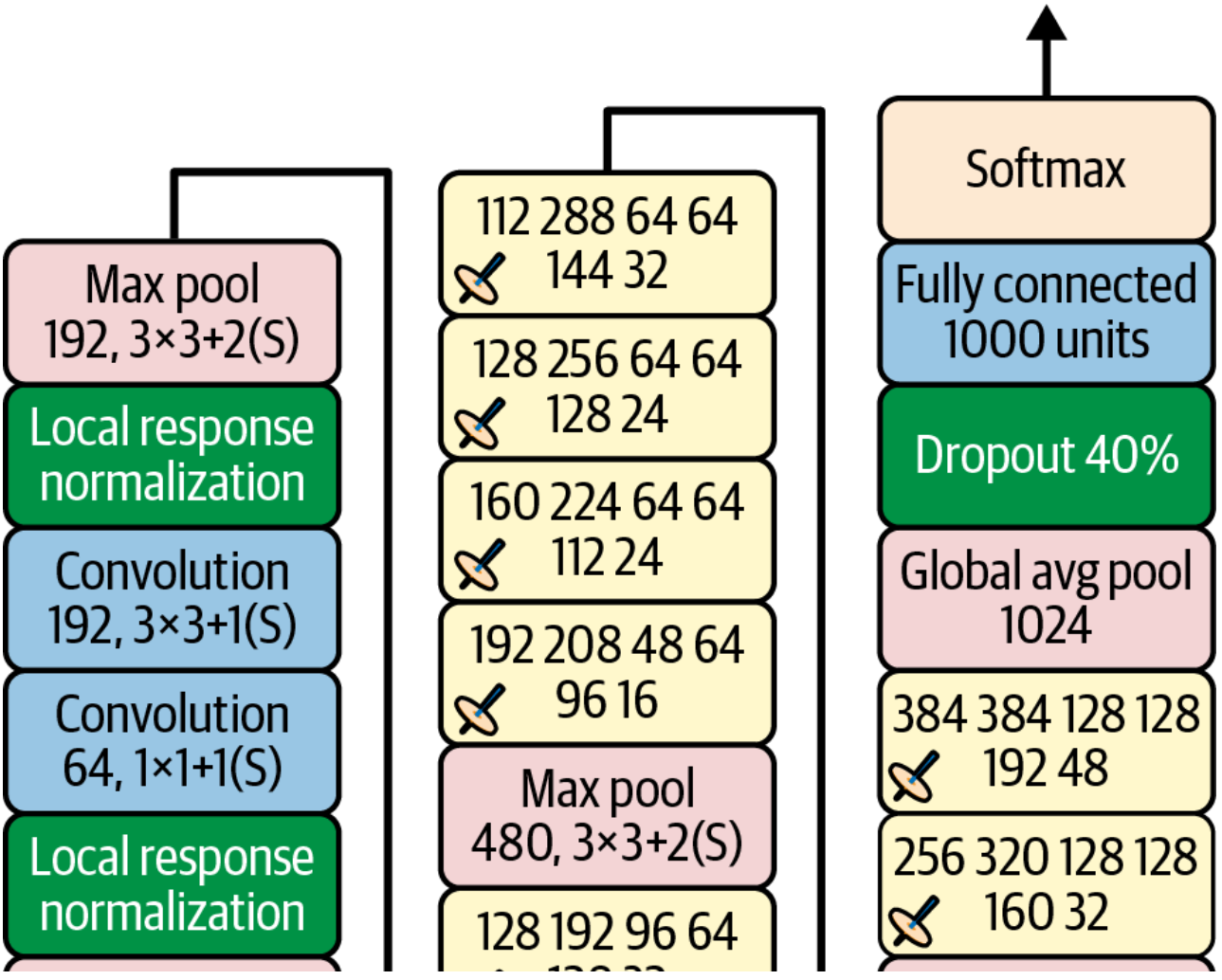
You may wonder why inception modules have convolutional layers with 1 × 1 kernels. Surely

these layers cannot capture any features because they look at only one pixel at a time, right? In fact, these layers serve three purposes:

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the computational cost and the number of parameters, speeding up training and improving generalization.
- Each pair of convolutional layers ([1 × 1, 3 × 3] and [1 × 1, 5 × 5]) acts like a single powerful convolutional layer, capable of capturing more complex patterns. A convolutional layer is equivalent to sweeping a dense layer across the image (at each location, it only looks at a small receptive field), and these pairs of convolutional layers are equivalent to sweeping two-layer neural networks across the image.

In short, you can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.

Now let's look at the architecture of the GoogLeNet CNN (see Figure 12-15). The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in Figure 12-14). Note that all the convolutional layers use the ReLU activation function.



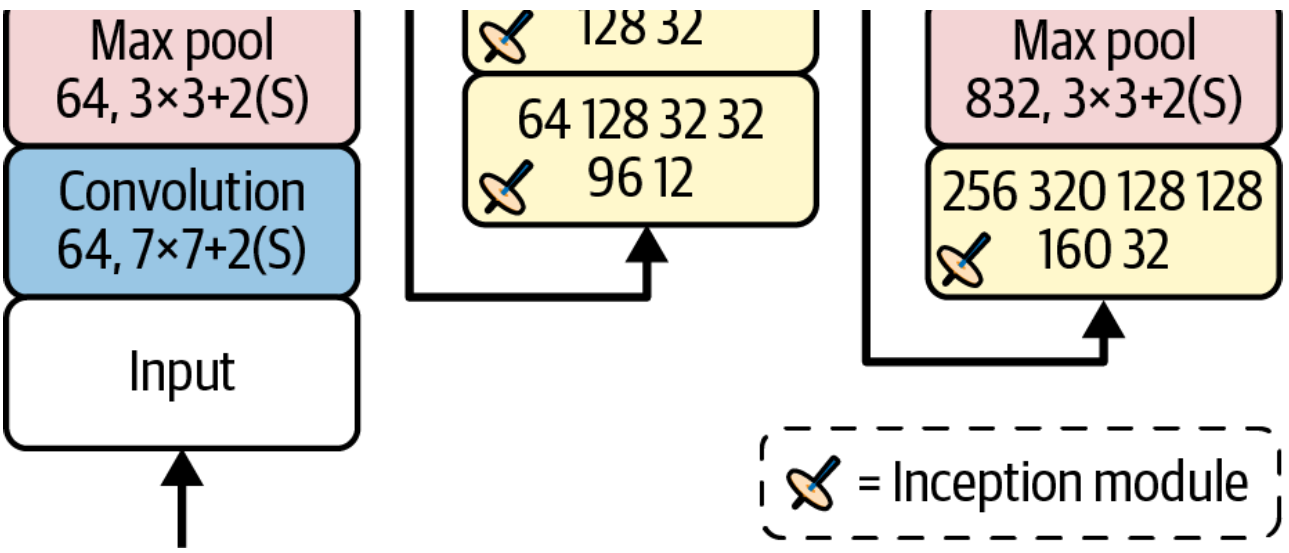


Figure 12-15. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size, 7 × 7, so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a bottleneck layer. As mentioned, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the CNN's *backbone*: a tall stack of nine inception modules, interleaved with a couple of max pooling layers to reduce dimensionality and speed up the net.
- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there is not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224 × 224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7 × 7. Moreover, this is a classification task, not localization, so it doesn't matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

The original GoogLeNet architecture included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed of one average pooling layer, one convolutional layer, two fully connected layers, and a softmax activation layer. During training, their loss (scaled down by 70%) was added to the overall loss. The goal was to fight the

vanishing gradients problem and regularize the network, but it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules to reach even better performance.

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (ResNet)[13] that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers). It confirmed the general trend: computer vision models were getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input $\mathbf{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 12-16).

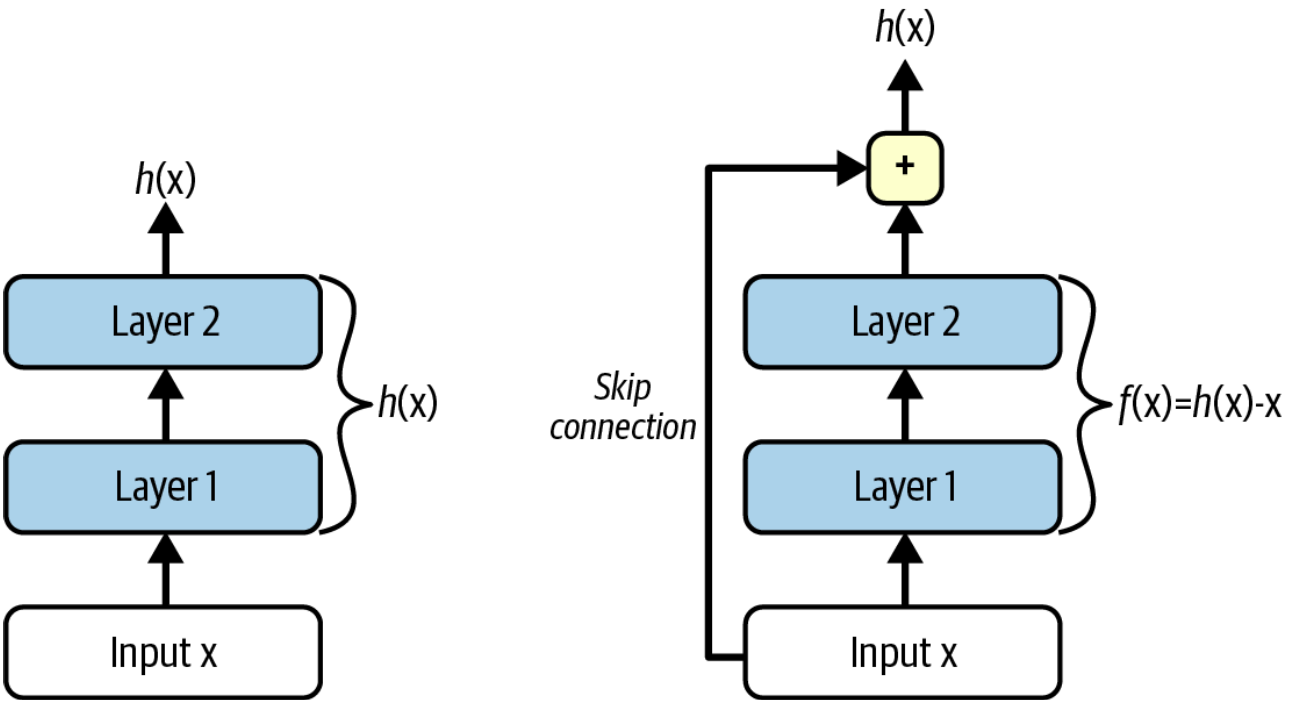


Figure 12-16. Residual learning

When you initialize a neural network, its weights are close to zero, so a regular network just outputs values close to zero when training starts. But if you add a skip connection, the resulting network outputs a copy of its inputs; in other words, it acts as the identity function at the start of training. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Moreover, if you add many skip connections, the network can start making progress even if

several layers have not started learning yet (see Figure 12-17). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units* (RUs), where each residual unit is a small neural network with a skip connection.

