Chapter 11. Training Deep Neural Networks

In Chapter 10 you built, trained, and fine-tuned your first artificial neural networks. But they were shallow nets, with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper ANN, perhaps with 10 layers or many more, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep neural network isn't a walk in the park. Here are some of the problems you could run into:

- You may be faced with the problem of gradients growing ever smaller or larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss various optimizers that can speed up training large models tremendously. Finally, we will cover a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets. Welcome to deep learning!

The Vanishing/Exploding Gradients Problems

As discussed in Chapter 10, the backpropagation algorithm's second phase works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a gradient descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (see Chapter 15). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a 2010 paper by Xavier Glorot and Yoshua Bengio. [1] The authors found a few suspects, including the combination of the popular sigmoid (logistic) activation function and the weight initialization technique that was most popular at the time (i.e., a normal distribution with a mean of 0 and a standard deviation of 1). In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This saturation is actually made worse by the fact that the sigmoid function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the sigmoid function in deep networks).

Looking at the sigmoid activation function (see Figure 11-1), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes). Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

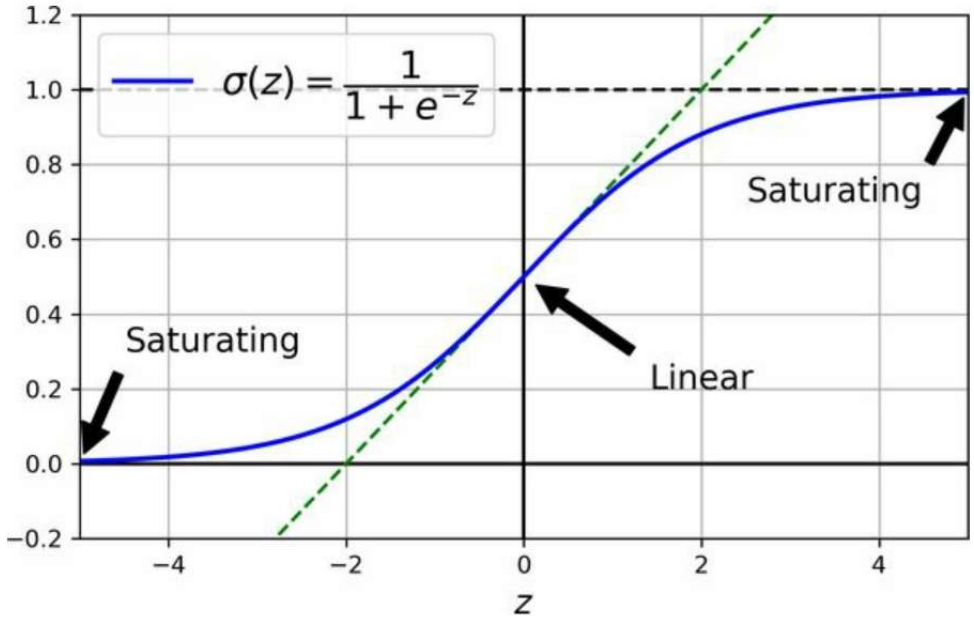

Figure 11-1. Sigmoid activation function saturation

Glorot and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate the unstable gradients problem. They point out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, [2] and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs (these numbers are called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in Equation 11-1, where $fan_{avg} = (fan_{in} + fan_{out}) / 2$. This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

Equation 11-1. Glorot initialization (when using the sigmoid activation function)

Normal distribution with mean 0 and variance σ2=1fanavgOr a uniform distribution between -r and +r, with r=3fanavg

If you replace fan_{avg} with fan_{in} in Equation 11-1, you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when $fan_{in} = fan_{out}$. It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the practices that led to the success of deep learning.

Some papers [3] have provided similar strategies for different activation functions. These strategies differ only by the scale of the variance and

whether they use fan_{avg} or fan_{in}, as shown in Table 11-1 (for the uniform distribution, just use r=3σ2). The initialization strategy proposed for the ReLU activation function and its variants is called *He initialization* or *Kaiming initialization*, after the paper's first author. For SELU, use Yann LeCun's initialization method, preferably with a normal distribution. We will cover all these activation functions shortly.

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, sigmoid, softmax	$1 / fan_{avg}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

By default, Keras uses Glorot initialization with a uniform distribution. When you create a layer, you can switch to He initialization by setting kernel_initializer="he_uniform" or kernel_initializer="he_normal" like this:

```
import tensorflow as tf

dense = tf.keras.layers.Dense(50, activation="relu",
                              kernel_initializer="he_normal")
```

Alternatively, you can obtain any of the initializations listed in Table 11-1 and more using the VarianceScaling initializer. For example, if you want He initialization with a uniform distribution and based on fan_{avg} (rather than fan_{in}), you can use the following code:

```
he_avg_init = tf.keras.initializers.VarianceScaling(scale=2., mode="fan_avg",
                                                    distribution="uniform")
dense = tf.keras.layers.Dense(50, activation="sigmoid",
                              kernel_initializer=he_avg_init)
```

Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the ReLU activation function, mostly because it does not saturate for positive values, and also because it is very fast to compute.

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively "die", meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron's inputs plus its bias term) is negative for all instances in the training set. When this happens, it just keeps outputting zeros, and gradient descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative. [4]

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*.

Leaky ReLU

The leaky ReLU activation function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$. Having a slope for $z < 0$ ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A 2015 paper by Bing Xu et al. [5] compared several variants of the ReLU activation function, and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting $\alpha = 0.2$ (a huge leak) seemed to result in

better performance than $\alpha = 0.01$ (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting the training set. Finally, the paper evaluated the *parametric leaky ReLU* (PReLU), where α is authorized to be learned during training: instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

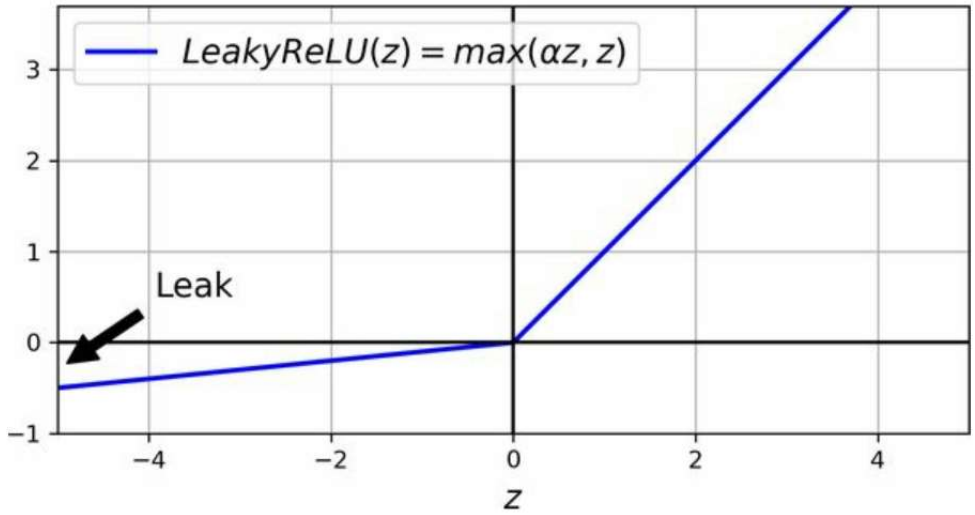

Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

Keras includes the classes LeakyReLU and PReLU in the tf.keras.layers package. Just like for other ReLU variants, you should use He initialization with these. For example:

```
leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2)  # defaults to alpha=0.3
dense = tf.keras.layers.Dense(50, activation=leaky_relu,
                              kernel_initializer="he_normal")
```

If you prefer, you can also use LeakyReLU as a separate layer in your model; it makes no difference for training and predictions:

```
model = tf.keras.models.Sequential([
    [...]  # more layers
    tf.keras.layers.Dense(50, kernel_initializer="he_normal"),  # no activation
    tf.keras.layers.LeakyReLU(alpha=0.2),  # activation as a separate layer
    [...]  # more layers
])
```

For PReLU, replace LeakyReLU with PReLU. There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own (to learn how to do that, see the exercises at the end of Chapter 12).

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their derivatives abruptly change (at $z = 0$). As we saw in Chapter 4 when we discussed lasso, this sort of discontinuity can make gradient descent bounce around the optimum, and slow down convergence. So now we will look at some smooth variants of the ReLU activation function, starting with ELU and SELU.

ELU and SELU

In 2015, a paper by Djork-Arné Clevert et al. [6] proposed a new activation function, called the *exponential linear unit* (ELU), that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. Equation 11-2 shows this activation function's definition.

Equation 11-2. ELU activation function

$$\text{ELU}_{\alpha}(z) = \alpha(\exp(z) - 1) \text{ if } z < 0 \quad z \text{ if } z \geq 0$$

The ELU activation function looks a lot like the ReLU function (see Figure 11-3), with a few major differences:

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter α defines the opposite of the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.
- It has a nonzero gradient for $z < 0$, which avoids the dead neurons

problem.

- If α is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps speed up gradient descent since it does not bounce as much to the left and right of $z = 0$.

Using ELU with Keras is as easy as setting activation="elu", and like with other ReLU variants, you should use He initialization. The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

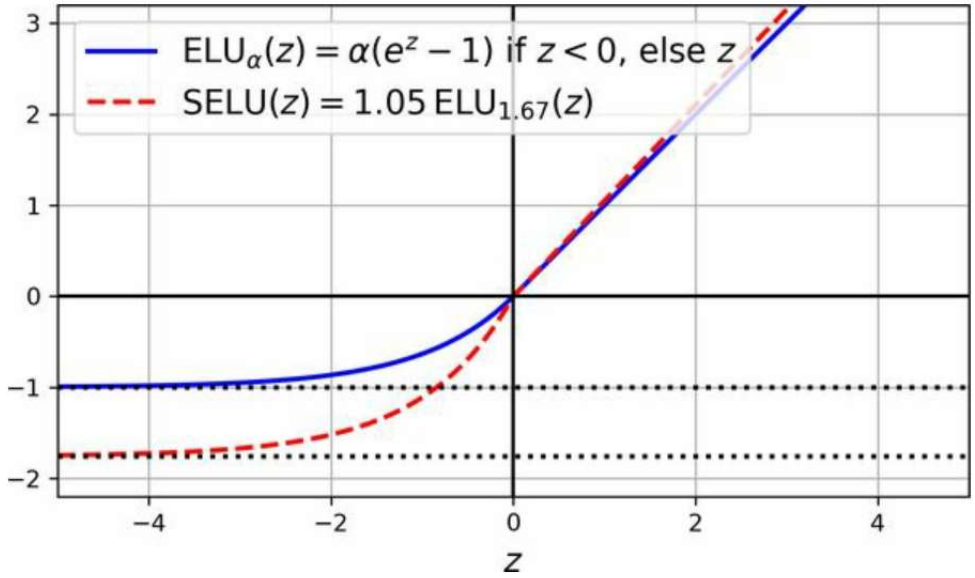

Figure 11-3. ELU and SELU activation functions

Not long after, a 2017 paper by Günter Klambauer et al. [7] introduced the *scaled ELU* (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function (about 1.05 times ELU, using $\alpha \approx 1.67$). The authors showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the

output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function may outperform other activation functions for MLPs, especially deep ones. To use it with Keras, just set activation="selu". There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization. In Keras, this means setting kernel_initializer="lecun_normal".
- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see Chapter 15) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), it will probably not outperform ELU.
- You cannot use regularization techniques like ℓ_1 or ℓ_2 regularization, max-norm, batch-norm, or regular dropout (these are discussed later in this chapter).

These are significant constraints, so despite its promises, SELU did not gain a lot of traction. Moreover, three more activation functions seem to outperform it quite consistently on most tasks: GELU, Swish, and Mish.