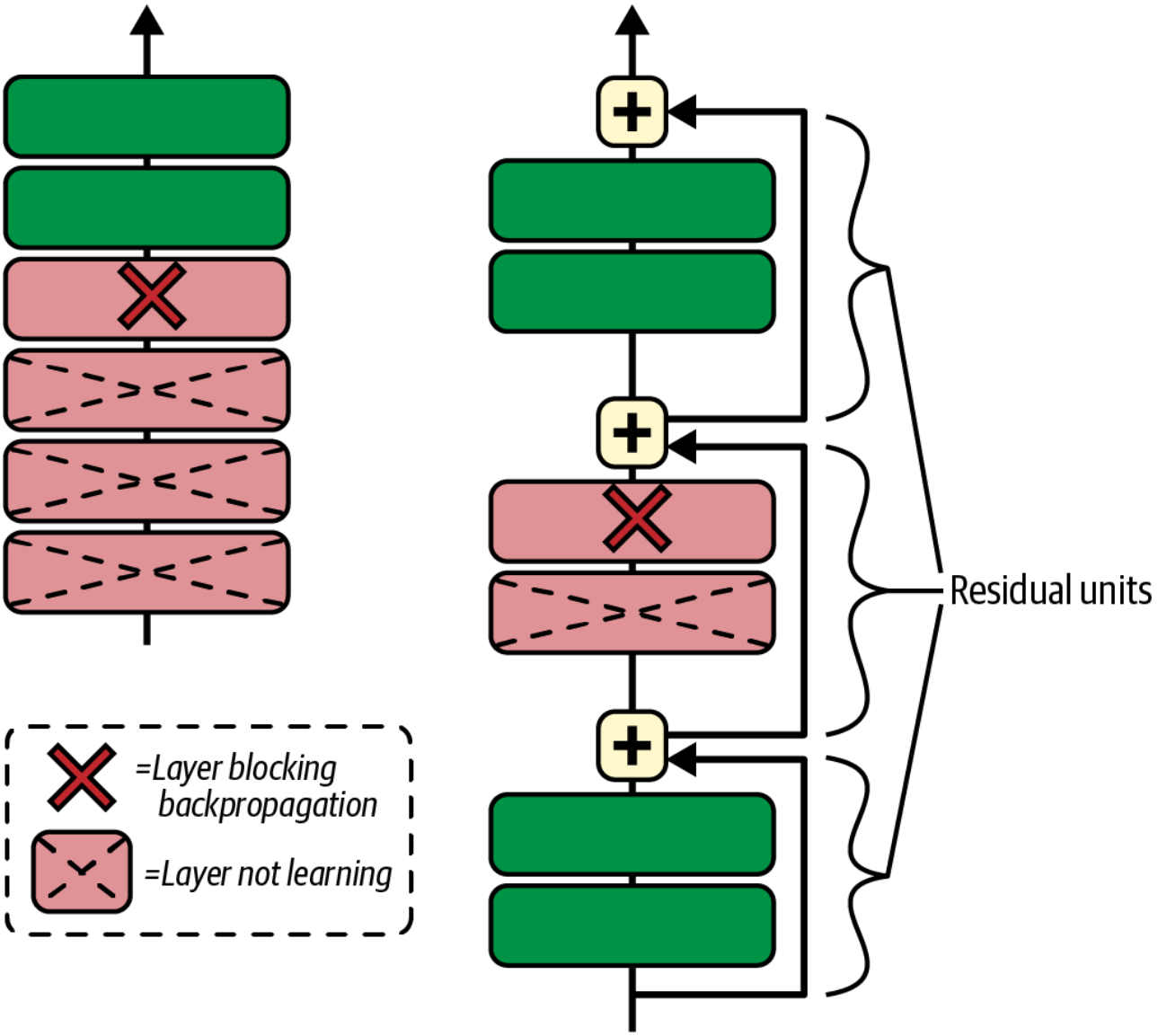
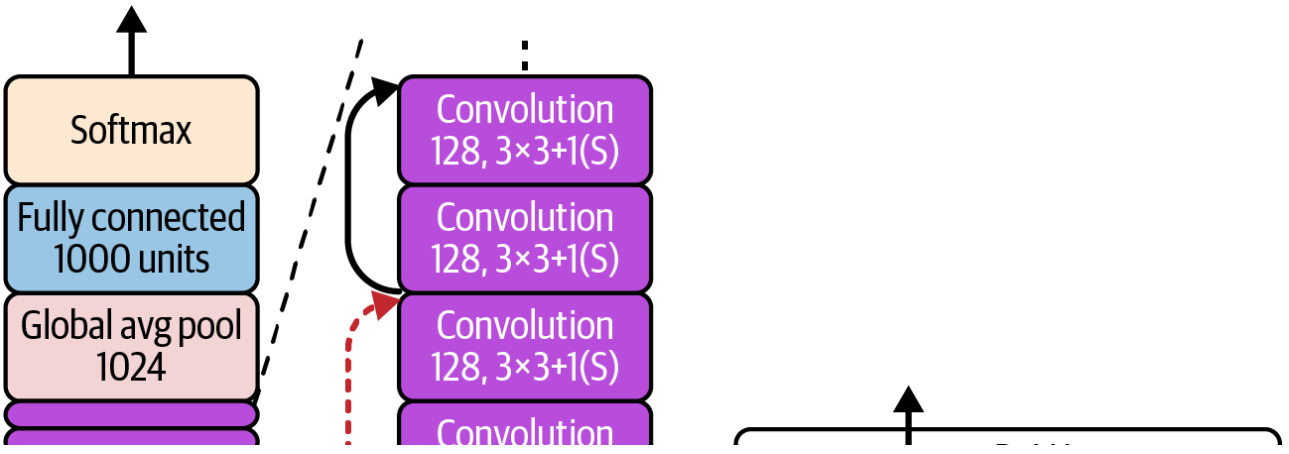


Figure 12-17. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see Figure 12-18). It is surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with batch normalization (BN) and ReLU activation, using 3 × 3 kernels and preserving spatial dimensions (stride 1, `"same"` padding).



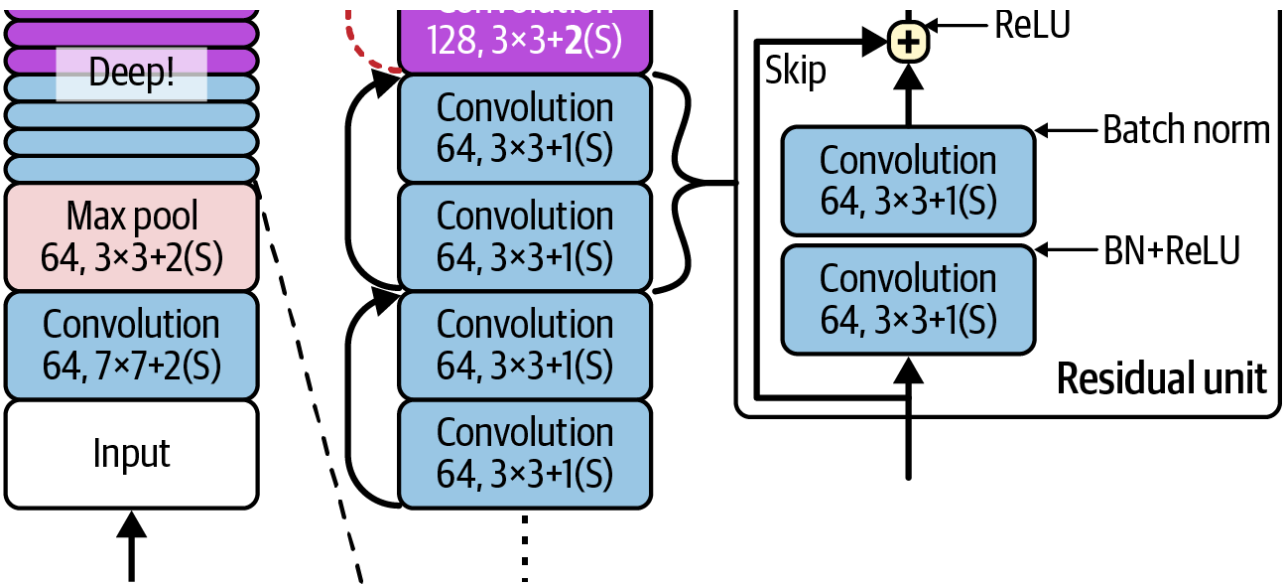


Figure 12-18. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 12-18). To solve this problem, the inputs are passed through a 1 × 1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 12-19).

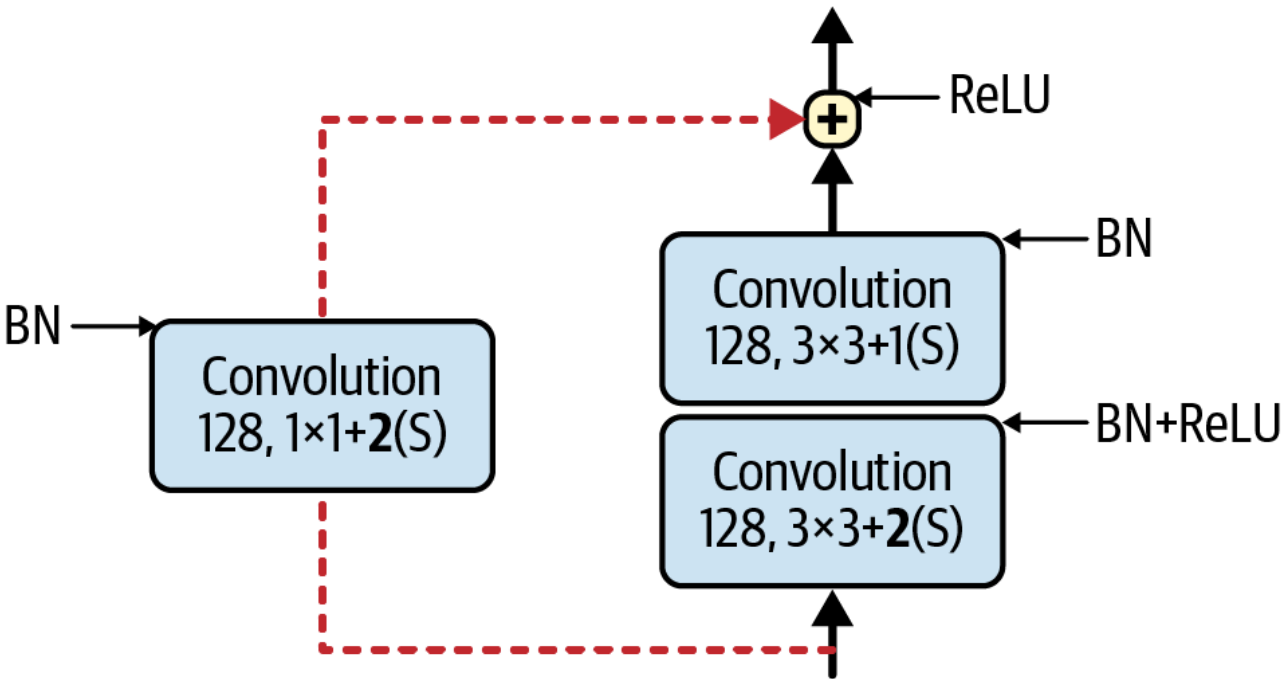


Figure 12-19. Skip connection when changing feature map size and depth

---

TIP

During training, for each mini-batch, you can skip a random set of residual units. This *stochastic depth technique*[14] speeds up training considerably without compromising accuracy. You can implement it using the `torchvision.ops.stochastic_depth()` function.

Different variations of the architecture exist, with different numbers of layers. ResNet-34 is a

ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)[15] containing 3 RUs that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

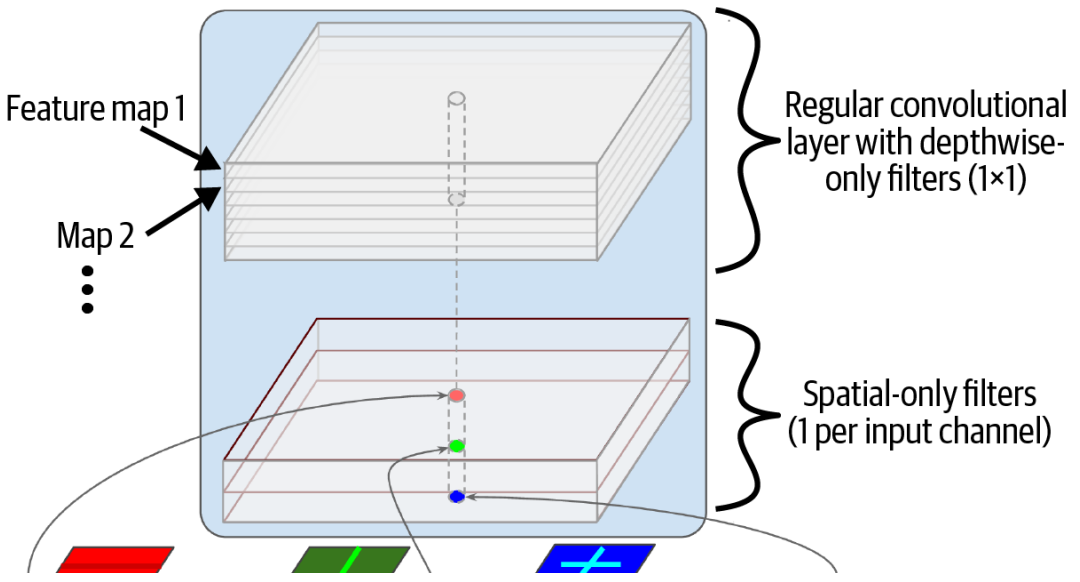
ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3 × 3 convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a 1 × 1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a 3 × 3 layer with 64 feature maps, and finally another 1 × 1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.

NOTE

Google's Inception-v4[16] architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

Xception

Another variant of the GoogLeNet architecture is worth noting: Xception[17] (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of the deep learning framework Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes). Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer* (or *separable convolution layer* for short[18]). These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see Figure 12-20). Thus, it is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part looks exclusively for cross-channel patterns—it is just a regular convolutional layer with 1 × 1 filters.



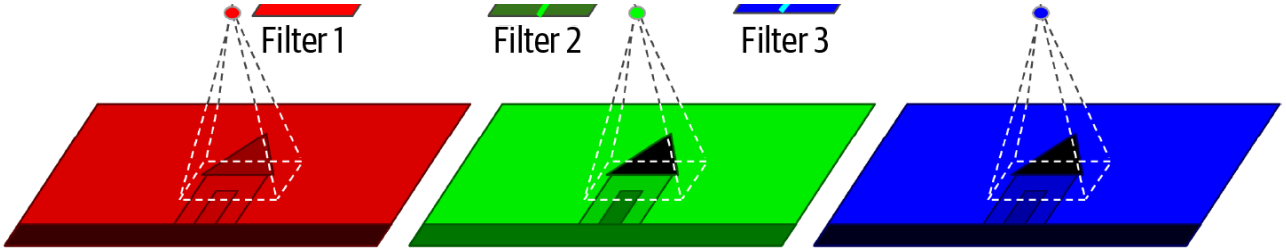


Figure 12-20. Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 12-20 represents, but it is just for illustration purposes). For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

You might wonder why Xception is considered a variant of GoogLeNet, since it contains no inception modules at all. Well, as discussed earlier, an inception module contains convolutional layers with 1 × 1 filters: these look exclusively for cross-channel patterns. However, the convolutional layers that sit on top of them are regular convolutional layers that look both for spatial and cross-channel patterns. So you can think of an inception module as an intermediate between a regular convolutional layer (which considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (which considers them separately). In practice, it seems that separable convolutional layers often perform better.

PyTorch does not include a `SeparableConv2d` module, but it's fairly straightforward to implement your own:

```
class SeparableConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0):
        super().__init__()
        self.depthwise_conv = nn.Conv2d(
            in_channels, in_channels, kernel_size, stride=stride,
            padding=padding, groups=in_channels)
        self.pointwise_conv = nn.Conv2d(
            in_channels, out_channels, kernel_size=1, stride=1, padding=0)

    def forward(self, inputs):
        return self.pointwise_conv(self.depthwise_conv(inputs))
```

Notice the `groups` argument on the seventh line: it lets you split the input channels into the given number of independent groups, each with its own filters (note that `in_channels` and `out_channels` need to be divisible by `groups`). By default `groups=1`, giving you a normal convolutional layer, but if you set both `groups=in_channels` and `out_channels=in_channels`, you get a depthwise convolutional layer, with one filter per input channel. That's the first layer in the separable convolutional layer. The second is a regular convolutional layer, except we set its kernel size and stride to 1. And that's it!