GELU, Swish, and Mish

GELU was introduced in a 2016 paper by Dan Hendrycks and Kevin Gimpel.[8] Once again, you can think of it as a smooth variant of the ReLU activation function. Its definition is given in Equation 11-3, where Φ is the standard Gaussian cumulative distribution function (CDF): $\Phi(z)$ corresponds to the probability that a value sampled randomly from a normal distribution of mean 0 and variance 1 is lower than z.

Equation 11-3. GELU activation function

$$\text{GELU}(z) = z\Phi(z)$$

As you can see in Figure 11-4, GELU resembles ReLU: it approaches 0 when its input z is very negative, and it approaches z when z is very positive. However, whereas all the activation functions we've discussed so far were both convex and monotonic,[9] the GELU activation function is neither: from left to right, it starts by going straight, then it wiggles down, reaches a low point around –0.17 (near $z \approx -0.75$), and finally bounces up and ends up going straight toward the top right. This fairly complex shape and the fact that it has a curvature at every point may explain why it works so well, especially for complex tasks: gradient descent may find it easier to fit complex patterns. In practice, it often outperforms every other activation function discussed so far. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. That said, it is possible to show that it is approximately equal to $z\sigma(1.702\, z)$, where σ is the sigmoid function: using this approximation also works very well, and it has the advantage of being much faster to compute.

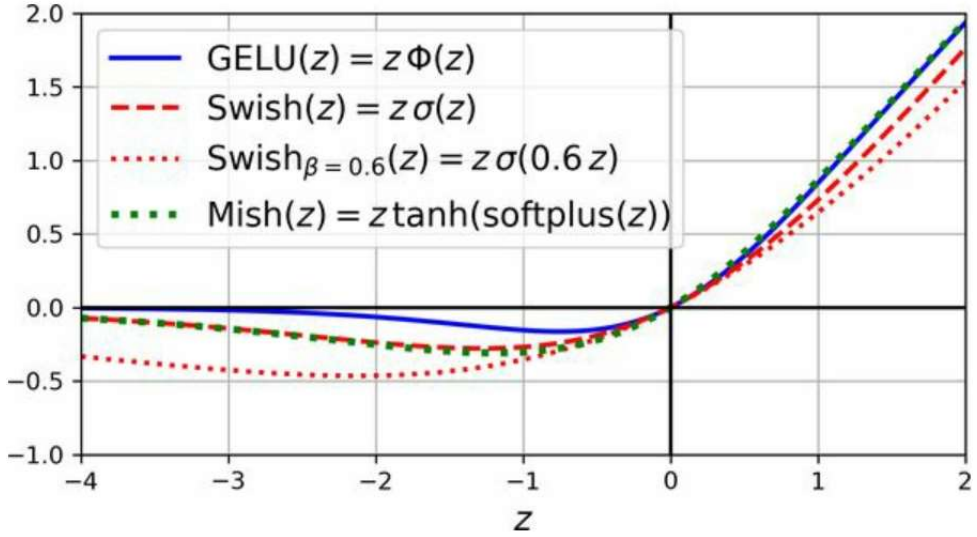

Figure 11-4. GELU, Swish, parametrized Swish, and Mish activation functions

The GELU paper also introduced the *sigmoid linear unit* (SiLU) activation function, which is equal to $z\sigma(z)$, but it was outperformed by GELU in the authors' tests. Interestingly, a 2017 paper by Prajit Ramachandran et al.[10] rediscovered the SiLU function by automatically searching for good activation functions. The authors named it *Swish*, and the name caught on. In their paper, Swish outperformed every other function, including GELU. Ramachandran et al. later generalized Swish by adding an extra hyperparameter β to scale the sigmoid function's input. The generalized Swish function is $\text{Swish}_\beta(z) = z\sigma(\beta z)$, so GELU is approximately equal to the generalized Swish function using $\beta = 1.702$. You can tune β like any other hyperparameter. Alternatively, it's also possible to make β trainable and let gradient descent optimize it: much like PReLU, this can make your model more powerful, but it also runs the risk of overfitting the data.

Another quite similar activation function is *Mish*, which was introduced in a 2019 paper by Diganta Misra.[11] It is defined as $\text{mish}(z) = z\tanh(\text{softplus}(z))$, where $\text{softplus}(z) = \log(1 + \exp(z))$. Just like GELU and Swish, it is a smooth, nonconvex, and nonmonotonic variant of ReLU, and once again the author ran many experiments and found that Mish generally outperformed other activation functions—even Swish and GELU, by a tiny margin. Figure 11-4 shows GELU, Swish (both with the default $\beta = 1$ and with $\beta = 0.6$), and lastly Mish. As you can see, Mish overlaps almost perfectly with Swish when z is negative, and almost perfectly with GELU when z is positive.

TIP

So, which activation function should you use for the hidden layers of your deep neural networks? ReLU remains a good default for simple tasks: it's often just as good as the more sophisticated activation functions, plus it's very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations. However, Swish is probably a better default for more complex tasks, and you can even try parametrized Swish with a learnable β parameter for the most complex tasks. Mish may give you slightly better results, but it requires a bit more compute. If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for more complex tasks. For deep MLPs, give SELU a try, but make sure to respect the constraints listed earlier. If you have spare time and computing power, you can use cross-validation to evaluate other activation functions as well.

Keras supports GELU and Swish out of the box; just use activation="gelu" or activation="swish". However, it does not support Mish or the generalized Swish activation function yet (but see Chapter 12 to see how to implement your own activation functions and layers).

That’s all for activation functions! Now, let’s look at a completely different way to solve the unstable gradients problem: batch normalization.

Batch Normalization

Although using He initialization along with ReLU (or any of its variants) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a 2015 paper, [12] Sergey Ioffe and Christian Szegedy proposed a technique called *batch normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set. That is, there's no need for StandardScaler or Normalization; the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "batch normalization"). The whole operation is summarized step by step in Equation 11-4.

Equation 11-4. Batch normalization algorithm

1 . μ B = 1 m B ∑ i=1 m B x (i) 2 . σ B 2 = 1 m B ∑ i=1 m B (x (i) -μ B) 2 3 . x ^ (i) = x (i) -μ B σ B 2 +ε 4 . z (i) = γ ⊗ x ^ (i) + β

In this algorithm:

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch *B* (it contains one mean per input).
- m_B is the number of instances in the mini-batch.

- $\boldsymbol{\sigma}_B$ is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- $\mathbf{x}^{\wedge(i)}$ is the vector of zero-centered and normalized inputs for instance *i*.
- ε is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically 10^{-5}). This is called a *smoothing term*.
- $\boldsymbol{\gamma}$ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's mean and standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable. One solution could be to wait until the end of training, then run the whole training set through the neural network and compute the mean and standard deviation of each input of the BN layer. These "final" input means and standard deviations could then be used instead of the batch input means and standard deviations when making predictions. However, most implementations of batch normalization estimate these final statistics during training by using a moving average of the layer's input means and standard deviations. This is what Keras does automatically when you use the BatchNormalization layer. To sum up, four parameter vectors are learned in

each batch-normalized layer: **γ** (the output scale vector) and **β** (the output offset vector) are learned through regular backpropagation, and **μ** (the final input mean vector) and **σ** (the final input standard deviation vector) are estimated using an exponential moving average. Note that **μ** and **σ** are estimated during training, but they are used only after training (to replace the batch input means and standard deviations in Equation 11-4).

Ioffe and Szegedy demonstrated that batch normalization considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the sigmoid activation function. The networks were also much less sensitive to the weight initialization. The authors were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that:

> *Applied to a state-of-the-art image classification model, batch normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. […] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.*

Finally, like a gift that keeps on giving, batch normalization acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in this chapter).

Batch normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it’s often possible to fuse the BN layer with the previous layer after training, thereby avoiding the runtime penalty. This is done by updating the previous layer’s weights and biases so that it directly produces outputs of

the appropriate scale and offset. For example, if the previous layer computes $\mathbf{XW} + \mathbf{b}$, then the BN layer will compute $\boldsymbol{\gamma} \otimes (\mathbf{XW} + \mathbf{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$ (ignoring the smoothing term ε in the denominator). If we define $\mathbf{W}' = \boldsymbol{\gamma} \otimes \mathbf{W} / \boldsymbol{\sigma}$ and $\mathbf{b}' = \boldsymbol{\gamma} \otimes (\mathbf{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$, the equation simplifies to $\mathbf{XW}' + \mathbf{b}'$. So, if we replace the previous layer's weights and biases ($\mathbf{W}$ and $\mathbf{b}$) with the updated weights and biases ($\mathbf{W}'$ and $\mathbf{b}'$), we can get rid of the BN layer (TFLite's converter does this automatically; see Chapter 19).

> **NOTE**
>
> You may find that training is rather slow, because each epoch takes much more time when you use batch normalization. This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance. All in all, *wall time* will usually be shorter (this is the time measured by the clock on your wall).

Implementing batch normalization with Keras

As with most things with Keras, implementing batch normalization is straightforward and intuitive. Just add a BatchNormalization layer before or after each hidden layer's activation function. You may also add a BN layer as the first layer in your model, but a plain Normalization layer generally performs just as well in this location (its only drawback is that you must first call its adapt() method). For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images):

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(300, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

That's all! In this tiny example with just two hidden layers batch normalization is unlikely to have a large impact, but for deeper networks it can make a tremendous difference.

Let's display the model summary:

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
_________________________________________________________________
```

As you can see, each BN layer adds four parameters per input: **γ**, **β**, **μ**, and **σ** (for example, the first BN layer adds 3,136 parameters, which is 4 × 784). The last two parameters, **μ** and **σ**, are the moving averages; they are not affected by backpropagation, so Keras calls them "non-trainable" [13] (if you count the total number of BN parameters, 3,136 + 1,200 + 400, and divide by 2, you get 2,368, which is the total number of non-trainable parameters in this model).

Let's look at the parameters of the first BN layer. Two are trainable (by backpropagation), and two are not:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
```

```
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True),
 ('batch_normalization/moving_mean:0', False),
 ('batch_normalization/moving_variance:0', False)]
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, as which is preferable seems to depend on the task—you can experiment with this too to see which option works best on your dataset. To add the BN layers before the activation function, you must remove the activation functions from the hidden layers and add them as separate layers after the BN layers. Moreover, since a batch normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer by passing use_bias=False when creating it. Lastly, you can usually drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers. The updated code looks like this:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

The BatchNormalization class has quite a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the momentum. This hyperparameter is used by the BatchNormalization layer when it updates the exponential moving averages; given a new value **v** (i.e., a new vector of input means or standard deviations computed over the current batch), the layer updates the running average $\hat{v}$ using the following equation:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

A good momentum value is typically close to 1; for example, 0.9, 0.99, or

0.999. You want more 9s for larger datasets and for smaller mini-batches.

Another important hyperparameter is axis: it determines which axis should be normalized. It defaults to –1, meaning that by default it will normalize the last axis (using the means and standard deviations computed across the *other* axes). When the input batch is 2D (i.e., the batch shape is [*batch size, features*]), this means that each input feature will be normalized based on the mean and standard deviation computed across all the instances in the batch. For example, the first BN layer in the previous code example will independently normalize (and rescale and shift) each of the 784 input features. If we move the first BN layer before the Flatten layer, then the input batches will be 3D, with shape [*batch size, height, width*]; therefore, the BN layer will compute 28 means and 28 standard deviations (1 per column of pixels, computed across all instances in the batch and across all rows in the column), and it will normalize all pixels in a given column using the same mean and standard deviation. There will also be just 28 scale parameters and 28 shift parameters. If instead you still want to treat each of the 784 pixels independently, then you should set axis=[1, 2].

Batch normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks discussed in (Chapter 14), to the point that it is often omitted in the architecture diagrams: it is assumed that BN is added after every layer. Now let’s look at one last technique to stabilize gradients during training: gradient clipping.

Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *gradient clipping*. [14] This technique is generally used in recurrent neural networks, where using batch normalization is tricky (as you will see in Chapter 15).

In Keras, implementing gradient clipping is just a matter of setting the clipvalue or clipnorm argument when creating an optimizer, like this:

```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile([...], optimizer=optimizer)
```

This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0. This means that all the partial derivatives of the loss (with regard to each and every trainable parameter) will be clipped between –1.0 and 1.0. The threshold is a hyperparameter you can tune. Note that it may change the orientation of the gradient vector. For instance, if the original gradient vector is [0.9, 100.0], it points mostly in the direction of the second axis; but once you clip it by value, you get [0.9, 1.0], which points roughly at the diagonal between the two axes. In practice, this approach works well. If you want to ensure that gradient clipping does not change the direction of the gradient vector, you should clip by norm by setting clipnorm instead of clipvalue. This will clip the whole gradient if its ℓ_2 norm is greater than the threshold you picked. For example, if you set clipnorm=1.0, then the vector [0.9, 100.0] will be clipped to [0.00899964, 0.9999595], preserving its orientation but almost eliminating the first component. If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try clipping by value or clipping by norm, with different thresholds, and see which option performs best on the validation set.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (I will discuss how to find them in Chapter 14). If you find such as neural network, then you can generally reuse most of its layers, except for the top ones. This technique is called *transfer learning*. It will not only speed up training considerably, but also requires significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects, and you now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see Figure 11-5).

> **NOTE**
>
> If the input pictures for your new task don't have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have similar low-level features.

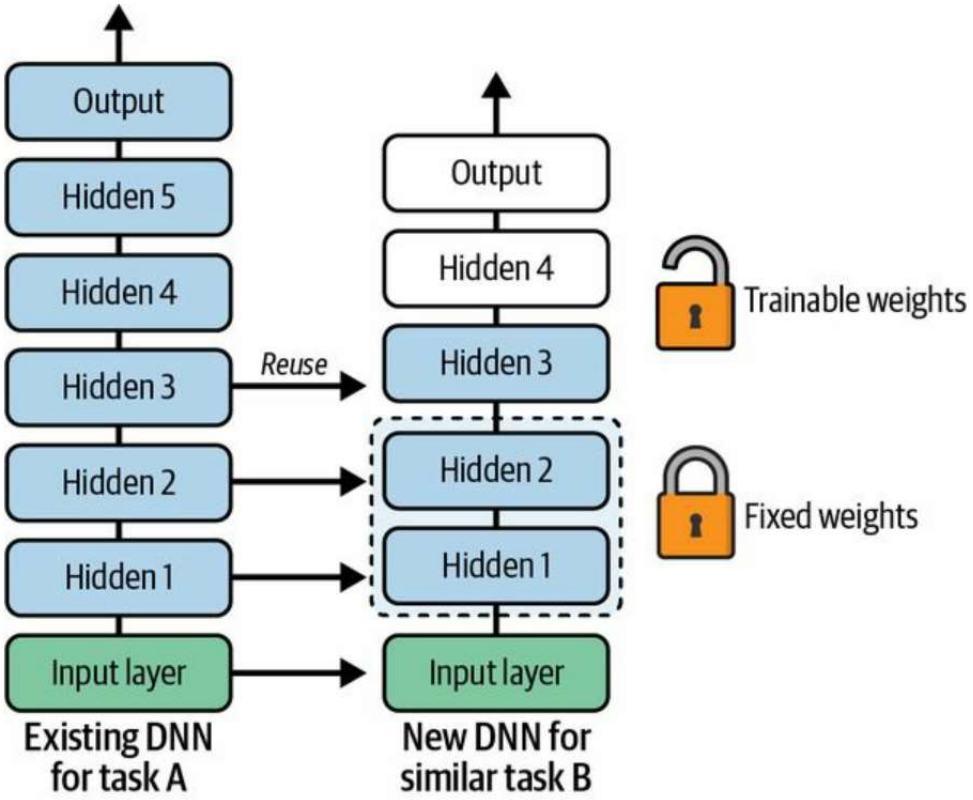

Figure 11-5. Reusing pretrained layers

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and probably will not have the right number of outputs.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

> **TIP**
>
> The more similar the tasks are, the more layers you will want to reuse (starting with the lower layers). For very similar tasks, try to keep all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their weights non-trainable so that gradient descent won’t modify them and they will remain fixed), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

Transfer Learning with Keras

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for sandal and shirt. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let's call this model A. You now want to tackle a different task: you have images of T-shirts and pullovers, and you want to train a binary classifier: positive for T-shirts (and tops), negative for sandals. Your dataset is quite small; you only have 200 labeled images. When you train a new model for this task (let's call it model B) with the same architecture as model A, you get 91.85% test accuracy. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, you need to load model A and create a new model based on that model's layers. You decide to reuse all the layers except for the output layer:

```
[...] # Assuming model A was already trained and saved to "my_model_A"
model_A = tf.keras.models.load_model("my_model_A")
model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

Note that model_A and model_B_on_A now share some layers. When you train model_B_on_A, it will also affect model_A. If you want to avoid that, you need to *clone* model_A before you reuse its layers. To do this, you clone model A's architecture with clone_model(), then copy its weights:

```
model_A_clone = tf.keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

> **WARNING**
>
> tf.keras.models.clone_model() only clones the architecture, not the weights. If you don't copy them manually using set_weights(), they will be initialized randomly when the cloned model is first used.

Now you could train model_B_on_A for task B, but since the new output layer was initialized randomly it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights. To do this, set every layer's trainable attribute to False and compile the model:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
```

> **NOTE**
>
> You must always compile your model after you freeze or unfreeze layers.

Now you can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights.

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

So, what's the final verdict? Well, this model's test accuracy is 93.85%, up

exactly two percentage points from 91.85%! This means that transfer learning reduced the error rate by almost 25%:

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.2546142041683197, 0.9384999871253967]
```

Are you convinced? You shouldn't be: I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses". When a paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results (which may be due to sheer luck), without mentioning how many failures they encountered on the way. Most of the time, this is not malicious at all, but it is part of the reason so many results in science can never be reproduced.

Why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful in other tasks. Transfer learning works best with deep convolutional neural networks, which tend to learn feature detectors that are much more general (especially in the lower layers). We will revisit transfer learning in Chapter 14, using the techniques we just discussed (and this time there will be no cheating, I promise!).

Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see Figure 11-6). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder or a generative adversarial network (GAN; see Chapter 17). Then you can reuse the lower layers of the autoencoder or the lower layers of the GAN's discriminator, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

It is this technique that Geoffrey Hinton and his team used in 2006, and which led to the revival of neural networks and the success of deep learning. Until 2010, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see the notebook at *https://homl.info/extra-anns*)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or GANs rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

Note that in the early days of deep learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in Figure 11-6). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot and use autoencoders or GANs rather than RBMs.

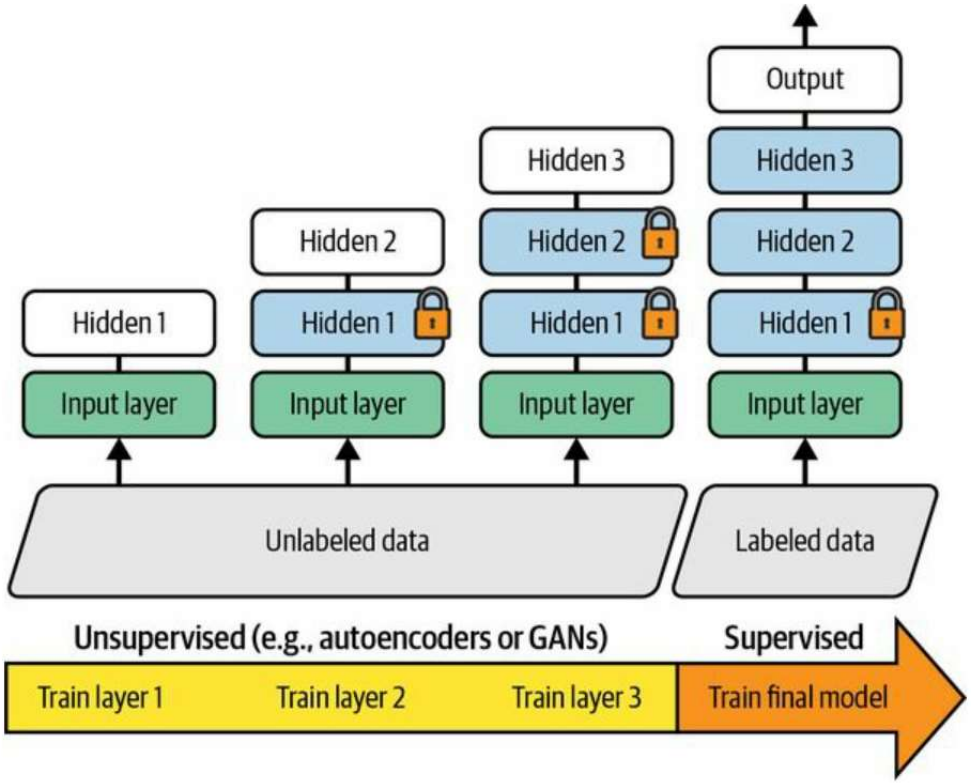

Figure 11-6. In unsupervised training, a model is trained on all data, including the unlabeled data, using an unsupervised learning technique, then it is fine-tuned for the final task on just the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. You could, however, gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

For natural language processing (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence "What ___ you saying?" is probably "are" or "were"). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data (we will discuss more pretraining tasks in Chapter 15).

> **NOTE**
>
> *Self-supervised learning* is when you automatically generate the labels from the data itself, as in the text-masking example, then you train a model on the resulting "labeled" dataset using supervised learning techniques.

Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using batch normalization, and reusing parts of a pretrained network (possibly built for an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular gradient descent optimizer. In this section we will present the most popular optimization algorithms: momentum, Nesterov accelerated gradient, AdaGrad, RMSProp, and finally Adam and its variants.

Momentum

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the core idea behind *momentum optimization*, proposed by Boris Polyak in 1964. [15] In contrast, regular gradient descent will take small steps when the slope is gentle and big steps when the slope is steep, but it will never pick up speed. As a result, regular gradient descent is generally much slower to reach the minimum than momentum optimization.

Recall that gradient descent updates the weights $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regard to the weights ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$) multiplied by the learning rate η. The equation is $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* **m** (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (see Equation 11-5). In other words, the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-5. Momentum algorithm

$$1 . \, m \leftarrow \beta \, m - \eta \, \nabla \, \theta \, J \, (\, \theta \,) \, 2 . \, \theta \leftarrow \theta + m$$

You can verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $1 / (1 - \beta)$ (ignoring the sign). For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10

times faster than gradient descent! This allows momentum optimization to escape from plateaus much faster than gradient descent. We saw in Chapter 4 that when the inputs have very different scales, the cost function will look like an elongated bowl (see Figure 4-7). Gradient descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use batch normalization, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.

> **NOTE**
>
> Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons it's good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing momentum optimization in Keras is a no-brainer: just use the SGD optimizer and set its momentum hyperparameter, then lie back and profit!

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular gradient descent.

Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by Yurii Nesterov in 1983, [16] is almost always faster than regular momentum optimization. The *Nesterov accelerated gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position **θ** but slightly ahead in the direction of the momentum, at **θ** + β**m** (see Equation 11-6).

Equation 11-6. Nesterov accelerated gradient algorithm

$$1.\ \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m}) \quad 2.\ \theta \leftarrow \theta + \mathbf{m}$$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as you can see in Figure 11-7 (where ∇_1 represents the gradient of the cost function measured at the starting point **θ**, and ∇_2 represents the gradient at the point located at **θ** + β**m**).

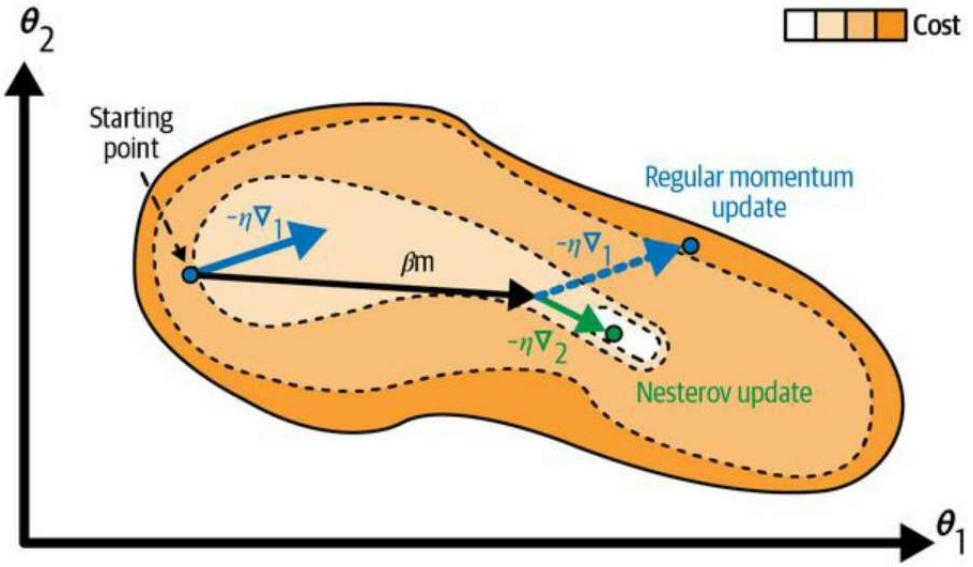

Figure 11-7. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

As you can see, the Nesterov update ends up closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push farther across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

To use NAG, simply set nesterov=True when creating the SGD optimizer:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,
                                    nesterov=True)
```