TIP

Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and they often perform better. Consider using them by default, except after layers with few channels (such as the input channel).

SENet

The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network (SENet).[19] This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing 2.25% top-five error rate! The extended versions of inception networks and ResNets are called *SE-Inception* and *SE-ResNet*, respectively. The boost comes from the fact that a SENet adds a small neural network, called an *SE block*, to every inception module or residual unit in the original architecture, as shown in Figure 12-21.

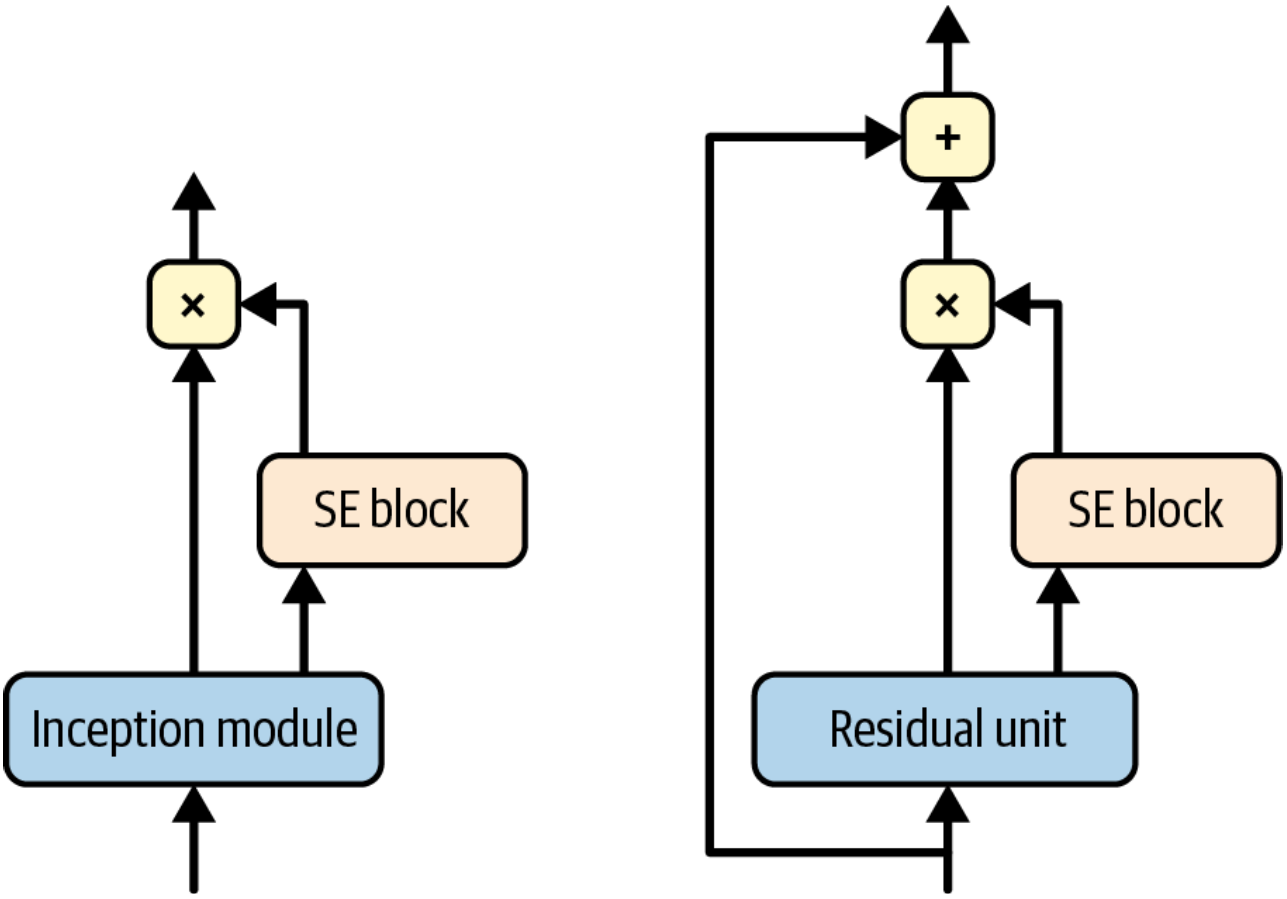


Figure 12-21. SE-Inception module (left) and SE-ResNet unit (right)

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in Figure 12-22. For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So, if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrele-

vant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

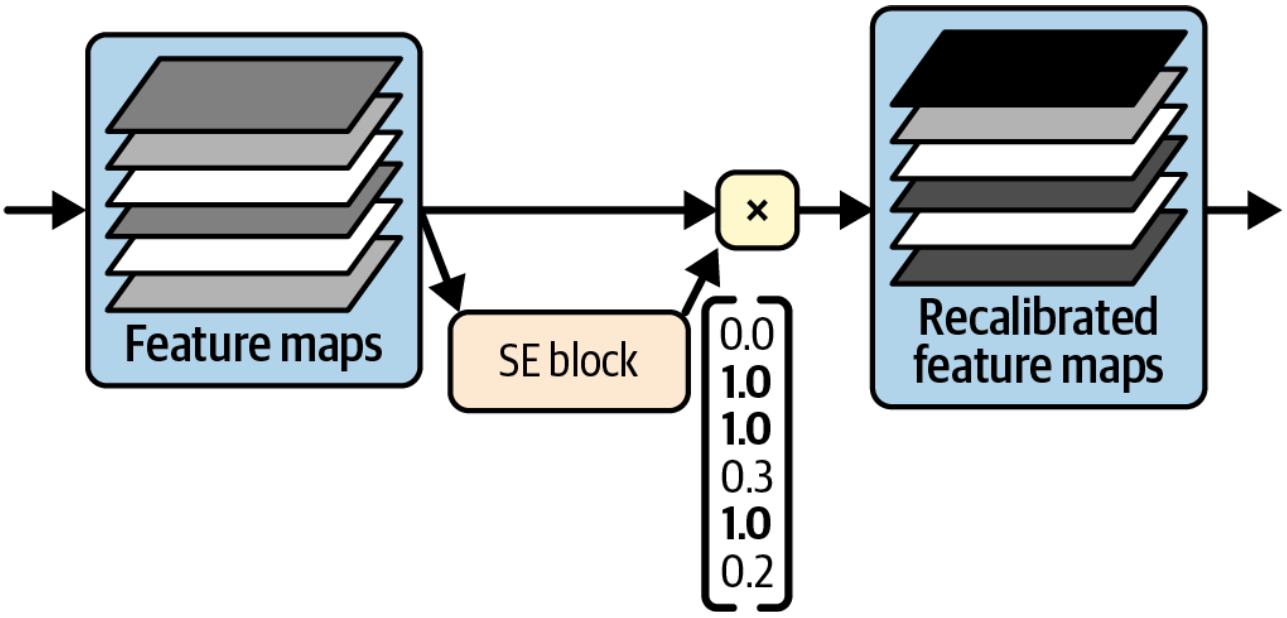


Figure 12-22. An SE block performs feature map recalibration

An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see Figure 12-23).

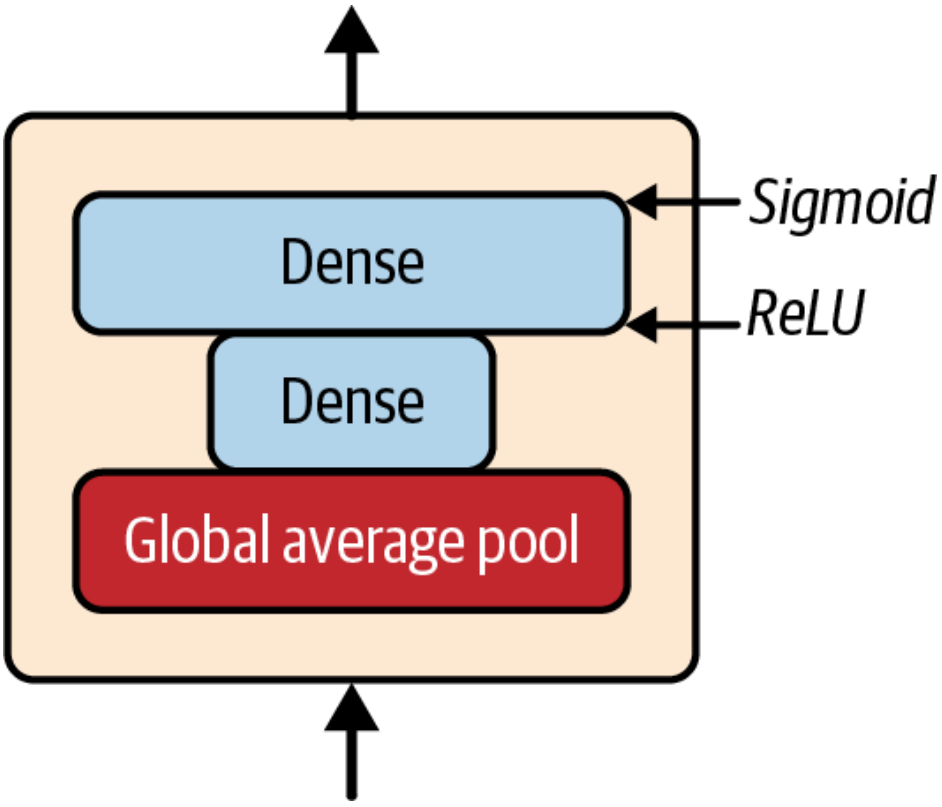


Figure 12-23. SE block architecture

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter. The next layer is where the “squeeze” happens: this layer has significantly fewer than 256 neurons—typically 16 times fewer than the number of feature maps (e.g., 16 neurons)—so the 256 numbers get compressed into a small vector (e.g., 16 dimensions). This is a low-dimensional vector representation (i.e., an embedding) of the dis-

10 dimensions). This is a low-dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in Chapter 18). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

Other Noteworthy Architectures

There are many other CNN architectures to explore. Here's a brief overview of some of the most noteworthy:

VGGNet[20]

VGGNet was the runner-up in the ILSVRC 2014 challenge. Karen Simonyan and Andrew Zisserman, from the Visual Geometry Group (VGG) research lab at Oxford University, developed a very simple and classical architecture; it had 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used small 3 × 3 filters, but it had many of them.

ResNeXt[21]

ResNeXt improves the residual units in ResNet. Whereas the residual units in the best ResNet models just contain 3 convolutional layers each, the ResNeXt residual units are composed of many parallel stacks (e.g., 32 stacks), with 3 convolutional layers each. However, the first two layers in each stack only use a few filters (e.g., just four), so the overall number of parameters remains the same as in ResNet. Then the outputs of all the stacks are added together, and the result is passed to the next residual unit (along with the skip connection).

DenseNet[22]

A DenseNet is composed of several dense blocks, each made up of a few densely connected convolutional layers. This architecture achieved excellent accuracy while using comparatively few parameters. What does "densely connected" mean? The output of each layer is fed as input to every layer after it within the same block. For example, layer four in a block takes as input the depthwise concatenation of the outputs of layers one, two, and three in that block. Dense blocks are separated by a few transition layers.

MobileNet[23]

MobileNets are streamlined models designed to be lightweight and fast, making them popular in mobile and web applications. They are based on depthwise separable convolutional layers, like Xception. The authors proposed several variants, trading a bit of accuracy for faster and smaller models. Several other CNN architectures are available for

mobile devices, such as SqueezeNet, ShuffleNet, or MNasNet.

CSPNet[24]

A Cross Stage Partial Network (CSPNet) is similar to a DenseNet, but part of each dense block's input is concatenated directly to that block's output, without going through the block.

EfficientNet[25]

EfficientNet is arguably the most important model in this list. The authors proposed a method to scale any CNN efficiently by jointly increasing the depth (number of layers), width (number of filters per layer), and resolution (size of the input image) in a principled way. This is called *compound scaling*. They used neural architecture search to find a good architecture for a scaled-down version of ImageNet (with smaller and fewer images), and then used compound scaling to create larger and larger versions of this architecture. When EfficientNet models came out, they vastly outperformed all existing models, across all compute budgets, and they remain among the best models out there today. The authors published a follow-up paper in 2021, introducing EfficientNetV2, which improved training time and parameter efficiency even further.

ConvNeXt[26]

ConvNeXt is quite similar to ResNet, but with a number of tweaks inspired from the most successful vision transformer architectures (see Chapter 16), such as using large kernels (e.g., 7 × 7 instead of 3 × 3), using fewer activation functions and normalization layers in each residual unit, and more.

Understanding EfficientNet's compound scaling method is helpful to gain a deeper understanding of CNNs, especially if you ever need to scale a CNN architecture. It is based on a logarithmic measure of the compute budget, noted ϕ: if your compute budget doubles, then ϕ increases by 1. In other words, the number of floating-point operations available for training is proportional to 2^{ϕ}. Your CNN architecture's depth, width, and resolution should scale as α^{ϕ}, β^{ϕ}, and γ^{ϕ}, respectively. The factors α, β, and γ must be greater than 1, and $\alpha\beta^2\gamma^2$ should be close to 2. The optimal values for these factors depend on the CNN's architecture. To find the optimal values for the EfficientNet architecture, the authors started with a small baseline model (EfficientNetB0), fixed $\phi = 1$, and simply ran a grid search: they found $\alpha = 1.2$, $\beta = 1.1$, and $\gamma = 1.1$. They then used these factors to create several larger architectures, named EfficientNetB1 to EfficientNetB7, for increasing values of ϕ.

I hope you enjoyed this deep dive into the main CNN architectures! But how do you choose the right one?