AdaGrad

Consider the elongated bowl problem again: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm[17] achieves this correction by scaling down the gradient vector along the steepest dimensions (see Equation 11-7).

Equation 11-7. AdaGrad algorithm

$$1.\ s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \quad 2.\ \theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash s + \varepsilon$$

The first step accumulates the square of the gradients into the vector **s** (recall that the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\mathbf{\theta}) / \partial \theta_i)^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regard to parameter θ_i. If the cost function is steep along the i^{th} dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to gradient descent, but with one big difference: the gradient vector is scaled down by a factor of $s+\varepsilon$ (the $\oslash$ symbol represents the element-wise division, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to simultaneously computing $\theta i \leftarrow \theta i - \eta \partial J(\theta)/\partial \theta i / si+\varepsilon$ for all parameters θ_i.

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-8). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

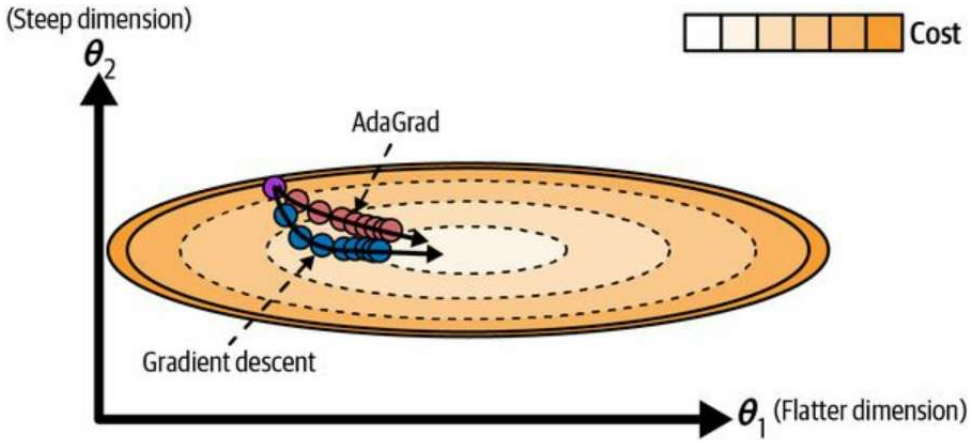

Figure 11-8. AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks: the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an Adagrad optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as linear regression, though). Still, understanding AdaGrad is helpful to comprehend the other adaptive learning rate optimizers.

RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm [18] fixes this by accumulating only the gradients from the most recent iterations, as opposed to all the gradients since the beginning of training. It does so by using exponential decay in the first step (see Equation 11-8).

Equation 11-8. RMSProp algorithm

$$1.\ s \leftarrow \rho\, s + (1-\rho)\, \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$
$$2.\ \theta \leftarrow \theta - \eta\, \nabla_\theta J(\theta) \oslash \sqrt{s + \varepsilon}$$

The decay rate ρ is typically set to 0.9. [19] Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an RMSprop optimizer:

```
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam

Adam, [20] which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-9). These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

Equation 11-9. Adam algorithm

$$1.\ \mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1-\beta_1)\nabla_\theta J(\theta) \quad 2.\ \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$
$$3.\ \hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t} \quad 4.\ \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\beta_2^t} \quad 5.\ \theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$$

In this equation, *t* represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both momentum optimization and RMSProp: β_1 corresponds to β in momentum optimization, and β_2 corresponds to ρ in RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ε is usually initialized to a tiny number such as 10^{-7}. These are the default values for the Adam class. Here is how to create an Adam optimizer using Keras:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
                                     beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than gradient descent.

> **TIP**
>
> If you are starting to feel overwhelmed by all these different techniques and are wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, three variants of Adam are worth mentioning: AdaMax, Nadam, and AdamW.

AdaMax

The Adam paper also introduced AdaMax. Notice that in step 2 of Equation 11-9, Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent gradients). In step 5, if we ignore ε and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares).

AdaMax replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in Equation 11-9 with $s \leftarrow \max(\beta 2s, \text{abs}(\nabla\theta J(\theta)))$, it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is the max of the absolute value of the time-decayed gradients.

In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

Nadam

Nadam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report introducing this technique, [21] the researcher Timothy Dozat compares many different optimizers on various tasks and finds that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

AdamW

AdamW[22] is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99. This may remind you of ℓ_2 regularization (introduced in Chapter 4), which also aims to keep the weights small, and indeed it can be shown mathematically that ℓ_2 regularization is equivalent to weight decay when using SGD. However, when using Adam or its variants, ℓ_2 regularization and weight decay are *not* equivalent: in practice, combining Adam with ℓ_2 regularization results in models that often don't generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.

> **WARNING**
>
> Adaptive optimization methods (including RMSProp, Adam, AdaMax, Nadam, and AdamW optimization) are often great, converging fast to a good solution. However, a 2017 paper[23] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using NAG instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, because it's moving fast.

To use Nadam, AdaMax, or AdamW in Keras, replace tf.keras.optimizers.Adam with tf.keras.optimizers.Nadam, tf.keras.optimizers.Adamax, or tf.keras.optimizers.experimental.AdamW. For AdamW, you probably want to tune the weight_decay hyperparameter.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed

to just *n* Jacobians per output. Since DNNs typically have tens of thousands of parameters or more, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

TRAINING SPARSE MODELS

All the optimization algorithms we just discussed produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero). However, this will typically not lead to a very sparse model, and it may degrade the model's performance.

A better option is to apply strong ℓ_1 regularization during training (you'll see how later in this chapter), as it pushes the optimizer to zero out as many weights as it can (as discussed in "Lasso Regression").

If these techniques remain insufficient, check out the TensorFlow Model Optimization Toolkit (TF-MOT), which provides a pruning API capable of iteratively removing connections during training based on their magnitude.

Table 11-2 compares all the optimizers we've discussed so far (* is bad, ** is average, and *** is good).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)

RMSprop	***	** or ***
Adam	***	** or ***
AdaMax	***	** or ***
Nadam	***	** or ***
AdamW	***	** or ***

Learning Rate Scheduling

Finding a good learning rate is very important. If you set it much too high, training may diverge (as discussed in "Gradient Descent"). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum and never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-9).

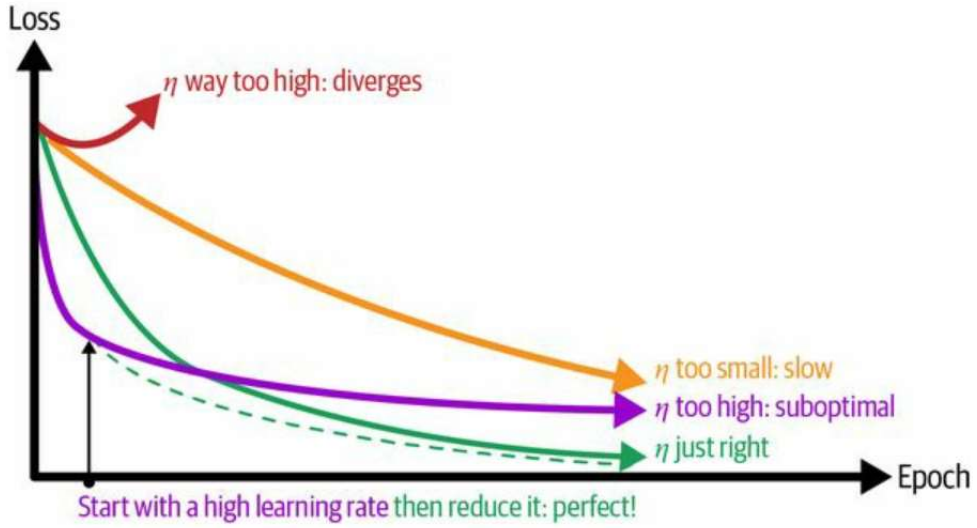

Figure 11-9. Learning curves for various learning rates η

As discussed in Chapter 10, you can find a good learning rate by training the model for a few hundred iterations, exponentially increasing the learning rate from a very small value to a very large value, and then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up. You can then reinitialize your model and train it with that learning rate.

But you can do better than a constant learning rate: if you start with a large

learning rate and then reduce it once training stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules* (I briefly introduced this concept in Chapter 4). These are the most commonly used learning schedules:

Power scheduling

Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0 / (1 + t/s)^c$. The initial learning rate η_0, the power c (typically set to 1), and the steps s are hyperparameters. The learning rate drops at each step. After s steps, the learning rate is down to $\eta_0 / 2$. After s more steps it is down to $\eta_0 / 3$, then it goes down to $\eta_0 / 4$, then $\eta_0 / 5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, power scheduling requires tuning η_0 and s (and possibly c).

Exponential scheduling

Set the learning rate to $\eta(t) = \eta_0 \, 0.1^{t/s}$. The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every s steps.

Piecewise constant scheduling

Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fiddling around to figure out the right sequence of learning rates and how long to use each of them.

Performance scheduling

Measure the validation error every N steps (just like for early stopping), and reduce the learning rate by a factor of λ when the error stops dropping.

1cycle scheduling

> 1cycle was introduced in a 2018 paper by Leslie Smith. [24] Contrary to the other approaches, it starts by increasing the initial learning rate η_0, growing linearly up to η_1 halfway through training. Then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate η_1 is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate η_0 is usually 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, compared to 90.3% accuracy in 800 epochs through a standard approach (with the same neural network architecture). This feat was dubbed *super-convergence*.

A 2013 paper by Andrew Senior et al. [25] compared the performance of some of the most popular learning schedules when using momentum optimization to train deep neural networks for speech recognition. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution. They also mentioned that it was easier to implement than performance scheduling, but in Keras both options are easy. That said, the 1cycle approach seems to perform even better.

Implementing power scheduling in Keras is the easiest option—just set the decay hyperparameter when creating an optimizer:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, decay=1e-4)
```

The decay is the inverse of *s* (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that *c* is equal to 1.