Choosing the Right CNN Architecture

As you might expect, the best architecture depends on what matters most for your project: Accuracy? Model size (e.g., for deployment to a mobile device)? Inference speed? Energy con-

sumption? Table 12-3 lists some of the pretrained classification models currently available in TorchVision (you'll see how to use them later in this chapter). You can find the full list at *https://pytorch.org/vision/stable/models* (including models for other computer vision tasks). The table shows each model's top-1 and top-5 accuracy on the ImageNet dataset, its number of parameters (in millions), and how much compute it requires for each image (measured in GFLOPs: a Giga-FLOP is one billion floating point operations). As you can see, larger models are generally more accurate, but not always; for example, the small variant of EfficientNet v2 outperforms Inception v3 both in size and accuracy (but not in compute).

Table 12-3. Some of the pretrained models available in TorchVision, sorted by size

Class name	Top-1 acc	Top-5 acc	Params	GFLOPs
MobileNet v3 small	67.7%	87.4%	2.5M	0.1
EfficientNet B0	77.7%	93.5%	5.3M	0.4
GoogLeNet	69.8%	89.5%	6.6M	1.5
DenseNet 121	74.4%	92.0%	8.0M	2.8
EfficientNet v2 small	84.2%	96.9%	21.5M	8.4
ResNet 34	73.3%	91.4%	21.8M	3.7
Inception V3	77.3%	93.5%	27.2M	5.7
ConvNeXt Tiny	82.6%	96.1%	28.6M	4.5
DenseNet 161	77.1%	93.6%	28.7M	7.7
ResNet 152	82.3%	96.0%	60.2M	11.5
AlexNet	56.5%	79.1%	61.1M	0.7
EfficientNet B7	84.1%	96.9%	66.3M	37.8
ResNeXt 101 32x8D	82.8%	96.2%	88.8M	16.4
EfficientNet v2 large	85.8%	97.8%	118.5M	56.1
VGG 11 with BN	70.4%	89.8%	132.9M	7.6
ConvNeXt Large	84.4%	97.0%	197.8M	34.4

The smaller models will run on any GPU, but what about a large model like ConvNeXt Large? Since each parameter is represented as a 32-bit float (4 bytes), you might think you just need 800 MB of RAM to run a 200M parameter model, but you actually need *much* more, typically 5 GB per image at inference time (depending on the image size), and even more at training time. Let's see why.

GPU RAM Requirements: Inference Versus Training

CNNs need a *lot* of RAM. For example, consider a single convolutional layer with 200 5 × 5 fil-

ters, stride 1 and `"same"` padding, processing a 150 × 100 RGB image (3 channels):

- The number of parameters is (5 × 5 × 3 + 1) × 200 = 15,200 (the + 1 corresponds to the bias terms). That's not much: to produce the same size outputs, a fully connected layer would need 200 × 150 × 100 neurons, each connected to all 150 × 100 × 3 inputs. It would have 200 × 150 × 100 × (150 × 100 × 3 + 1) ≈ 135 billion parameters!
- However, each of the 200 feature maps contains 150 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5 × 5 × 3 = 75 inputs: that's a total of 225 million float multiplications. Not as bad as a fully connected layer, but still quite computationally intensive.
- Importantly, the convolutional layer's output will occupy 200 × 150 × 100 × 32 = 96 million bits (12 MB) of RAM, assuming we're using 32-bit floats.[27] And that's just for one instance—if a training batch contains 100 instances, then this single convolutional layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the backward pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers. You can easily run out of GPU RAM.

If training crashes because of an out-of-memory error, you can try reducing the batch size. To still get some of the benefits of large batches, you can accumulate the gradients after each batch, and only update the model weights every few batches. Alternatively, you can try reducing dimensionality using a stride, removing a few layers, using 16-bit floats instead of 32-bit floats, distributing the CNN across multiple devices, or offloading the most memory-hungry modules to the CPU (using `module.to("cpu")`).

Yet another option is to trade off some compute in exchange for a lower memory usage. For example, instead of saving all the activations during the forward pass, you can save some of them, called *checkpoints*, then during the backward pass, you can recompute the missing activations as needed by running a partial forward pass starting from the previous checkpoint. To implement checkpointing in PyTorch, you can use the `torch.utils.checkpoint()` function: instead of calling a module `z = foo(x)`, you can call it using `z = checkpoint(foo, x)`. During inference, it will make no difference, but during training the activations will no longer be saved during the forward pass, and `foo(x)` will be recomputed during the backward pass when needed. This approach is fairly simple to implement, and it doesn't require any changes to your model architecture.

That said, if you're OK with tweaking your model architecture, then there's a much more efficient solution you can use to exchange compute for memory: reversible residual networks.

Reversible Residual Networks (RevNets)

RevNets were proposed by Aidan Gomez et al. in 2017:[28] they typically only increase compute by about 33% and actually don't require you to save any activations at all during the forward

pass! Here's how they work:

- Each layer, called a *reversible layer*, takes two inputs of equal sizes, $\mathbf{x}_1$ and $\mathbf{x}_2$, and computes two outputs: $\mathbf{y}_1 = \mathbf{x}_1 + f(\mathbf{x}_2)$ and $\mathbf{y}_2 = g(\mathbf{y}_1) + \mathbf{x}_2$, where f and g can be any functions, as long as the output size equals the input size, and as long as they always produce the same output for a given input. For example, f and g can be identical modules composed of a few convolutional layers with stride 1 and `"same"` padding (each convolutional layer comes with its own batch-norm and ReLU activation).
- During backpropagation, the inputs of each reversible layer can be recomputed from the outputs whenever needed, using: $\mathbf{x}_2 = \mathbf{y}_2 - g(\mathbf{y}_1)$ and $\mathbf{x}_1 = \mathbf{y}_1 - f(\mathbf{x}_2)$ (you can easily verify that these two equalities follow directly from the first two). No need to store any activations during the forward pass: brilliant!

Since f and g must output the same shape as the input, reversible layers cannot contain convolutional layers with a stride greater than 1, or with `"valid"` padding. You can still use such layers in your CNN, but the RevNet trick won't be applicable to them, so you will have to save their activations during the forward pass; luckily, a CNN usually requires only a handful of such layers. This includes the very first layer, which reduces the spatial dimensions and increases the number of channels: the result can be split in two equal parts along the channels dimension and fed to the first reversible layer.

RevNets aren't limited to CNNs. In fact, they are at the heart of an influential Transformer architecture named Reformer (see Chapter 17).

OK, it's now time to get our hands dirty! Let's implement one of the most popular CNN architectures from scratch using PyTorch.

Implementing a ResNet-34 CNN Using PyTorch

Most CNN architectures described so far can be implemented pretty naturally using PyTorch (although generally you would load a pretrained network instead, as you will see). To illustrate the process, let's implement a ResNet-34 from scratch with PyTorch. First, we'll create a `ResidualUnit` layer:

```
class ResidualUnit(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        DefaultConv2d = partial(
            nn.Conv2d, kernel_size=3, stride=1, padding=1, bias=False)
        self.main_layers = nn.Sequential(
            DefaultConv2d(in_channels, out_channels, stride=stride),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            DefaultConv2d(out_channels, out_channels),
            nn.BatchNorm2d(out_channels),
        )
```

```
        if stride > 1:
            self.skip_connection = nn.Sequential(
                DefaultConv2d(in_channels, out_channels, kernel_size=1,
                              stride=stride, padding=0),
                nn.BatchNorm2d(out_channels),
            )
        else:
            self.skip_connection = nn.Identity()

    def forward(self, inputs):
        return F.relu(self.main_layers(inputs) + self.skip_connection(inputs))
```

As you can see, this code matches Figure 12-19 pretty closely. In the constructor, we create all the layers we need: the main layers are the ones on the righthand side of the figure, and the skip connection corresponds to the layers on the left when the stride is greater than 1, or an `nn.Identity` module when the stride is 1—the `nn.Identity` module does nothing at all, it just returns its inputs. Then in the `forward()` method, we make the inputs go through both the main layers and the skip connection, then we add both outputs and apply the activation function.

Next, let's build our `ResNet34` module! Now that we have our `ResidualUnit` module, the whole ResNet-34 architecture becomes one big stack of modules, so we can base our `ResNet34` class on a single `nn.Sequential` module. The code closely matches Figure 12-18:

```
class ResNet34(nn.Module):
    def __init__(self):
        super().__init__()
        layers = [
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2,
                      padding=3, bias=False),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        ]
        prev_filters = 64
        for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
            stride = 1 if filters == prev_filters else 2
            layers.append(ResidualUnit(prev_filters, filters, stride=stride))
            prev_filters = filters
        layers += [
            nn.AdaptiveAvgPool2d(output_size=1),
            nn.Flatten(),
            nn.LazyLinear(10),
        ]
        self.resnet = nn.Sequential(*layers)

    def forward(self, inputs):
        return self.resnet(inputs)
```

The only tricky part in this code is the loop that adds the `ResidualUnit` layers to the list of layers: as explained earlier, the first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on. At each iteration, we must set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2; then we append the `ResidualUnit` to the list, and finally we update `prev_filters`.

And that's it, you could now train this model on ImageNet or any other dataset of 224 × 224 images. It is amazing that in just 45 lines of code, we can build the model that won the ILSVRC 2015 challenge! This demonstrates both the elegance of the ResNet model and the expressiveness of PyTorch (and Python). Implementing the other CNN architectures we discussed would take more time, but it wouldn't be much harder. However, TorchVision comes with several of these architectures built in, so why not use them instead?

Using TorchVision's Pretrained Models

In general, you won't have to implement standard models like GoogLeNet, ResNet, or ConvNeXt manually, since pretrained networks are readily available with a couple lines of code using TorchVision.

TIP

TIMM is another very popular library built on PyTorch: it provides a collection of pretrained image classification models, as well as many related tools such as data loaders, data augmentation utilities, optimizers, schedulers, and more. Hugging Face's Hub is also a great place to get all sorts of pretrained models (see Chapter 14).

For example, you can load a ConvNeXt model pretrained on ImageNet with the following code. There are several variants of the ConvNeXt model—tiny, small, base, and large—and this code loads the base variant:

```
weights = torchvision.models.ConvNeXt_Base_Weights.IMAGENET1K_V1
model = torchvision.models.convnext_base(weights=weights).to(device)
```

That's all! This code automatically downloads the weights (338 MB) from the *Torch Hub*, an online repository of pretrained models. The weights are saved and cached for future use (e.g., in `~/.cache/torch/hub`; run `torch.hub.get_dir()` to find the exact path on your system). Some models have newer weights versions (e.g., IMAGENET1K_V2) or other weight variants. For the full list of available models, run `torchvision.models.list_models()`. To find the list of pretrained weights available for a given model, such as `convnext_base`, run `list(torchvision.models.get_model_weights("convnext_base"))`. Alternatively, visit *https://pytorch.org/vision/main/models*.

Let's use this model to classify the two sample images we loaded earlier. Before we can do this,

we must first ensure that the images are preprocessed exactly as the model expects. In particular, they must have the right size. A ConvNeXt model expects 224 × 224 pixel images (other models may expect other sizes, such as 299 × 299). Since our sample images are 427 × 640 pixels, we need to resize them. We could do this using TorchVision's `CenterCrop` and/or `Resize` transform, but it's much easier and safer to use the transforms returned by `weights.transforms()`, as they are specifically designed for this particular pretrained model:

```
transforms = weights.transforms()
preprocessed_images = transforms(sample_images_permuted)
```

Importantly, these transforms also normalize the pixel intensities just like during training. In this case, the transforms standardize the pixel intensities separately for each color channel, using ImageNet's means and standard deviations for each channel (we will see how to do this manually later in this chapter).

Next we can move the images to the GPU and pass them to the model. As always, remember to switch the model to evaluation mode before making predictions—the model is in training mode by default—and also turn off autograd:

```
model.eval()
with torch.no_grad():
    y_logits = model(preprocessed_images.to(device))
```

The result is a 2 × 1,000 tensor containing the class logits for each image (recall that ImageNet has 1,000 classes). As we did in Chapter 10, we can use `torch.argmax()` to get the predicted class for each image (i.e., the class with the maximum logit):

```
>>> y_pred = torch.argmax(y_logits, dim=1)
>>> y_pred
tensor([698, 985], device='cuda:0')
```

So far, so good, but what exactly do these classes represent? Well you could find the ImageNet class names online, but once again it's simpler and safer to get the class names directly from the `weights` object. Indeed, its `meta` attribute is a dictionary containing metadata about the pretrained model, including the class names:

```
>>> class_names = weights.meta["categories"]
>>> [class_names[class_id] for class_id in y_pred]
['palace', 'daisy']
```

There you have it: the first image is classified as a palace, and the second as a daisy. Since the ImageNet dataset does not have classes for Chinese towers or dahlia flowers, a palace and a daisy are reasonable substitutes (the tower is part of the Summer Palace in Beijing). Let's look

at the top-three predictions using `topk()`:

```
>>> y_top3_logits, y_top3_class_ids = y_logits.topk(k=3, dim=1)
>>> [[class_names[class_id] for class_id in top3] for top3 in y_top3_class_ids]
[['palace', 'monastery', 'lakeside'], ['daisy', 'pot', 'ant']]
```

Let's look at the estimated probabilities for each of these classes:

```
>>> y_top3_logits.softmax(dim=1)
tensor([[0.8618, 0.1185, 0.0197],
        [0.8106, 0.0964, 0.0930]], device='cuda:0')
```

As you can see, TorchVision makes it easy to download and use pretrained models, and it works quite well out of the box for ImageNet classes. But what if you need to classify images into classes that don't belong to the ImageNet dataset, such as various flower species? In that case, you may still benefit from the pretrained models by using them to perform transfer learning.