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1 ** (epoch / 20)
```

If you do not want to hardcode η_0 and *s*, you can create a function that returns a configured function:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 ** (epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Next, create a LearningRateScheduler callback, giving it the schedule function, and pass this callback to the fit() method:

```
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```

The LearningRateScheduler will update the optimizer's learning_rate attribute at the beginning of each epoch. Updating the learning rate once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you can always write your own callback (see the "Exponential Scheduling" section of this chapter's notebook for an example). Updating the learning rate at every step may help if there are many steps per epoch. Alternatively, you can use the tf.keras.optimiz ers.schedules approach, described shortly.

> **TIP**
>
> After training, history.history["lr"] gives you access to the list of learning rates used during training.

The schedule function can optionally take the current learning rate as a second argument. For example, the following schedule function multiplies the previous learning rate by $0.1^{1/20}$, which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1 ** (1 / 20)
```

This implementation relies on the optimizer's initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. Things are not so simple if your schedule function uses the epoch argument, however: the epoch does not get saved, and it gets reset to 0 every time you call the fit() method. If you were to continue training a model where it left off, this could lead to a very large learning rate, which would likely damage your model's weights. One solution is to manually set the fit() method's initial_epoch argument so the epoch starts at the right value.

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want; see the "Piecewise Constant Scheduling" section of the notebook for an example), then create a LearningRateScheduler callback with this function and pass it to the fit() method, just like for exponential scheduling:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
```

```
    else:
        return 0.001
```

For performance scheduling, use the ReduceLROnPlateau callback. For example, if you pass the following callback to the fit() method, it will multiply the learning rate by 0.5 whenever the best validation loss does not improve for five consecutive epochs (other options are available; please check the documentation for more details):

```
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```

Lastly, Keras offers an alternative way to implement learning rate scheduling: you can define a scheduled learning rate using one of the classes available in tf.keras.opti mizers.schedules, then pass it to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as the exponential_decay_fn() function we defined earlier:

```
import math

batch_size = 32
n_epochs = 25
n_steps = n_epochs * math.ceil(len(X_train) / batch_size)
scheduled_learning_rate = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.01, decay_steps=n_steps, decay_rate=0.1)
optimizer = tf.keras.optimizers.SGD(learning_rate=scheduled_learning_rate)
```

This is nice and simple, plus when you save the model, the learning rate and its schedule (including its state) get saved as well.

As for 1cycle, Keras does not support it, but it’s possible to implement it in less than 30 lines of code by creating a custom callback that modifies the learning rate at each iteration. To update the optimizer’s learning rate from within the callback’s on_batch_begin() method, you need to call tf.keras. back end.set_value(self.model.optimizer.learning_rate, new_learning_rate). See the “1Cycle Scheduling” section of the notebook for an example.

To sum up, exponential decay, performance scheduling, and 1cycle can

considerably speed up convergence, so give them a try!

Avoiding Overfitting Through Regularization

> *With four parameters I can fit an elephant and with five I can make him wiggle his trunk.*
>
> —John von Neumann, cited by Enrico Fermi in *Nature* 427

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

We already implemented one of the best regularization techniques in Chapter 10: early stopping. Moreover, even though batch normalization was designed to solve the unstable gradients problems, it also acts like a pretty good regularizer. In this section we will examine other popular regularization techniques for neural networks: ℓ_1 and ℓ_2 regularization, dropout, and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_2 regularization to constrain a neural network's connection weights, and/or ℓ_1 regularization if you want a sparse model (with many weights equal to 0). Here is how to apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

```
layer = tf.keras.layers.Dense(100, activation="relu",
                              kernel_initializer="he_normal",
                              kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

The l2() function returns a regularizer that will be called at each step during training to compute the regularization loss. This is then added to the final loss. As you might expect, you can just use tf.keras.regularizers.l1() if you want ℓ_1 regularization; if you want both ℓ_1 and ℓ_2 regularization, use tf.keras.regularizers.l1_l2() (specifying both regularization factors).

Since you will typically want to apply the same regularizer to all layers in your network, as well as using the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments. This makes the code ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python's functools.partial() function, which lets you create a thin wrapper for any callable, with some default argument values:

```
from functools import partial

RegularizedDense = partial(tf.keras.layers.Dense,
                           activation="relu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=tf.keras.regularizers.l2(0.01))

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(100),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
```

])

WARNING

As we saw earlier, ℓ_2 regularization is fine when using SGD, momentum optimization, and Nesterov momentum optimization, but not with Adam and its variants. If you want to use Adam with weight decay, then do not use ℓ_2 regularization: use AdamW instead.

Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper[26] by Geoffrey Hinton et al. in 2012 and further detailed in a 2014 paper[27] by Nitish Srivastava et al., and it has proven to be highly successful: many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability *p* of being temporarily "dropped out", meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-10). The hyperparameter *p* is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20%–30% in recurrent neural nets (see Chapter 15), and closer to 40%–50% in convolutional neural networks (see Chapter 14). After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay

attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.

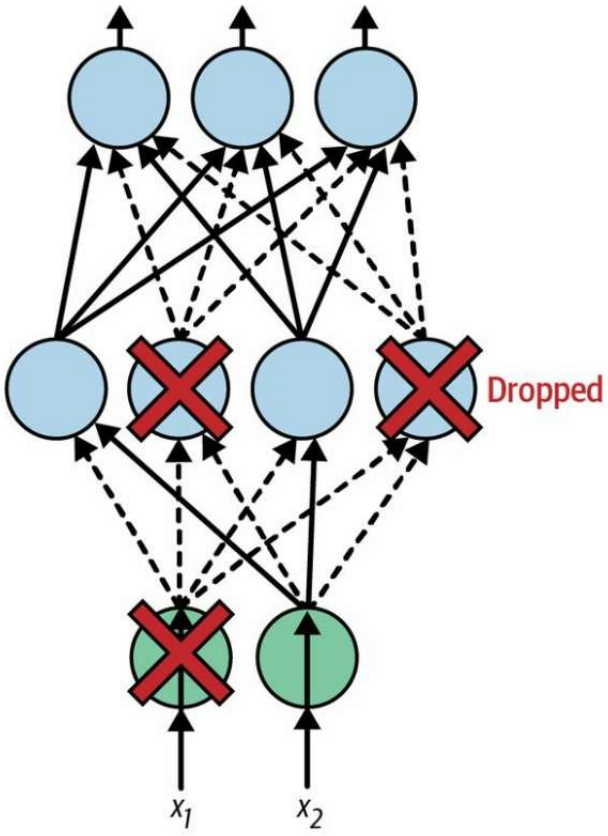

Figure 11-10. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are "dropped out"; these neurons output 0 at this iteration (represented by the dashed arrows)

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there are a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks, each with just one training instance. These neural

networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

> **TIP**
>
> In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).

There is one small but important technical detail. Suppose $p = 75\%$: on average only 25% of all neurons are active at each step during training. This means that after training, a neuron would be connected to four times as many input neurons as it would be during training. To compensate for this fact, we need to multiply each neuron's input connection weights by four during training. If we don't, the neural network will not perform well as it will see different data during and after training. More generally, we need to divide the connection weights by the *keep probability* $(1 – p)$ during training.

To implement dropout using Keras, you can use the tf.keras.layers.Dropout layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every dense layer, using a dropout rate of 0.2:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation="softmax")
])
[...]  # compile and train the model
```

> **WARNING**
>
> Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So, make sure to evaluate the training loss without dropout (e.g., after training).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly. So, it is generally well worth the extra time and effort, especially for large models.

> **TIP**
>
> If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: this is a variant of dropout that preserves the mean and standard deviation of its inputs. It was introduced in the same paper as SELU, as regular dropout would break self-normalization.

Monte Carlo (MC) Dropout

In 2016, a paper[28] by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing Dropout layers) and approximate Bayesian inference, [29] giving dropout a solid mathematical justification.
- Second, the authors introduced a powerful technique called *MC dropout*, which can boost the performance of any trained dropout model without having to retrain it or even modify it at all. It also provides a much better measure of the model's uncertainty, and it can be implemented in just a few lines of code.

If this all sounds like some "one weird trick" clickbait, then take a look at the following code. It is the full implementation of MC dropout, boosting the dropout model we trained earlier without retraining it:

```
import numpy as np

y_probas = np.stack([model(X_test, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Note that model(X) is similar to model.predict(X) except it returns a tensor rather than a NumPy array, and it supports the training argument. In this code example, setting training=True ensures that the Dropout layer remains active, so all predictions will be a bit different. We just make 100 predictions over the test set, and we compute their average. More specifically, each call to the model returns a matrix with one row per instance and one column per class. Because there are 10,000 instances in the test set and 10 classes, this is a matrix of shape [10000, 10]. We stack 100 such matrices, so y_probas is a 3D array of shape [100, 10000, 10]. Once we average over the first dimension (axis=0) we get y_proba, an array of shape [10000, 10], like we would get

with a single prediction. That's all! Averaging over multiple predictions with dropout turned on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout turned off. For example, let's look at the model's prediction for the first instance in the Fashion MNIST test set, with dropout turned off:

```
>>> model.predict(X_test[:1]).round(3)
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.024, 0.   , 0.132, 0.   ,
        0.844]], dtype=float32)
```

The model is fairly confident (84.4%) that this image belongs to class 9 (ankle boot). Compare this with the MC dropout prediction:

```
>>> y_proba[0].round(3)
array([0.   , 0.   , 0.   , 0.   , 0.   , 0.067, 0.   , 0.209, 0.001,
       0.723], dtype=float32)
```

The model still seems to prefer class 9, but its confidence dropped down to 72.3%, and the estimated probabilities for classes 5 (sandal) and 7 (sneaker) have increased, which makes sense given they're also footwear.

MC dropout tends to improve the reliability of the model's probability estimates. This means that it's less likely to be confident but wrong, which can be dangerous: just imagine a self-driving car confidently ignoring a stop sign. It's also useful to know exactly which other classes are most likely. Additionally, you can take a look at the standard deviation of the probability estimates:

```
>>> y_std = y_probas.std(axis=0)
>>> y_std[0].round(3)
array([0.   , 0.   , 0.   , 0.001, 0.   , 0.096, 0.   , 0.162, 0.001,
       0.183], dtype=float32)
```

Apparently there's quite a lot of variance in the probability estimates for class 9: the standard deviation is 0.183, which should be compared to the estimated probability of 0.723: if you were building a risk-sensitive system (e.g., a medical or financial system), you would probably treat such an uncertain

prediction with extreme caution. You would definitely not treat it like an 84.4% confident prediction. The model's accuracy also got a (very) small boost from 87.0% to 87.2%:

```
>>> y_pred = y_proba.argmax(axis=1)
>>> accuracy = (y_pred == y_test).sum() / len(y_test)
>>> accuracy
0.8717
```

> **NOTE**
>
> The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates will be. However, if you double it, inference time will also be doubled. Moreover, above a certain number of samples, you will notice little improvement. Your job is to find the right trade-off between latency and accuracy, depending on your application.