Pretrained Models for Transfer Learning

If you want to build an image classifier but you do not have enough data to train it from scratch, then it is often a good idea to reuse the lower layers of a pretrained model, as we discussed in Chapter 11. In this section we will reuse the ConvNeXt model we loaded earlier—which was pretrained on ImageNet—and after replacing its classification head, we will fine-tune it on the *102 Category Flower Dataset*[29] (Flowers102 for short). This dataset only contains 10 images per class, and there are 102 classes in total (as the name indicates), so if you try to train a model from scratch, you will really struggle to get high accuracy. However, it's quite easy to get over 90% accuracy using a good pretrained model. Let's see how. First, let's download the dataset using Torchvision:

```
DefaultFlowers102 = partial(torchvision.datasets.Flowers102, root="datasets",
                            transform=weights.transforms(), download=True)
train_set = DefaultFlowers102(split="train")
valid_set = DefaultFlowers102(split="val")
test_set = DefaultFlowers102(split="test")
```

This code uses `partial()` to avoid repeating the same arguments three times. We also set `transform=weights.transforms()` to preprocess the images immediately when they are loaded. The Flowers102 dataset comes with three predefined splits, for training, validation, and testing. The first two have 10 images per class, but surprisingly the test set has many more (it has a variable number of images per class, between 20 and 238). In a real project, you would normally use most of your data for training rather than for testing, but this dataset was designed for computer vision research, and the authors purposely restricted the training set and

the validation set.

We then create the data loaders, as usual:

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=32)
test_loader = DataLoader(test_set, batch_size=32)
```

Many TorchVision datasets conveniently contain the class names in the `classes` attribute, but sadly not this dataset.[30] If you prefer to see lovely names like "tiger lily", "monkshood", or "snapdragon" rather than boring class IDs, then you need to manually define the list of class names:

```
class_names = ['pink primrose', ..., 'trumpet creeper', 'blackberry lily']
```

Now let's adapt our pretrained ConvNeXt-base model to this dataset. Since it was pretrained on ImageNet, which has 1,000 classes, the model's head (i.e., its upper layers) was designed to output 1,000 logits. But we only have 102 classes, so we must chop the model's head off and replace it with a smaller one. But how can we find it? Well let's use the model's `named_children()` method to find the name of its submodules:

```
>>> [name for name, child in model.named_children()]
['features', 'avgpool', 'classifier']
```

The `features` module is the main part of the model, which includes all layers except for the global average pooling layer (`avgpool`) and the model's head (`classifier`). Let's look more closely at the head:

```
>>> model.classifier
Sequential(
  (0): LayerNorm2d((1024,), eps=1e-06, elementwise_affine=True)
  (1): Flatten(start_dim=1, end_dim=-1)
  (2): Linear(in_features=1024, out_features=1000, bias=True)
)
```

As you can see, it's a `nn.Sequential` module composed of a layer normalization layer, a `nn.Flatten` layer, and a `nn.Linear` layer with 1,024 inputs and 1,000 outputs. This `nn.Linear` layer is the output layer, and it's the one we need to replace. We must only change the number of outputs:

```
n_classes = 102  # len(class_names) == 102
model.classifier[2] = nn.Linear(1024, n_classes).to(device)
```

```
model.classifier[2] = nn.Linear(1024, n_classes).to(device)
```

As explained in Chapter 11, it's usually a good idea to freeze the weights of the pretrained layers, at least at the beginning of training. We can do this by freezing every single parameter in the model, and then unfreezing only the parameters of the head:

```
for param in model.parameters():
    param.requires_grad = False

for param in model.classifier.parameters():
    param.requires_grad = True
```

Next, you can train this model for a few epochs, and you will already reach about 90% accuracy just by training the new head, without even fine-tuning the pretrained layers. After that, you can unfreeze the whole model, lower the learning rate—typically by a factor of 10—and continue training the model. Give this a try, and see what accuracy you can reach!

To reach an even higher accuracy, it's usually a good idea to perform some data augmentation on the training images. For this, you can try randomly flipping the training images horizontally, randomly rotating them by a small angle, randomly resizing and cropping them, and randomly tweaking their colors. This must all be done before running the ImageNet normalization step, which you can implement using a `Normalize` transform:

```
import torchvision.transforms.v2 as T

transforms = T.Compose([
    T.RandomHorizontalFlip(p=0.5),
    T.RandomRotation(degrees=30),
    T.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0)),
    T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    T.ToImage(),
    T.ToDtype(torch.float32, scale=True),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

TIP

TorchVision comes with an `AutoAugment` transform which applies multiple augmentation operations optimized for ImageNet. It generalizes well to many other image datasets, and it also offers predefined settings for two other datasets: CIFAR10 and the street view house numbers (SVHN) dataset.

Here are some more ideas to continue to improve your model's accuracy:

- Try other pretrained models.
- Extend the training set: find more flower images and label them.
- Create an ensemble of models, and combine their predictions.

- Create an ensemble of models, and combine their predictions.
- Analyze failure cases, and see whether they share specific characteristics, such as similar texture or color. You can then try to tweak image preprocessing to address these issues.
- Use a learning schedule such as performance scheduling.
- Unfreeze the layers gradually, starting from the top. Alternatively, you can use *differential learning rates*: apply a smaller learning rate to lower layers, and a larger learning rate to upper layers. You can do this by using parameter groups (see Chapter 11).
- Explore different optimizers and fine-tune their hyperparameters.
- Try different regularization techniques.

TIP

It's worth spending time looking for models that were pretrained on similar images. For example, if you're dealing with satellite images, aerial images, or even raster data such as digital elevation models (DEM), then models pretrained on ImageNet won't help much. Instead, check out Microsoft's *TorchGeo* library, which is similar to TorchVision but for geospatial data. For medical images, check out Project MONAI. For agricultural images, check out AgML. And so on.

With that, you can start training amazing image classifiers on your own images and classes! But there's more to computer vision than just classification. For example, what if you also want to know *where* the flower is in a picture? Let's look at this now.

Classification and Localization

Localizing an object in a picture can be expressed as a regression task, as discussed in Chapter 9: to predict a bounding box around the object, a common approach is to predict the location of the bounding box's center, as well as its width and height (alternatively, you could predict the horizontal and vertical coordinates of the object's upper left and lower right corners). This means we have four numbers to predict. It does not require much change to the ConvNeXt model; we just need to add a second dense output layer with four units (e.g., on top of the global average pooling layer). Here's a `FlowerLocator` model that adds a localization head to a given base model, such as our ConvNeXt model:

```
class FlowerLocator(nn.Module):
    def __init__(self, base_model):
        super().__init__()
        self.base_model = base_model
        self.localization_head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(base_model.classifier[2].in_features, 4)
        )

    def forward(self, X):
        features = self.base_model.features(X)
        pool = self.base_model.avgpool(features)
        logits = self.base_model.classifier(pool)
```

```
        bbox = self.localization_head(pool)
        return logits, bbox

torch.manual_seed(42)
locator_model = FlowerLocator(model).to(device)
```

This locator model has two heads: the first outputs class logits, while the second outputs the bounding box. The localization head has the same number of inputs as the `nn.Linear` layer of the classification head, but it outputs just four numbers. The `forward()` method takes a batch of preprocessed images as input and outputs both the predicted class logits (102 per image) and the predicted bounding boxes (1 per image). After training this model, you can use it as follows:

```
preproc_images = [...]  # a batch of preprocessed images
y_pred_logits, y_pred_bbox = locator_model(preprocessed_images.to(device))
```

But how can we train this model? Well, we saw how to train a model with two or more outputs in Chapter 10, and this one is no different: in this case, we can use the `nn.CrossEntropyLoss` for the classification head, and the `nn.MSELoss` for the localization head. The final loss can just be a weighted sum of the two. Voilà, that's all there is to it.

Hey, not so fast! We have a problem: the Flowers102 dataset does not include any bounding boxes, so we need to add them ourselves. This is often one of the hardest and most costly parts of a machine learning project: labeling and annotating the data. It's a good idea to spend time looking for the right tools. To annotate images with bounding boxes, you may want to use an open source labeling tool like Label Studio, OpenLabeler, ImgLab, Labelme, VoTT, or VGG Image Annotator, or perhaps a commercial tool like LabelBox, Supervisely, Roboflow, or RectLabel. Many of these are now AI assisted, greatly speeding up the annotation task. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if you have a very large number of images to annotate. However, it is quite a lot of work to set up a crowdsourcing platform, prepare the form to be sent to the workers, supervise them, and ensure that the quality of the bounding boxes they produce is good, so make sure it is worth the effort. If there are just a few hundred or a even a couple of thousand images to label, and you don't plan to do this frequently, it may be preferable to do it yourself: with the right tools, it will only take a few days, and you'll also gain a better understanding of your dataset and task.

You can then create a custom dataset (see Chapter 10) where each entry contains an image, a label, and a bounding box. TorchVision conveniently includes a `BoundingBoxes` class that represents a list of bounding boxes. For example, the following code creates a bounding box for the largest flower in the first image of the Flowers102 training set (for now we only consider one bounding box per image, but we'll discuss multiple bounding boxes per image later in this chapter):

```
import torchvision.tv_tensors
```

```
bbox = torchvision.tv_tensors.BoundingBoxes(
    [[377, 199, 248, 262]],  # center x=377, center y=199, width=248, height=262
    format="CXCYWH",  # other possible formats: "XYXY" and "XYWH"
    canvas_size=(500, 754)  # raw image size before preprocessing
)
```

TIP

To visualize bounding boxes, you can use the `torchvision.utils.draw_bounding_boxes()` function. You will first need to convert the bounding boxes to the XYXY format using `torchvision.ops.box_convert()`.

The `BoundingBoxes` class is a subclass of `TVTensor`, which is a subclass of `torch.Tensor`, so you can treat bounding boxes exactly like regular tensors, with extra features. Most importantly, you can transform bounding boxes using TorchVision's transforms API v2. For example, let's use the transform we defined earlier to preprocess this bounding box:

```
>>> transform(bbox)
BoundingBoxes([[102,  90, 109, 147]],
              format=BoundingBoxFormat.CXCYWH, canvas_size=(224, 224))
```

WARNING

Resizing and cropping a bounding box works as expected, but rotation is special: the bounding box can't be rotated since it doesn't have any rotation parameter, so instead it is resized to fit the rotated box (*not* the rotated object). As a result, it may end up being a bit too large for the object.

You can pass a nested data structure to a transform and the output will have the same structure, except with all the images and bounding boxes transformed. For example, the following code transforms the first flower image in the training set and its bounding box, leaving the label unchanged. In this example, the input and output are both 2-tuples containing an image and a dictionary composed of a label and a bounding box, but you could use any other data structure:

```
first_image = [...]  # load the first training image without any preprocessing
preproc_image, preproc_target = transform(
    (first_image, {"label": 0, "bbox": bbox})
)
preproc_bbox = preproc_target["bbox"]
```

TIP

When using the MSE, a 10-pixel error for a large bounding box will be penalized just as much as a 10-pixel error for a small bounding box. To avoid this, you can use a custom loss function that computes the square root of the width and height—for both the target and the prediction—before computing the MSE.

square root of the width and height—for both the target and the prediction—before computing the MSE.

The MSE is simple and often works fairly well to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes. The most common metric for this is the *intersection over union* (IoU, also known as the *Jaccard index*): it is the area of overlap between the target bounding box T and the predicted bounding box P, divided by the area of their union P ∪ T (see Figure 12-24). In short, IoU = |P ∩ T| / |P ∪ T|, where |*x*| is the area of *x*. The IoU ranges from 0 (no overlap) to 1 (perfect overlap). It is implemented by the `torchvision.ops.box_iou()` function.

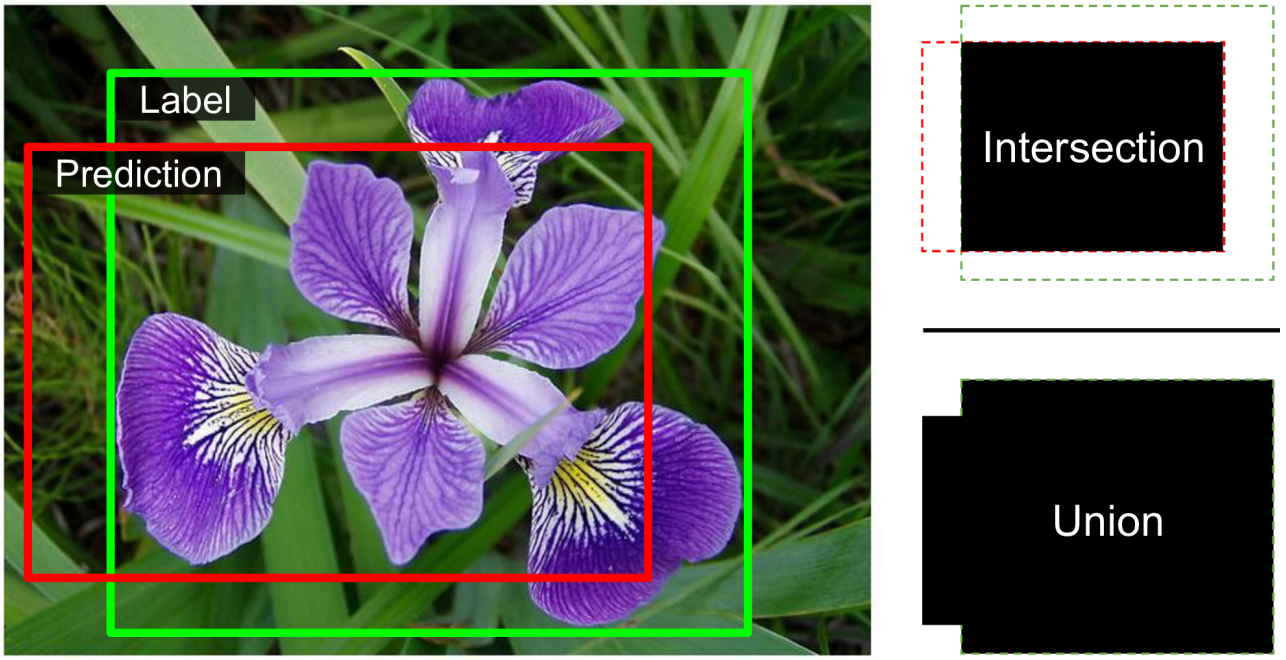


Figure 12-24. IoU metric for bounding boxes

The IoU is not great for training because it is equal to zero whenever P and T have no overlap, regardless of the distance between them or their shapes: in this case the gradient is also equal to zero and therefore gradient descent cannot make any progress. Luckily, it's possible to fix this flaw by incorporating extra information. For example, the *Generalized IoU* (GIoU), introduced in a 2019 paper by H. Rezatofighi et al.,[31] considers the smallest box S that contains both P and T, and it subtracts from the IoU the ratio of S that is not covered by P or T. In short, GIoU = IoU – |S – (P ∪ T)| / |S|. This means that the GIoU gets smaller as P and T get further apart, which gives gradient descent something to play with so it can pull P closer to T. Since we want to maximize the GIoU, the GIoU loss is equal to 1 – GIoU. This loss quickly became popular, and it is implemented by the `torchvision.ops.generalized_box_iou_loss()` function.