If your model contains other layers that behave in a special way during training (such as BatchNormalization layers), then you should not force training mode like we just did. Instead, you should replace the Dropout layers with the following MCDropout class: [30]

```
class MCDropout(tf.keras.layers.Dropout):
    def call(self, inputs, training=False):
        return super().call(inputs, training=True)
```

Here, we just subclass the Dropout layer and override the call() method to force its training argument to True (see Chapter 12). Similarly, you could define an MCAlphaDropout class by subclassing AlphaDropout instead. If you are creating a model from scratch, it's just a matter of using MCDropout rather than Dropout. But if you have a model that was already trained using Dropout, you need to create a new model that's identical to the existing model except with Dropout instead of MCDropout, then copy the existing model's weights to your new model.

In short, MC dropout is a great technique that boosts dropout models and

provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

Max-Norm Regularization

Another popular regularization technique for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

Max-norm regularization does not add a regularization loss term to the overall loss function. Instead, it is typically implemented by computing $\| \mathbf{w} \|_2$ after each training step and rescaling $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w} r / \| \mathbf{w} \|_2$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using batch normalization).

To implement max-norm regularization in Keras, set the kernel_constraint argument of each hidden layer to a max_norm() constraint with the appropriate max value, like this:

```
dense = tf.keras.layers.Dense(
    100, activation="relu", kernel_initializer="he_normal",
    kernel_constraint=tf.keras.constraints.max_norm(1.))
```

After each training iteration, the model's fit() method will call the object returned by max_norm(), passing it the layer's weights and getting rescaled weights in return, which then replace the layer's weights. As you'll see in Chapter 12, you can define your own custom constraint function if necessary and use it as the kernel_con straint. You can also constrain the bias terms by setting the bias_constraint argument.

The max_norm() function has an axis argument that defaults to 0. A Dense layer usually has weights of shape [*number of inputs*, *number of neurons*], so using axis=0 means that the max-norm constraint will apply independently to each neuron's weight vector. If you want to use max-norm with convolutional layers (see Chapter 14), make sure to set the max_norm() constraint's axis argument appropriately (usually axis=[0, 1, 2]).

Summary and Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in Table 11-3 to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

If the network is a simple stack of dense layers, then it can self-normalize, and you should use the configuration in Table 11-4 instead.

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed

Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1cycle

Don’t forget to normalize the input features! You should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, here are some exceptions:

- If you need a sparse model, you can use ℓ_1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can use the TensorFlow Model Optimization Toolkit. This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers, use a fast activation function such as ReLU or leaky ReLU, and fold the batch normalization layers into the previous layers after training. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits (see “Deploying a Model to a Mobile or Embedded Device”). Again, check out TF-MOT.
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

With these guidelines, you are now ready to train very deep nets! I hope you are now convinced that you can go quite a long way using just the convenient Keras API. There may come a time, however, when you need to have even more control; for example, to write a custom loss function or to tweak the training algorithm. For such cases you will need to use TensorFlow’s lower-

level API, as you will see in the next chapter.

Exercises

1. What is the problem that Glorot initialization and He initialization aim to fix?
2. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
3. Is it OK to initialize the bias terms to 0?
4. In which cases would you want to use each of the activation functions we discussed in this chapter?
5. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
    a. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the Swish activation function.
    b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with tf.keras.datasets.cifar10.load_data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you'll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
    c. Now try adding batch normalization and compare the learning curves: is it converging faster than before? Does it produce a better

model? How does it affect training speed?

d. Try replacing batch normalization with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).

e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC dropout.

f. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

---

1 Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

2 Here's an analogy: if you set a microphone amplifier's knob too close to zero, people won't hear your voice, but if you set it too close to the max, your voice will be saturated and people won't understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

3 E.g., Kaiming He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

4 A dead neuron may come back to life if its inputs evolve over time and eventually return within a range where the ReLU activation function gets a positive input again. For example, this may happen if gradient descent tweaks the neurons in the layers below the dead neuron.

5 Bing Xu et al., "Empirical Evaluation of Rectified Activations in Convolutional Network," arXiv preprint arXiv:1505.00853 (2015).

6 Djork-Arné Clevert et al., "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *Proceedings of the International Conference on Learning Representations*, arXiv preprint (2015).

7 Günter Klambauer et al., "Self-Normalizing Neural Networks", *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

8 Dan Hendrycks and Kevin Gimpel, “Gaussian Error Linear Units (GELUs)”, arXiv preprint arXiv:1606.08415 (2016).

9 A function is convex if the line segment between any two points on the curve never lies below the curve. A monotonic function only increases, or only decreases.

10 Prajit Ramachandran et al., “Searching for Activation Functions”, arXiv preprint arXiv:1710.05941 (2017).

11 Diganta Misra, “Mish: A Self Regularized Non-Monotonic Activation Function”, arXiv preprint arXiv:1908.08681 (2019).

12 Sergey Ioffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

13 However, they are estimated during training based on the training data, so arguably they *are* trainable. In Keras, “non-trainable” really means “untouched by backpropagation”.

14 Razvan Pascanu et al., “On the Difficulty of Training Recurrent Neural Networks”, *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

15 Boris T. Polyak, “Some Methods of Speeding Up the Convergence of Iteration Methods”, *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.

16 Yurii Nesterov, “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$,” *Doklady AN USSR* 269 (1983): 543–547.

17 John Duchi et al., “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, *Journal of Machine Learning Research* 12 (2011): 2121–2159.

18 This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012 and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *https://homl.info/57*; video: *https://homl.info/58*). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite “slide 29 in lecture 6e” in their papers.

19 ρ is the Greek letter rho.

20 Diederik P. Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization”, arXiv preprint arXiv:1412.6980 (2014).

21 Timothy Dozat, “Incorporating Nesterov Momentum into Adam” (2016).

22 Ilya Loshchilov, and Frank Hutter, “Decoupled Weight Decay Regularization”, arXiv preprint arXiv:1711.05101 (2017).

23 Ashia C. Wilson et al., “The Marginal Value of Adaptive Gradient Methods in Machine Learning”, *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

24 Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay”, arXiv preprint arXiv:1803.09820 (2018).

25 Andrew Senior et al., “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition”, *Proceedings of the IEEE International Conference on Acoustics, Speech,*

and Signal Processing (2013): 6724–6728.

26 Geoffrey E. Hinton et al., “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors”, arXiv preprint arXiv:1207.0580 (2012).

27 Nitish Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research* 15 (2014): 1929–1958.

28 Yarin Gal and Zoubin Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

29 Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *deep Gaussian process*.

30 This MCDropout class will work with all Keras APIs, including the sequential API. If you only care about the functional API or the subclassing API, you do not have to create an MCDropout class; you can create a regular Dropout layer and call it with training=True.

Chapter 12. Custom Models and Training with TensorFlow

Up until now, we've used only TensorFlow's high-level API, Keras, but it already got us pretty far: we built various neural network architectures, including regression and classification nets, Wide & Deep nets, and self-normalizing nets, using all sorts of techniques, such as batch normalization, dropout, and learning rate schedules. In fact, 95% of the use cases you will encounter will not require anything other than Keras (and tf.data; see Chapter 13). But now it's time to dive deeper into TensorFlow and take a look at its lower-level Python API. This will be useful when you need extra control to write custom loss functions, custom metrics, layers, models, initializers, regularizers, weight constraints, and more. You may even need to fully control the training loop itself; for example, to apply special transformations or constraints to the gradients (beyond just clipping them) or to use multiple optimizers for different parts of the network. We will cover all these cases in this chapter, and we will also look at how you can boost your custom models and training algorithms using TensorFlow's automatic graph generation feature. But first, let's take a quick tour of TensorFlow.

A Quick Tour of TensorFlow

As you know, TensorFlow is a powerful library for numerical computation, particularly well suited and fine-tuned for large-scale machine learning (but you can use it for anything else that requires heavy computations). It was developed by the Google Brain team and it powers many of Google's large-scale services, such as Google Cloud Speech, Google Photos, and Google Search. It was open sourced in November 2015, and it is now the most widely used deep learning library in the industry:[1] countless projects use TensorFlow for all sorts of machine learning tasks, such as image classification, natural language processing, recommender systems, and time series forecasting.

So what does TensorFlow offer? Here's a summary:

- Its core is very similar to NumPy, but with GPU support.
- It supports distributed computing (across multiple devices and servers).
- It includes a kind of just-in-time (JIT) compiler that allows it to optimize computations for speed and memory usage. It works by extracting the *computation graph* from a Python function, optimizing it (e.g., by pruning unused nodes), and running it efficiently (e.g., by automatically running independent operations in parallel).
- Computation graphs can be exported to a portable format, so you can train a TensorFlow model in one environment (e.g., using Python on Linux) and run it in another (e.g., using Java on an Android device).
- It implements reverse-mode autodiff (see Chapter 10 and Appendix B) and provides some excellent optimizers, such as RMSProp and Nadam (see Chapter 11), so you can easily minimize all sorts of loss functions.

TensorFlow offers many more features built on top of these core features: the most important is of course Keras, [2] but it also has data loading and preprocessing ops (tf.data, tf.io, etc.), image processing ops (tf.image), signal

processing ops (tf.signal), and more (see Figure 12-1 for an overview of TensorFlow's Python API).

> **TIP**
>
> We will cover many of the packages and functions of the TensorFlow API, but it's impossible to cover them all, so you should really take some time to browse through the API; you will find that it is quite rich and well documented.

At the lowest level, each TensorFlow operation (*op* for short) is implemented using highly efficient C++ code. [3] Many operations have multiple implementations called *kernels*: each kernel is dedicated to a specific device type, such as CPUs, GPUs, or even TPUs (*tensor processing units*). As you may know, GPUs can dramatically speed up computations by splitting them into many smaller chunks and running them in parallel across many GPU threads. TPUs are even faster: they are custom ASIC chips built specifically for deep learning operations [4] (we will discuss how to use TensorFlow with GPUs or TPUs in Chapter 19).

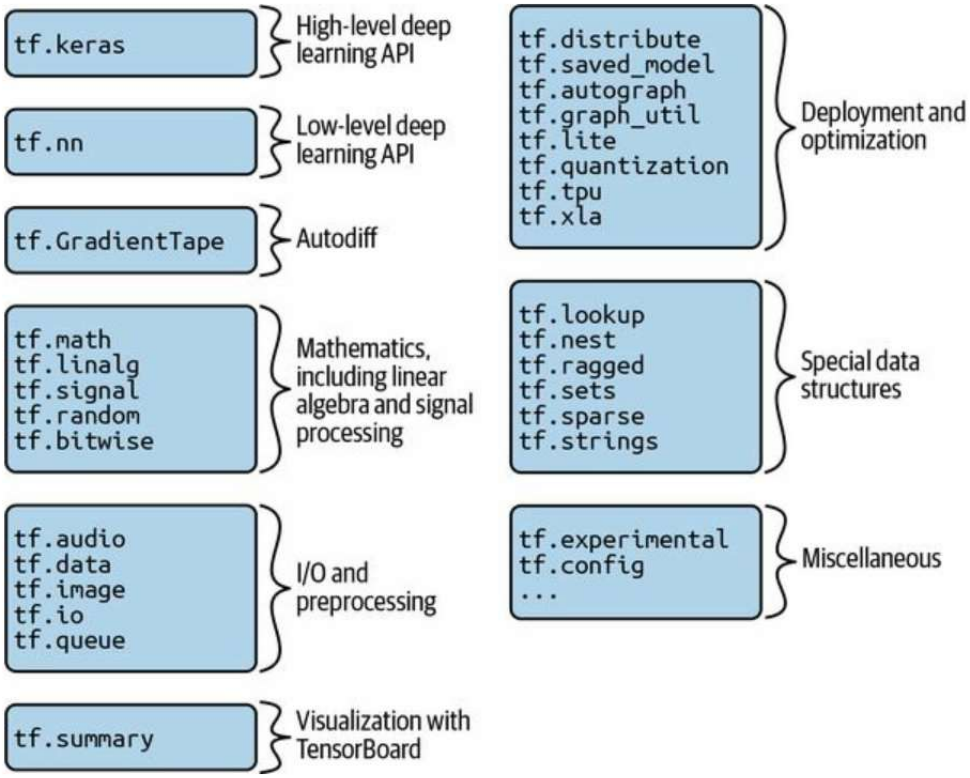

Figure 12-1. TensorFlow's Python API

TensorFlow's architecture is shown in Figure 12-2. Most of the time your code will use the high-level APIs (especially Keras and tf.data), but when you need more flexibility you will use the lower-level Python API, handling tensors directly. In any case, TensorFlow's execution engine will take care of running the operations efficiently, even across multiple devices and machines if you tell it to.