Another important variant of the IoU is the *Complete IoU* (CIoU), introduced in a 2020 paper by Z. Zheng et al.[32] It considers three geometric factors: the IoU (the more overlap, the better), the distance between the centers of P and T (the closer, the better), normalized by the length of the diagonal of S, and the similarity between the aspect ratios of P and T (the closer, the better). The loss is 1 – CIoU, and it is implemented by the `torchvision.ops.complete_box_iou_loss()` function. It generally performs better than the MSE or the GIoU, converging faster and leading to more accurate bounding boxes, so it is becoming the default loss for localization.

Classifying and localizing a single object is nice, but what if the images contain multiple objects (as is often the case in the flowers dataset)?

Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object roughly centered in the image, then slide this CNN across the image and make predictions at each step. The CNN was generally trained to predict not only class probabilities and a bounding box, but also an *objectness score*: this is the estimated probability that the image does indeed contain an object centered near the middle. This is a binary classification output; it can be produced by a dense output layer with a single unit, using the sigmoid activation function and trained using the binary cross-entropy loss.

NOTE

Instead of an objectness score, a "no-object" class was sometimes added, but in general this did not work as well. The questions "Is an object present?" and "What type of object is it?" are best answered separately.

This sliding-CNN approach is illustrated in Figure 12-25. In this example, the image was chopped into a 5 × 7 grid, and we see a CNN—the thick black rectangle—sliding across all 3 × 3 regions and making predictions at each step.

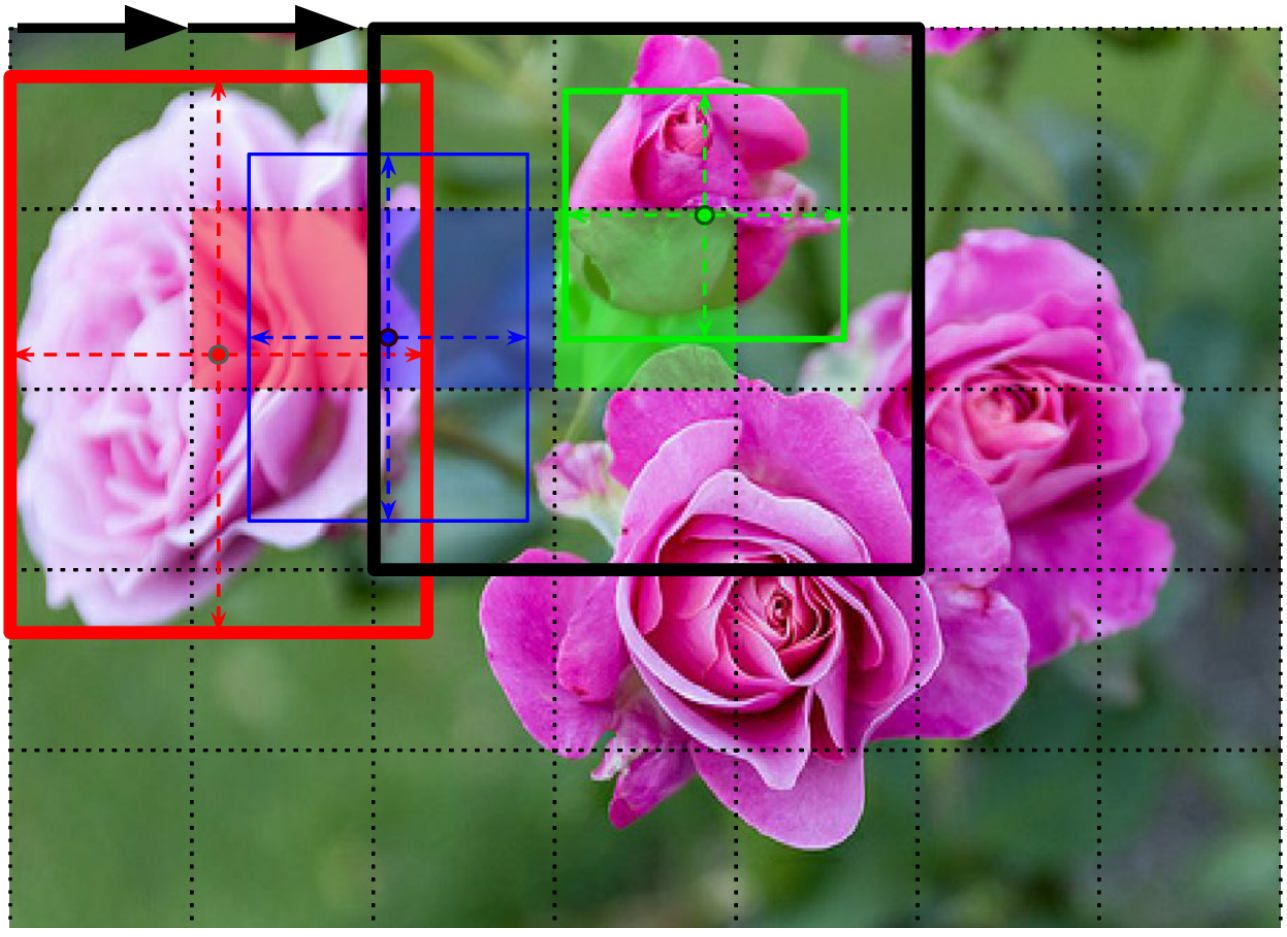


Figure 12-25. Detecting multiple objects by sliding a CNN across the image

In this figure, the CNN has already made predictions for three of these 3 × 3 regions:

- When looking at the top-left 3 × 3 region (centered on the red-shaded grid cell located in the second row and second column), it detected the leftmost rose. Notice that the predicted bounding box exceeds the boundary of this 3 × 3 region. That's absolutely fine: even though the CNN could not see the bottom part of the rose, it was able to make a reasonable guess as to where it might be. It also predicted class probabilities, giving a high probability to the "rose" class. Lastly, it predicted a fairly high objectness score, since the center of the bounding box lies within the central grid cell (in this figure, the objectness score is represented by the thickness of the bounding box).
- When looking at the next 3 × 3 region, one grid cell to the right (centered on the shaded blue square), it did not detect any flower centered in that region, so it predicted a very low objectness score; therefore, the predicted bounding box and class probabilities can safely be ignored. You can see that the predicted bounding box was no good anyway.
- Finally, when looking at the next 3 × 3 region, again one grid cell to the right (centered on the shaded green cell), it detected the rose at the top, although not perfectly. This rose is not well centered within this region, so the predicted objectness score was not very high.

You can imagine how sliding the CNN across the whole image would give you a total of 15 predicted bounding boxes, organized in a 3 × 5 grid, with each bounding box accompanied by its estimated class probabilities and objectness score. Since objects can have varying sizes, you may then want to slide the CNN again across 2 × 2 and 4 × 4 regions as well, to capture smaller and larger objects.

This technique is fairly straightforward, but as you can see it will often detect the same object multiple times, at slightly different positions. Some post-processing is needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression* (NMS). Here's how it works:

1. First, get rid of all the bounding boxes for which the objectness score is below some threshold, since the CNN believes there's no object at that location, the bounding box is useless.
2. Find the remaining bounding box with the highest objectness score, and get rid of all the other remaining bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in Figure 12-25, the bounding box with the max objectness score is the thick bounding box over the leftmost rose. The other bounding box that touches this same rose overlaps a lot with the max bounding box, so we will get rid of it (although in this example it would already have been removed in the previous step).
3. Repeat step 2 until there are no more bounding boxes to get rid of.

This simple approach to object detection works pretty well, but it requires running the CNN many times (15 times in this example), so it is quite slow. Fortunately, there is a much faster way to slide a CNN across an image: using a *fully convolutional network* (FCN).

Fully Convolutional Networks

Fully Convolutional Networks

The idea of FCNs was first introduced in a 2015 paper[33] by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to). The authors pointed out that you could replace the dense layers at the top of a CNN with convolutional layers. To understand this, let's look at an example: suppose a dense layer with 200 neurons sits on top of a convolutional layer that outputs 100 feature maps, each of size 7 × 7 (this is the feature map size, not the kernel size). Each neuron will compute a weighted sum of all 100 × 7 × 7 activations from the convolutional layer (plus a bias term). Now let's see what happens if we replace the dense layer with a convolutional layer using 200 filters, each of size 7 × 7, and with `"valid"` padding. This layer will output 200 feature maps, each 1 × 1 (since the kernel is exactly the size of the input feature maps and we are using `"valid"` padding). In other words, it will output 200 numbers, just like the dense layer did; and if you look closely at the computations performed by a convolutional layer, you will notice that these numbers will be precisely the same as those the dense layer produced. The only difference is that the dense layer's output was a tensor of shape [*batch size*, 200], while the convolutional layer will output a tensor of shape [*batch size*, 200, 1, 1].

TIP

To convert a dense layer to a convolutional layer, the number of filters in the convolutional layer must be equal to the number of units in the dense layer, the filter size must be equal to the size of the input feature maps, and you must use `"valid"` padding. The stride may be set to 1 or more, as we will see shortly.

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size[34] (however, it does expect its inputs to have a specific number of channels, since each kernel contains a different set of weights for each input channel). Since an FCN contains only convolutional layers (and pooling layers, which have the same property), it can be trained and executed on images of any size!

For example, suppose we'd already trained a CNN for flower classification and localization, with an extra head for objectness. It was trained on 224 × 224 images, and it outputs 107 values per image:

- The classification head outputs 102 class logits (one per class), trained using the `nn.CrossEntropyLoss`.
- The objectness head outputs a single objectness logit, trained using the `nn.BCELoss`.
- The localization head outputs four numbers describing the bounding box, trained using the CIoU loss.

We can now convert the CNN's dense layers (`nn.Linear`) to convolutional layers (`nn.Conv2d`). In fact, we don't even need to retrain the model; we can just copy the weights from the dense layers to the convolutional layers! Alternatively, we could have converted the CNN into an FCN before training.

Now suppose the last convolutional layer before the output layer (also called the bottleneck layer) outputs 7 × 7 feature maps when the network is fed a 224 × 224 image (see the left side of Figure 12-26). For example, this would be the case if the network contains 5 layers with stride 2 and `"same"` padding, so the spatial dimensions get divided by 2^5 = 32 overall. If we feed the FCN a 448 × 448 image (see the righthand side of Figure 12-26), the bottleneck layer will now output 14 × 14 feature maps. Since the dense output layer was replaced by a convolutional layer using 107 filters of size 7 × 7, with `"valid"` padding and stride 1, the output will be composed of 107 feature maps, each of size 8 × 8 (since 14 – 7 + 1 = 8).

In other words, the FCN will process the whole image only once, and it will output an 8 × 8 grid where each cell contains the predictions for one region of the image: 107 numbers representing 102 class probabilities, 1 objectness score, and 4 bounding box coordinates. It's exactly like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column. To visualize this, imagine chopping the original image into a 14 × 14 grid, then sliding a 7 × 7 window across this grid; there will be 8 × 8 = 64 possible locations for the window, hence 8 × 8 predictions. However, the FCN approach is *much* more efficient, since the network only looks at the image once. In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture, which we'll look at next.