TensorFlow runs not only on Windows, Linux, and macOS, but also on mobile devices (using *TensorFlow Lite*), including both iOS and Android (see Chapter 19). Note that APIs for other languages are also available, if you do not want to use the Python API: there are C++, Java, and Swift APIs. There is even a JavaScript implementation called *TensorFlow.js* that makes it possible to run your models directly in your browser.

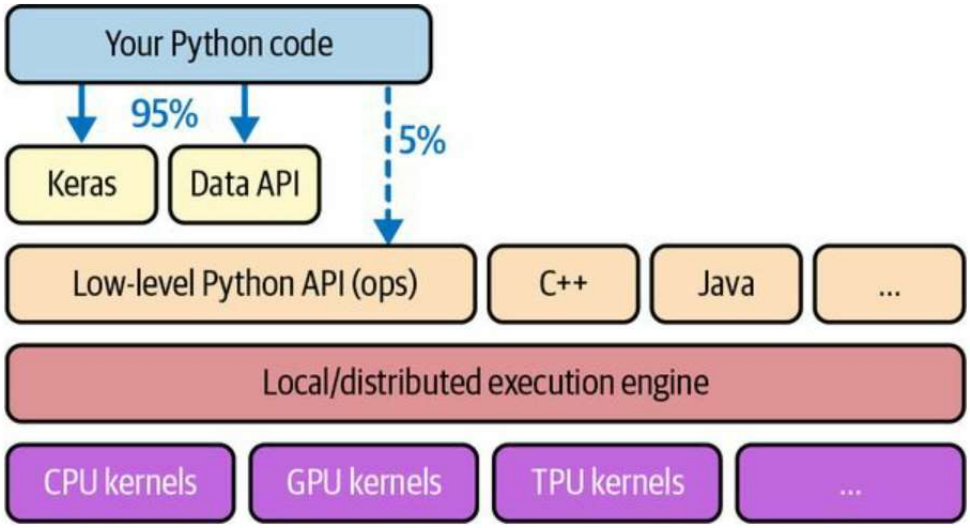

Figure 12-2. TensorFlow's architecture

There's more to TensorFlow than the library. TensorFlow is at the center of an extensive ecosystem of libraries. First, there's TensorBoard for visualization (see Chapter 10). Next, there's TensorFlow Extended (TFX), which is a set of libraries built by Google to productionize TensorFlow projects: it includes tools for data validation, preprocessing, model analysis, and serving (with TF Serving; see Chapter 19). Google's *TensorFlow Hub* provides a way to easily download and reuse pretrained neural networks. You can also get many neural network architectures, some of them pretrained, in TensorFlow's model garden. Check out the TensorFlow Resources and *https://github.com/jtoy/awesome-tensorflow* for more TensorFlow-based projects. You will find hundreds of TensorFlow projects on GitHub, so it is often easy to find existing code for whatever you are trying to do.

> **TIP**
>
> More and more ML papers are released along with their implementations, and sometimes even with pretrained models. Check out *https://paperswithcode.com* to easily find them.

Last but not least, TensorFlow has a dedicated team of passionate and helpful

developers, as well as a large community contributing to improving it. To ask technical questions, you should use *https://stackoverflow.com* and tag your question with *tensorflow* and *python*. You can file bugs and feature requests through GitHub. For general discussions, join the TensorFlow Forum.

OK, it’s time to start coding!

Using TensorFlow like NumPy

TensorFlow's API revolves around *tensors*, which flow from operation to operation—hence the name Tensor*Flow*. A tensor is very similar to a NumPy ndarray: it is usually a multidimensional array, but it can also hold a scalar (a simple value, such as 42). These tensors will be important when we create custom cost functions, custom metrics, custom layers, and more, so let's see how to create and manipulate them.

Tensors and Operations

You can create a tensor with tf.constant(). For example, here is a tensor representing a matrix with two rows and three columns of floats:

```
>>> import tensorflow as tf
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])  # matrix
>>> t
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Just like an ndarray, a tf.Tensor has a shape and a data type (dtype):

```
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indexing works much like in NumPy:

```
>>> t[:, 1:]
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Most importantly, all sorts of tensor operations are available:

```
>>> t + 10
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Note that writing t + 10 is equivalent to calling tf.add(t, 10) (indeed, Python calls the magic method t.__add__(10), which just calls tf.add(t, 10)). Other operators, like - and *, are also supported. The @ operator was added in Python 3.5, for matrix multiplication: it is equivalent to calling the tf.matmul() function.

> **NOTE**
>
> Many functions and classes have aliases. For example, tf.add() and tf.math.add() are the same function. This allows TensorFlow to have concise names for the most common operations [5] while preserving well-organized packages.

A tensor can also hold a scalar value. In this case, the shape is empty:

```
>>> tf.constant(42)
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

> **NOTE**
>
> The Keras API has its own low-level API, located in tf.keras.backend. This package is usually imported as K, for conciseness. It used to include functions like K.square(), K.exp(), and K.sqrt(), which you may run across in existing code: this was useful to write portable code back when Keras supported multiple backends, but now that Keras is TensorFlow-only, you should call TensorFlow's low-level API directly (e.g., tf.square() instead of K.square()). Technically K.square() and its friends are still there for backward compatibility, but the documentation of the tf.keras.backend package only lists a handful of utility functions, such as clear_session() (mentioned in Chapter 10).

You will find all the basic math operations you need (tf.add(), tf.multiply(), tf.square(), tf.exp(), tf.sqrt(), etc.) and most operations that you can find in NumPy (e.g., tf.reshape(), tf.squeeze(), tf.tile()). Some functions have a different name than in NumPy; for instance, tf.reduce_mean(),

tf.reduce_sum(), tf.reduce_max(), and tf.math.log() are the equivalent of np.mean(), np.sum(), np.max(), and np.log(). When the name differs, there is usually a good reason for it. For example, in TensorFlow you must write tf.transpose(t); you cannot just write t.T like in NumPy. The reason is that the tf.transpose() function does not do exactly the same thing as NumPy's T attribute: in TensorFlow, a new tensor is created with its own copy of the transposed data, while in NumPy, t.T is just a transposed view on the same data. Similarly, the tf.reduce_sum() operation is named this way because its GPU kernel (i.e., GPU implementation) uses a reduce algorithm that does not guarantee the order in which the elements are added: because 32-bit floats have limited precision, the result may change ever so slightly every time you call this operation. The same is true of tf.reduce_mean() (but of course tf.reduce_max() is deterministic).

Tensors and NumPy

Tensors play nice with NumPy: you can create a tensor from a NumPy array, and vice versa. You can even apply TensorFlow operations to NumPy arrays and NumPy operations to tensors:

```
>>> import numpy as np
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy()  # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

WARNING

Notice that NumPy uses 64-bit precision by default, while TensorFlow uses 32-bit. This is because 32-bit precision is generally more than enough for neural networks, plus it runs faster and uses less RAM. So when you create a tensor from a NumPy array, make sure to set dtype=tf.float32.

Type Conversions

Type conversions can significantly hurt performance, and they can easily go unnoticed when they are done automatically. To avoid this, TensorFlow does not perform any type conversions automatically: it just raises an exception if you try to execute an operation on tensors with incompatible types. For example, you cannot add a float tensor and an integer tensor, and you cannot even add a 32-bit float and a 64-bit float:

```
>>> tf.constant(2.) + tf.constant(40)
[...] InvalidArgumentError: [...] expected to be a float tensor [...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
[...] InvalidArgumentError: [...] expected to be a float tensor [...]
```

This may be a bit annoying at first, but remember that it’s for a good cause! And of course you can use tf.cast() when you really need to convert types:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

Variables

The tf.Tensor values we've seen so far are immutable: we cannot modify them. This means that we cannot use regular tensors to implement weights in a neural network, since they need to be tweaked by backpropagation. Plus, other parameters may also need to change over time (e.g., a momentum optimizer keeps track of past gradients). What we need is a tf.Variable:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

A tf.Variable acts much like a tf.Tensor: you can perform the same operations with it, it plays nicely with NumPy as well, and it is just as picky with types. But it can also be modified in place using the assign() method (or assign_add() or assign_sub(), which increment or decrement the variable by the given value). You can also modify individual cells (or slices), by using the cell's (or slice's) assign() method or by using the scatter_update() or scatter_nd_update() methods:

```
v.assign(2 * v)          # v now equals [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)       # v now equals [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.]) # v now equals [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(     # v now equals [[100., 42., 0.], [8., 10., 200.]]
  indices=[[0, 0], [1, 2]], updates=[100., 200.])
```

Direct assignment will not work:

```
>>> v[1] = [7., 8., 9.]
[...] TypeError: 'ResourceVariable' object does not support item assignment
```

> **NOTE**
>
> In practice you will rarely have to create variables manually; Keras provides an add_weight() method that will take care of it for you, as you'll see. Moreover, model

parameters will generally be updated directly by the optimizers, so you will rarely need to update variables manually.

Other Data Structures

TensorFlow supports several other data structures, including the following (see the “Other Data Structures” section in this chapter’s notebook or Appendix C for more details):

Sparse tensors (tf.SparseTensor)

Efficiently represent tensors containing mostly zeros. The tf.sparse package contains operations for sparse tensors.

Tensor arrays (tf.TensorArray)

Are lists of tensors. They have a fixed length by default but can optionally be made extensible. All tensors they contain must have the same shape and data type.

Ragged tensors (tf.RaggedTensor)

Represent lists of tensors, all of the same rank and data type, but with varying sizes. The dimensions along which the tensor sizes vary are called the *ragged dimensions*. The tf.ragged package contains operations for ragged tensors.

String tensors

Are regular tensors of type tf.string. These represent byte strings, not Unicode strings, so if you create a string tensor using a Unicode string (e.g., a regular Python 3 string like "café"), then it will get encoded to UTF-8 automatically (e.g., b"caf\xc3\xa9"). Alternatively, you can represent Unicode strings using tensors of type tf.int32, where each item represents a Unicode code point (e.g., [99, 97, 102, 233]). The tf.strings package (with an s) contains ops for byte strings and Unicode strings (and to convert one into the other). It’s important to note that a tf.string is atomic, meaning that its length does not appear in the tensor’s shape. Once you convert it to a Unicode tensor (i.e., a tensor of type tf.int32 holding Unicode code points), the length appears in the shape.

Sets

Are represented as regular tensors (or sparse tensors). For example, tf.constant([[1, 2], [3, 4]]) represents the two sets {1, 2} and {3, 4}. More generally, each set is represented by a vector in the tensor's last axis. You can manipulate sets using operations from the tf.sets package.

Queues

Store tensors across multiple steps. TensorFlow offers various kinds of queues: basic first-in, first-out (FIFO) queues (FIFOQueue), plus queues that can prioritize some items (PriorityQueue), shuffle their items (RandomShuffleQueue), and batch items of different shapes by padding (PaddingFIFOQueue). These classes are all in the tf.queue package.