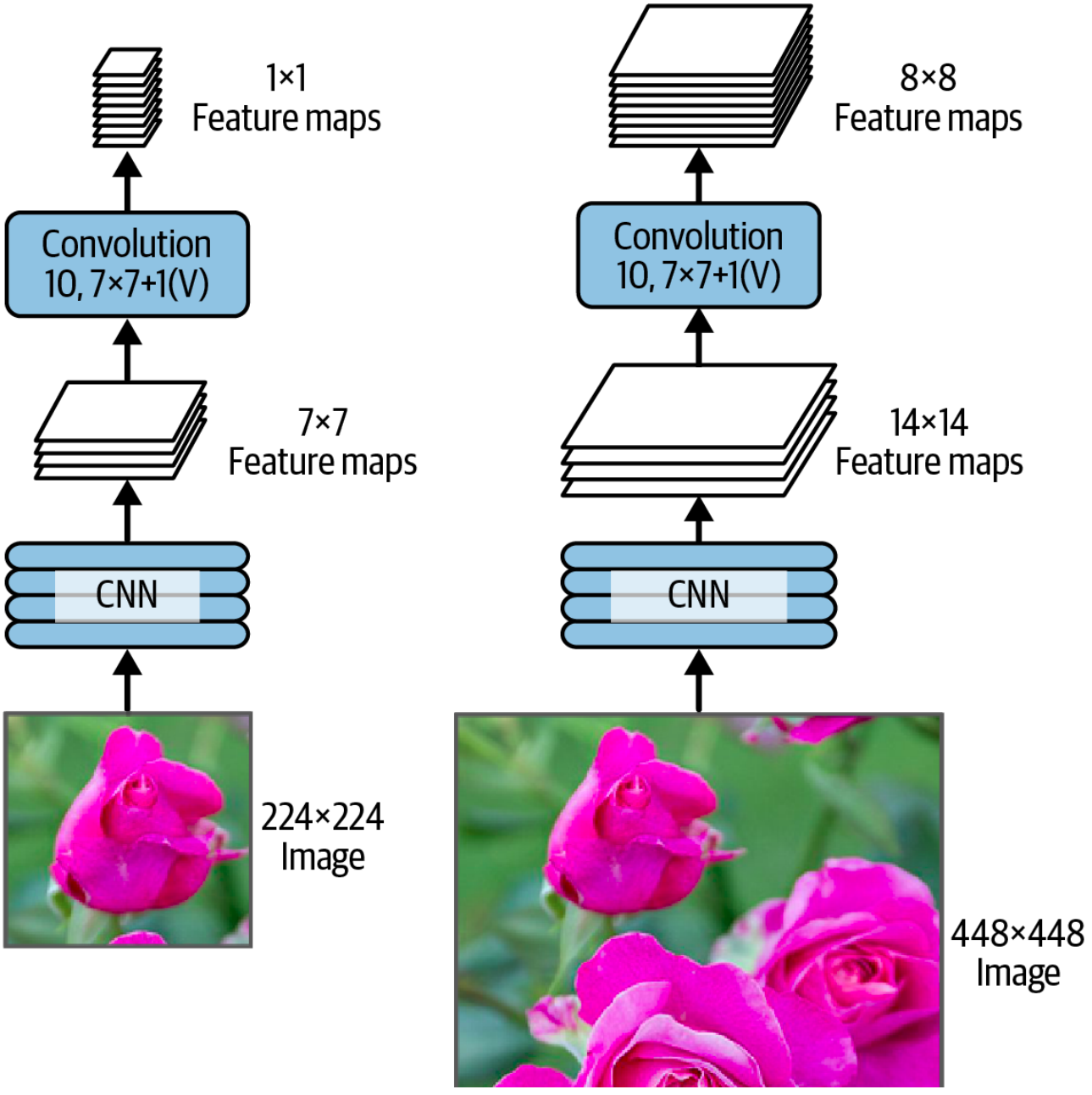


Figure 12-26. The same fully convolutional network processing a small image (left) and a large one (right)

You Only Look Once

YOLO is a fast and accurate object detection architecture proposed by Joseph Redmon et al. in a 2015 paper.[35] It is so fast that it can run in real time on a video, as seen in Redmon's demo. YOLO's architecture is quite similar to the one we just discussed, but with a few important differences:

- For each grid cell, YOLO only considers objects whose bounding box center lies within that cell. The bounding box coordinates are relative to that cell, where (0, 0) means the top-left corner of the cell and (1, 1) means the bottom right. However, the bounding box's height and width may extend well beyond the cell.
- It outputs two bounding boxes for each grid cell (instead of just one), which allows the model to handle cases where two objects are so close to each other that their bounding box centers lie within the same cell. Each bounding box also comes with its own objectness score.
- YOLO also outputs a class probability distribution for each grid cell, predicting 20 class probabilities per grid cell since YOLO was trained on the PASCAL VOC dataset, which contains 20 classes. This produces a coarse *class probability map*. Note that the model predicts one class probability distribution per grid cell, not per bounding box. However, it's possible to estimate class probabilities for each bounding box during post-processing by measuring how well each bounding box matches each class in the class probability map. For example, imagine a picture of a person standing in front of a car. There will be two bounding boxes: one large horizontal one for the car, and a smaller vertical one for the person. These bounding boxes may have their centers within the same grid cell. So how can we tell which class should be assigned to each bounding box? Well, the class probability map will contain a large region where the "car" class is dominant, and inside it there will be a smaller region where the "person" class is dominant. Hopefully, the car's bounding box will roughly match the "car" region, while the person's bounding box will roughly match the "person" region: this will allow the correct class to be assigned to each bounding box.

YOLO was originally developed using Darknet, an open source deep learning framework initially developed in C by Joseph Redmon, but it was soon ported to PyTorch and other libraries. It has been continuously improved over the years, initially by Joseph Redmon et al. (YOLOv2, YOLOv3, and YOLO9000), then by various other teams since 2020. Each version brought some impressive improvements in speed and accuracy, using a variety of techniques; for example, YOLOv3 boosted accuracy in part thanks to *anchor priors*, exploiting the fact that some bounding box shapes are more likely than others, depending on the class (e.g., people tend to have vertical bounding boxes, while cars usually don't). They also increased the number of bounding boxes per grid cell, they trained on different datasets with many more classes (up to 9,000 classes organized in a hierarchy in the case of YOLO9000), they added skip connections to recover some of the spatial resolution that is lost in the CNN (we will discuss this shortly when

we look at semantic segmentation), and much more. There are many variants of these models too, such as scaled down "tiny" YOLOs, optimized to be trained on less powerful machines and which can run extremely fast (at over 1,000 frames per second!), but with a slightly lower *mean average precision* (mAP).

MEAN AVERAGE PRECISION

A very common metric used in object detection tasks is the mean average precision. "Mean average" sounds a bit redundant, doesn't it? To understand this metric, let's go back to two classification metrics we discussed in Chapter 3: precision and recall. Remember the trade-off: in general, the higher the recall, the lower the precision. You can visualize this in a precision/recall curve (see Figure 3-6). To summarize this curve into a single number, we could compute its area under the curve (AUC). But note that the precision/recall curve may contain a few sections where precision actually goes up when recall increases, especially at low recall values (you can see this at the top right of Figure 3-6). This is one of the motivations for the mAP metric.

Suppose the classifier has 90% precision at 10% recall, but 96% precision at 20% recall. There's really no trade-off here: it simply makes more sense to use the classifier at 20% recall rather than at 10% recall, as you will get both higher recall and higher precision. So instead of looking at the precision *at* 10% recall, we should really be looking at the *maximum* precision that the classifier can offer with *at least* 10% recall. It would be 96%, not 90%. Therefore, one way to get a fair idea of the model's performance is to compute the maximum precision you can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions. This is called the *average precision* (AP) metric. Now when there are more than two classes, we can compute the AP for each class, and then compute the mean AP (mAP). Conveniently, the TorchMetrics library implements all of this in the `MeanAveragePrecision` metric.

In an object detection system, there is an additional level of complexity: what if the system detected the correct class, but at the wrong location (i.e., the bounding box is completely off)? Surely we should not count this as a positive prediction. One approach is to define an IoU threshold: for example, we may consider that a prediction is correct only if the IoU is greater than, say, 0.5, and the predicted class is correct. The corresponding mAP is generally noted mAP@0.5 (or mAP@50%, or sometimes just AP_{50}). In some competitions (such as the PASCAL VOC challenge), this is what is done. In others (such as the COCO competition), the mAP is computed for different IoU thresholds (0.50, 0.55, 0.60, ..., 0.95), and the final metric is the mean of all these mAPs (noted mAP@[.50:.95] or mAP@[.50:0.05:.95]). Yes, that's a mean mean average.

TorchVision does not include any YOLO model, but you can use the Ultralytics library, which provides a simple API to download and use various pretrained YOLO models, based on PyTorch. These models were pretrained on the COCO dataset which contains over 330,000 images, including 200,000 images annotated for object detection with 80 different classes (person, car, truck, bicycle, ball, etc.). The Ultralytics library is not installed on Colab by default, so we must run `%pip install ultralytics`. Then we can download a YOLO model and use it. For example, here is how to use this library to download the YOLOv9 model (medium variant) and

example, here is how to use this library to download the YOLOv9 model (medium variant) and detect objects in a batch of images (the model accepts PIL images, NumPy arrays, and even URLs):

```
from ultralytics import YOLO

model = YOLO('yolov9m.pt')  # n=nano, s=small, m=medium, x=large
images = ["https://homl.info/soccer", "https://homl.info/traffic"]
results = model(images)
```

The output is a list of `Results` objects which offers a handy `summary()` method. For example, here is how we can see the first detected object in the first image:

```
>>> results[0].summary()[0]
{'name': 'sports ball',
 'class': 32,
 'confidence': 0.96214,
 'box': {'x1': 245.3573, 'y1': 286.03003, 'x2': 300.62506, 'y2': 343.57184}}
```

TIP

The Ultralytics library also provides a simple API to train a YOLO model on other common object detection datasets, or on your own dataset. See *https://docs.ultralytics.com/modes/train* for more details.

Several other pretrained object detection models are available via TorchVision. You can use them just like the pretrained classification models (e.g., ConvNeXt), except that each image prediction is a represented as a dictionary containing two entries: `"labels"` (i.e., class IDs) and `"boxes"`. The available models are listed below (see the models page for the full list of variants available):

Faster R-CNN[36]

This model has two stages: the image first goes through a CNN, then the output is passed to a *region proposal network* (RPN) that proposes bounding boxes that are most likely to contain an object; a classifier is then run for each bounding box, based on the cropped output of the CNN.

SSD[37]

SSD is a single-stage detector ("look once") similar to YOLO.

SSDlite[38]

A lightweight version of SSD, well suited for mobile devices.

RetinaNet[39]

A single-stage detector which introduced a variant of the cross-entropy loss called the *fo-*

cal loss (see `torchvision.ops.sigmoid_focal_loss()`). This loss gives more weight to difficult samples and thereby improves performance on small objects and less frequent classes.

FCOS[40]

A single-stage fully convolutional net which directly predicts bounding boxes without relying on anchor boxes.

So far, we've only considered detecting objects in single images. But what about videos? Objects must not only be detected in each frame, they must also be tracked over time. Let's take a quick look at object tracking now.

Object Tracking

Object tracking is a challenging task: objects move, they may grow or shrink as they get closer or further away, their appearance may change as they turn around or move to different lighting conditions or backgrounds, they may be temporarily occluded by other objects, and so on.

One of the most popular object tracking systems is DeepSORT.[41] It is based on a combination of classical algorithms and deep learning:

- It uses *Kalman filters* to estimate the most likely current position of an object given prior detections, and assuming that objects tend to move at a constant speed.
- It uses a deep learning model to measure the resemblance between new detections and existing tracked objects.
- Lastly, it uses the *Hungarian algorithm* to map new detections to existing tracked objects (or to new tracked objects). This algorithm efficiently finds the combination of mappings that minimizes the distance between the detections and the predicted positions of tracked objects, while also minimizing the appearance discrepancy.

For example, imagine a red ball that just bounced off a blue ball traveling in the opposite direction. Based on the previous positions of the balls, the Kalman filter will predict that the balls will go through each other; indeed, it assumes that objects move at a constant speed, so it will not expect the bounce. If the Hungarian algorithm only considered positions, then it would happily map the new detections to the wrong balls, as if they had just gone through each other and swapped colors. But thanks to the resemblance measure, the Hungarian algorithm will notice the problem. Assuming the balls are not too similar, the algorithm will map the new detections to the correct balls.

The Ultralytics library supports object tracking. It uses the Bot-SORT algorithm by default: this algorithm is very similar to DeepSORT but it's faster and more accurate thanks to improvements such as camera-motion compensation and tweaks to the Kalman filter.[42] For example, we can track objects in a video using the YOLOv9 model we created earlier by executing the following code. In this example, we also print the ID of each tracked object at every frame, and we save a copy of the video with annotations (its path is displayed at the end):

We save a copy of the video with annotations (its path is displayed at the end):

```
my_video = "https://homl.info/cars.mp4"
results = model.track(source=my_video, stream=True, save=True)
for frame_results in results:
    summary = frame_results.summary()  # similar summary as earlier + track id
    track_ids = [obj["track_id"] for obj in summary]
    print("Track ids:", track_ids)
```

So far we have located objects using bounding boxes. This is often sufficient, but sometimes you need to locate objects with much more precision—for example, to remove the background behind a person during a videoconference call. Let's see how to go down to the pixel level.

Semantic Segmentation

In *semantic segmentation*, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.), as shown in Figure 12-27. Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the righthand side of the segmented image end up as one big lump of pixels. The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1); so, a regular CNN may end up knowing that there's a person somewhere in the bottom left of the image, but it might not be much more precise than that.

Figure 12-27. Semantic segmentation

Just like for object detection, there are many different approaches to tackle this problem, some quite complex. However, a fairly simple solution was proposed in the 2015 paper by Jonathan Long et al., that I mentioned earlier, on fully convolutional networks. The authors start by taking a pretrained CNN and turning it into an FCN. The CNN applies an overall stride of 32 to the input image (i.e., if you multiply all the strides), meaning the last layer outputs feature maps that are 32 times smaller than the input image. This is clearly too coarse, so they added a single *upsampling layer* that multiplies the resolution by 32.

There are several solutions available for upsampling (increasing the size of an image), such as bilinear interpolation, but that only works reasonably well up to ×4 or ×8. Instead, they use a

bilinear interpolation, but that only works reasonably well up to ×4 or ×8. Instead, they use a *transposed convolutional layer*:[43] this is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 12-28). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., the stride is 1/2 in Figure 12-28). The transposed convolutional layer can be initialized to perform something close to linear interpolation, but since it is a trainable layer, it will learn to do better during training. In PyTorch, you can use the `nn.ConvTranspose2d` layer.

NOTE

In a transposed convolutional layer, the stride defines how much the input will be stretched, not the size of the filter steps, so the larger the stride, the larger the output (unlike for convolutional layers or pooling layers).

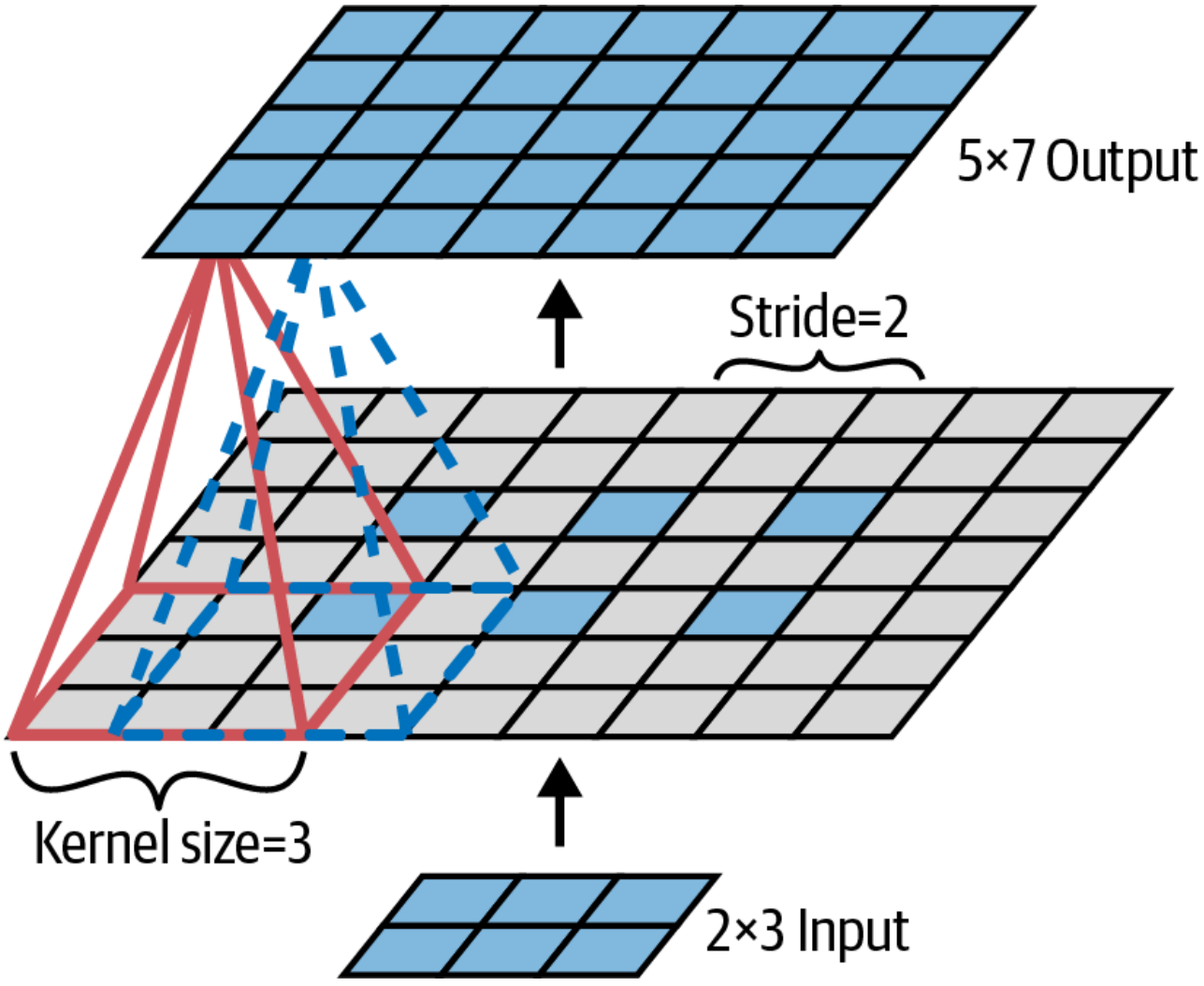


Figure 12-28. Upsampling using a transposed convolutional layer

---

OTHER PYTORCH CONVOLUTIONAL LAYERS

PyTorch also offers a few other kinds of convolutional layers:

`nn.Conv1d`

A convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as you will see in Chapter 13.

`nn.Conv3d`

A convolutional layer for 3D inputs, such as 3D PET scans.

À-trous convolutional layer

Setting the `dilation` hyperparameter of any convolutional layer to a value of 2 or more creates an *à-trous convolutional layer* (*à trous* is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1 × 3 filter equal to `[[1,2,3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* of `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This lets the convolutional layer have a larger receptive field at no computational cost and using no extra parameters.

Using transposed convolutional layers for upsampling is OK, but still too imprecise. To do better, Long et al. added skip connections from lower layers: for example, they upsampled the output image by a factor of 2 (instead of 32), and they added the output of a lower layer that had this double resolution. Then they upsampled the result by a factor of 16, leading to a total upsampling factor of 32 (see Figure 12-29). This recovered some of the spatial resolution that was lost in earlier pooling layers. In their best architecture, they used a second similar skip connection to recover even finer details from an even lower layer. In short, the output of the original CNN goes through the following extra steps: upsample ×2, add the output of a lower layer (of the appropriate scale), upsample ×2, add the output of an even lower layer, and finally upsample ×8. It is even possible to scale up beyond the size of the original image: this can be used to increase the resolution of an image, which is a technique called *super-resolution*.

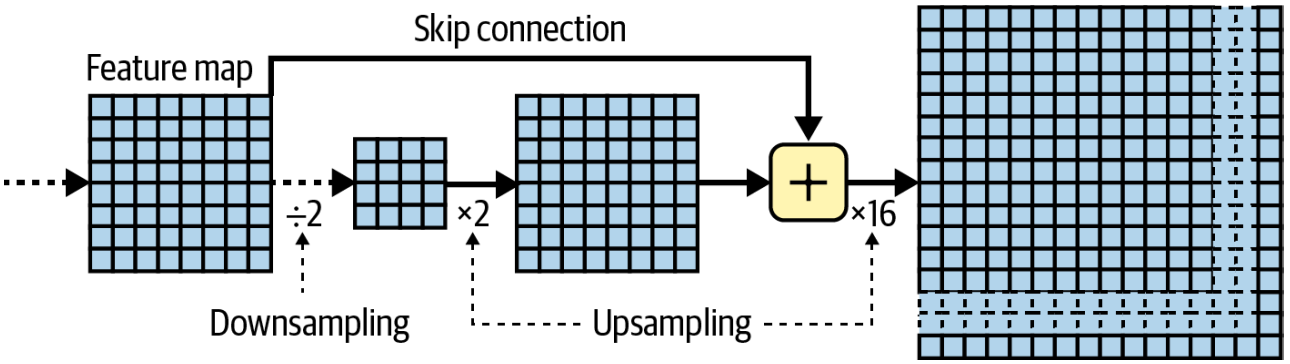


Figure 12-29. Skip layers recover some spatial resolution from lower layers

---

TIP

The FCN model is available in TorchVision, along with a couple other semantic segmentation models. See the notebook for a code example.

Instance segmentation is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle). For example the *Mask R-CNN* architecture, proposed in a 2017 paper[44] by Kaiming He et al., extends the Faster R-CNN model by additionally producing a pixel mask

by Kaiming He et al., extends the Faster R-CNN model by additionally producing a pixel mask for each bounding box. So not only do you get a bounding box around each object, with a set of estimated class probabilities, you also get a pixel mask that locates pixels in the bounding box that belong to the object. This model is available in TorchVision, pretrained on the COCO 2017 dataset.

TIP

TorchVision's transforms API v2 can apply to masks and videos, just like it applies to bounding boxes, thanks to the `Video` and `Mask` TVTensors.

As you can see, the field of deep computer vision is vast and fast-paced, with all sorts of architectures popping up every year. If you want to try the latest and greatest models, check out the state-of-the-art section of *https://paperswithcode.com*. Most of them used to be based on convolutional neural networks, but since 2020 another neural net architecture has entered the computer vision space: Transformers (which we will discuss in Chapter 14). The progress made over the last 15 years has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts to make the network more resistant to images designed to fool it), *explainability* (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in Chapter 18), *single-shot learning* (a system that can recognize an object after it has seen it just once), predicting the next frames in a video, combining text and image tasks, and more.

Now on to the next chapter, where we will look at how to process sequential data such as time series using recurrent neural networks and convolutional neural networks.

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3 × 3 kernels, a stride of 2, and `"same"` padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200 × 300 pixels:
   a. What is the total number of parameters in the CNN?
   b. If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance?
   c. What about when training on a mini-batch of 50 images?
3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. Can you name the main innovations in AlexNet, as compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, Xception, EfficientNet, and ConvNeXt?
6. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?

7. What is the main technical difficulty of semantic segmentation?
8. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
9. Use transfer learning for large image classification, going through these steps:
    a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.). Alternatively, you can use an existing dataset, such as the one used in PyTorch's transfer learning for computer vision tutorial.
    b. Split it into a training set, a validation set, and a test set.
    c. Build the input pipeline, apply the appropriate preprocessing operations, and optionally add data augmentation.
    d. Fine-tune a pretrained model on this dataset.
10. Go through PyTorch's object detection fine-tuning tutorial.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

[1] David H. Hubel, "Single Unit Activity in Striate Cortex of Unrestrained Cats", *The Journal of Physiology* 147 (1959): 226–238.

[2] David H. Hubel and Torsten N. Wiesel, "Receptive Fields of Single Neurons in the Cat's Striate Cortex", *The Journal of Physiology* 148 (1959): 574–591.

[3] David H. Hubel and Torsten N. Wiesel, "Receptive Fields and Functional Architecture of Monkey Striate Cortex", *The Journal of Physiology* 195 (1968): 215–243.

[4] Kunihiko Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological Cybernetics* 36 (1980): 193–202.

[5] Yann LeCun et al., "Gradient-Based Learning Applied to Document Recognition", *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

[6] A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform, and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions (see *https://homl.info/76* for more details).

[7] Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

[8] Yann LeCun et al., "Gradient-Based Learning Applied to Document Recognition", *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

[9] Alex Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural Networks", *Proceedings of the 25th International Conference on Neural Information Processing Systems* 1 (2012): 1097–1105.

[10] Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks", *Proceedings of the European Conference on Computer Vision* (2014): 818–833.

[11] Christian Szegedy et al., “Going Deeper with Convolutions”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1–9.

[12] In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

[13] Kaiming He et al., “Deep Residual Learning for Image Recognition”, arXiv preprint arXiv:1512:03385 (2015).

[14] Gao Huang, Yu Sun, et al., “Deep Networks with Stochastic Depth”, arXiv preprint arXiv:1603.09382 (2016).

[15] It is a common practice when describing a neural network to count only layers with parameters.

[16] Christian Szegedy et al., “Inception–v4, Inception-ResNet and the Impact of Residual Connections on Learning”, arXiv preprint arXiv:1602.07261 (2016).

[17] François Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions”, arXiv preprint arXiv:1610.02357 (2016).

[18] This name can sometimes be ambiguous, since spatially separable convolutions are often called “separable convolutions” as well.

[19] Jie Hu et al., “Squeeze-and-Excitation Networks”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

[20] Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, arXiv preprint arXiv:1409.1556 (2014).

[21] Saining Xie et al., “Aggregated Residual Transformations for Deep Neural Networks”, arXiv preprint arXiv:1611.05431 (2016).

[22] Gao Huang et al., “Densely Connected Convolutional Networks”, arXiv preprint arXiv:1608.06993 (2016).

[23] Andrew G. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”, arXiv preprint arXiv:1704.04861 (2017).

[24] Chien-Yao Wang et al., “CSPNet: A New Backbone That Can Enhance Learning Capability of CNN”, arXiv preprint arXiv:1911.11929 (2019).

[25] Mingxing Tan and Quoc V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, arXiv preprint arXiv:1905.11946 (2019).

[26] Zhuang Liu et al, “A ConvNet for the 2020s”, arXiv preprint arXiv:2201.03545 (2022).

[27] In the international system of units (SI), 1 MB = 1,000 KB = 1,000 × 1,000 bytes = 1,000 × 1,000 × 8 bits. And 1 MiB = 1,024 kiB = 1,024 × 1,024 bytes. So 12 MB $\approx$ 11.44 MiB.

[28] Aidan Gomez et al., “The Reversible Residual Network: Backpropagation Without Storing Activations”, arXiv preprint arXiv:1707.04585 (2017).

[29] M. Nilsback and A. Zisserman, “Automated Flower Classification over a Large Number of Classes”, *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing* (2008).

[30] TorchVision PR #8838 might have fixed this by the time your read these lines.

[31] H. Rezatofighi et al., “Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression”, arXiv preprint arXiv:1902.09630 (2019).

[32] Z. Zheng et al., “Enhancing Geometric Factors in Model Learning and Inference for Object Detection and Instance Segmentation”, arXiv preprint arXiv:2005.03572 (2020).

[33] Jonathan Long et al., “Fully Convolutional Networks for Semantic Segmentation”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.

[34] There is one small exception: a convolutional layer using `"valid"` padding will complain if the input size is smaller than the kernel size.

[35] Joseph Redmon et al., “You Only Look Once: Unified, Real-Time Object Detection”, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016): 779–788.

[36] Shaoqing Ren et al., “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”, *Proceedings of e 28th International Conference on Neural Information Processing Systems* 1 (2015): 91–99.

[37] Wei Liu et al., “SSD: Single Shot Multibox Detector”, *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.

[38] Mark Sandler et al., “MobileNetV2: Inverted Residuals and Linear Bottlenecks”, arXiv preprint arXiv:1801.04381 (2018).

[39] Tsung-Yi Lin et al., “Focal Loss for Dense Object Detection”, arXiv preprint arXiv:1708.02002 (2017).

[40] Zhi Tian et al., “FCOS: Fully Convolutional One-Stage Object Detection”, arXiv preprint arXiv:1904.01355 (2019).

[41] Nicolai Wojke et al., “Simple Online and Realtime Tracking with a Deep Association Metric”, arXiv preprint arXiv:1703.07402 (2017).

[42] Nir Aharon et al., “BoT-SORT: Robust Associations Multi-Pedestrian Tracking”, arXiv preprint arXiv:2206.14651 (2022).

[43] This type of layer is sometimes referred to as a *deconvolution layer*, but it does *not* perform what mathematicians call a deconvolution, so this name should be avoided.

[44] Kaiming He et al., “Mask R-CNN”, arXiv preprint arXiv:1703.06870 (2017).