With tensors, operations, variables, and various data structures at your disposal, you are now ready to customize your models and training algorithms!

Customizing Models and Training Algorithms

You'll start by creating a custom loss function, which is a straightforward and common use case.

Custom Loss Functions

Suppose you want to train a regression model, but your training set is a bit noisy. Of course, you start by trying to clean up your dataset by removing or fixing the outliers, but that turns out to be insufficient; the dataset is still noisy. Which loss function should you use? The mean squared error might penalize large errors too much and cause your model to be imprecise. The mean absolute error would not penalize outliers as much, but training might take a while to converge, and the trained model might not be very precise. This is probably a good time to use the Huber loss (introduced in Chapter 10) instead of the good old MSE. The Huber loss is available in Keras (just use an instance of the tf.keras.losses.Huber class), but let's pretend it's not there. To implement it, just create a function that takes the labels and the model's predictions as arguments, and uses TensorFlow operations to compute a tensor containing all the losses (one per sample):

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss  = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

> **WARNING**
>
> For better performance, you should use a vectorized implementation, as in this example. Moreover, if you want to benefit from TensorFlow's graph optimization features, you should use only TensorFlow operations.

It is also possible to return the mean loss instead of the individual sample losses, but this is not recommended as it makes it impossible to use class weights or sample weights when you need them (see Chapter 10).

Now you can use this Huber loss function when you compile the Keras model, then train your model as usual:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

And that’s it! For each batch during training, Keras will call the huber_fn() function to compute the loss, then it will use reverse-mode autodiff to compute the gradients of the loss with regard to all the model parameters, and finally it will perform a gradient descent step (in this example using a Nadam optimizer). Moreover, it will keep track of the total loss since the beginning of the epoch, and it will display the mean loss.

But what happens to this custom loss when you save the model?

Saving and Loading Models That Contain Custom Components

Saving a model containing a custom loss function works fine, but when you load it, you'll need to provide a dictionary that maps the function name to the actual function. More generally, when you load a model containing custom objects, you need to map the names to the objects:

```
model = tf.keras.models.load_model("my_model_with_a_custom_loss",
                                   custom_objects={"huber_fn": huber_fn})
```

> **TIP**
>
> If you decorate the huber_fn() function with @keras.utils.reg ister_keras_serializable(), it will automatically be available to the load_model() function: there's no need to include it in the custom_objects dictionary.

With the current implementation, any error between –1 and 1 is considered "small". But what if you want a different threshold? One solution is to create a function that creates a configured loss function:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = threshold * tf.abs(error) - threshold ** 2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Unfortunately, when you save the model, the threshold will not be saved. This means that you will have to specify the threshold value when loading the model (note that the name to use is "huber_fn", which is the name of the function you gave Keras, not the name of the function that created it):

```
model = tf.keras.models.load_model(
    "my_model_with_a_custom_loss_threshold_2",
    custom_objects={"huber_fn": create_huber(2.0)}
)
```

You can solve this by creating a subclass of the tf.keras.losses.Loss class, and then implementing its get_config() method:

```
class HuberLoss(tf.keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)

    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let’s walk through this code:

- The constructor accepts **kwargs and passes them to the parent constructor, which handles standard hyperparameters: the name of the loss and the reduction algorithm to use to aggregate the individual instance losses. By default this is "AUTO", which is equivalent to "SUM_OVER_BATCH_SIZE": the loss will be the sum of the instance losses, weighted by the sample weights, if any, and divided by the batch size (not by the sum of weights, so this is *not* the weighted mean). [6] Other possible values are "SUM" and "NONE".
- The call() method takes the labels and predictions, computes all the instance losses, and returns them.
- The get_config() method returns a dictionary mapping each hyperparameter name to its value. It first calls the parent class’s get_config() method, then adds the new hyperparameters to this

dictionary. [7]

You can then use any instance of this class when you compile the model:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

When you save the model, the threshold will be saved along with it; and when you load the model, you just need to map the class name to the class itself:

```
model = tf.keras.models.load_model("my_model_with_a_custom_loss_class",
                                   custom_objects={"HuberLoss": HuberLoss})
```

When you save a model, Keras calls the loss instance's get_config() method and saves the config in the SavedModel format. When you load the model, it calls the from_config() class method on the HuberLoss class: this method is implemented by the base class (Loss) and creates an instance of the class, passing **config to the constructor.

That's it for losses! As you'll see now, custom activation functions, initializers, regularizers, and constraints are not much different.

Custom Activation Functions, Initializers, Regularizers, and Constraints

Most Keras functionalities, such as losses, regularizers, constraints, initializers, metrics, activation functions, layers, and even full models, can be customized in much the same way. Most of the time, you will just need to write a simple function with the appropriate inputs and outputs. Here are examples of a custom activation function (equivalent to tf.keras.activations.softplus() or tf.nn.softplus()), a custom Glorot initializer (equivalent to tf.keras.initializers.glorot_normal()), a custom ℓ_1 regularizer (equivalent to tf.keras.regularizers.l1(0.01)), and a custom constraint that ensures weights are all positive (equivalent to tf.keras.con straints.nonneg() or tf.nn.relu()):

```
def my_softplus(z):
    return tf.math.log(1.0 + tf.exp(z))

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights):  # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

As you can see, the arguments depend on the type of custom function. These custom functions can then be used normally, as shown here:

```
layer = tf.keras.layers.Dense(1, activation=my_softplus,
                              kernel_initializer=my_glorot_initializer,
                              kernel_regularizer=my_l1_regularizer,
                              kernel_constraint=my_positive_weights)
```

The activation function will be applied to the output of this Dense layer, and its result will be passed on to the next layer. The layer's weights will be initialized using the value returned by the initializer. At each training step the

weights will be passed to the regularization function to compute the regularization loss, which will be added to the main loss to get the final loss used for training. Finally, the constraint function will be called after each training step, and the layer's weights will be replaced by the constrained weights.

If a function has hyperparameters that need to be saved along with the model, then you will want to subclass the appropriate class, such as tf.keras.regu larizers.Reg ularizer, tf.keras.constraints.Constraint, tf.keras.initializers.Ini tializer, or tf.keras.layers.Layer (for any layer, including activation functions). Much as you did for the custom loss, here is a simple class for ℓ_1 regularization that saves its factor hyperparameter (this time you do not need to call the parent constructor or the get_config() method, as they are not defined by the parent class):

```
class MyL1Regularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))

    def get_config(self):
        return {"factor": self.factor}
```

Note that you must implement the call() method for losses, layers (including activation functions), and models, or the __call__() method for regularizers, initializers, and constraints. For metrics, things are a bit different, as you will see now.

Custom Metrics

Losses and metrics are conceptually not the same thing: losses (e.g., cross entropy) are used by gradient descent to *train* a model, so they must be differentiable (at least at the points where they are evaluated), and their gradients should not be zero everywhere. Plus, it's OK if they are not easily interpretable by humans. In contrast, metrics (e.g., accuracy) are used to *evaluate* a model: they must be more easily interpretable, and they can be nondifferentiable or have zero gradients everywhere.

That said, in most cases, defining a custom metric function is exactly the same as defining a custom loss function. In fact, we could even use the Huber loss function we created earlier as a metric; [8] it would work just fine (and persistence would also work the same way, in this case only saving the name of the function, "huber_fn", not the threshold):

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch. Most of the time, this is exactly what you want. But not always! Consider a binary classifier's precision, for example. As you saw in Chapter 3, precision is the number of true positives divided by the number of positive predictions (including both true positives and false positives). Suppose the model made five positive predictions in the first batch, four of which were correct: that's 80% precision. Then suppose the model made three positive predictions in the second batch, but they were all incorrect: that's 0% precision for the second batch. If you just compute the mean of these two precisions, you get 40%. But wait a second—that's *not* the model's precision over these two batches! Indeed, there were a total of four true positives (4 + 0) out of eight positive predictions (5 + 3), so the overall precision is 50%, not 40%. What we need is an object that can keep track of the number of true positives and the number of false positives and that can compute the precision based on these numbers when requested. This is precisely what the tf.keras.metrics.Precision class does:

```
>>> precision = tf.keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
```

In this example, we created a Precision object, then we used it like a function, passing it the labels and predictions for the first batch, then for the second batch (you can optionally pass sample weights as well, if you want). We used the same number of true and false positives as in the example we just discussed. After the first batch, it returns a precision of 80%; then after the second batch, it returns 50% (which is the overall precision so far, not the second batch's precision). This is called a *streaming metric* (or *stateful metric*), as it is gradually updated, batch after batch.

At any point, we can call the result() method to get the current value of the metric. We can also look at its variables (tracking the number of true and false positives) by using the variables attribute, and we can reset these variables using the reset_states() method:

```
>>> precision.result()
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...], numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...], numpy=array([4.], dtype=float32)>]
>>> precision.reset_states()  # both variables get reset to 0.0
```

If you need to define your own custom streaming metric, create a subclass of the tf.keras.metrics.Metric class. Here is a basic example that keeps track of the total Huber loss and the number of instances seen so far. When asked for the result, it returns the ratio, which is just the mean Huber loss:

```
class HuberMetric(tf.keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)  # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
```

```
    def update_state(self, y_true, y_pred, sample_weight=None):
        sample_metrics = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(sample_metrics))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's walk through this code: [9]

- The constructor uses the add_weight() method to create the variables needed to keep track of the metric's state over multiple batches—in this case, the sum of all Huber losses (total) and the number of instances seen so far (count). You could just create variables manually if you preferred. Keras tracks any tf.Variable that is set as an attribute (and more generally, any "trackable" object, such as layers or models).
- The update_state() method is called when you use an instance of this class as a function (as we did with the Precision object). It updates the variables, given the labels and predictions for one batch (and sample weights, but in this case we ignore them).
- The result() method computes and returns the final result, in this case the mean Huber metric over all instances. When you use the metric as a function, the update_state() method gets called first, then the result() method is called, and its output is returned.
- We also implement the get_config() method to ensure the threshold gets saved along with the model.
- The default implementation of the reset_states() method resets all variables to 0.0 (but you can override it if needed).

> **NOTE**
>
> Keras will take care of variable persistence seamlessly; no action is required.

When you define a metric using a simple function, Keras automatically calls it for each batch, and it keeps track of the mean during each epoch, just like we did manually. So the only benefit of our HuberMetric class is that the threshold will be saved. But of course, some metrics, like precision, cannot simply be averaged over batches: in those cases, there's no other option than to implement a streaming metric.

Now that you've built a streaming metric, building a custom layer will seem like a walk in the park!

Custom Layers

You may occasionally want to build an architecture that contains an exotic layer for which TensorFlow does not provide a default implementation. Or you may simply want to build a very repetitive architecture, in which a particular block of layers is repeated many times, and it would be convenient to treat each block as a single layer. For such cases, you'll want to build a custom layer.

There are some layers that have no weights, such as tf.keras.layers.Flatten or tf.keras.layers.ReLU. If you want to create a custom layer without any weights, the simplest option is to write a function and wrap it in a tf.keras.layers.Lambda layer. For example, the following layer will apply the exponential function to its inputs:

```
exponential_layer = tf.keras.layers.Lambda(lambda x: tf.exp(x))
```

This custom layer can then be used like any other layer, using the sequential API, the functional API, or the subclassing API. You can also use it as an activation function, or you could use activation=tf.exp. The exponential layer is sometimes used in the output layer of a regression model when the values to predict have very different scales (e.g., 0.001, 10., 1,000.). In fact, the exponential function is one of the standard activation functions in Keras, so you can just use activation="exponential".

As you might guess, to build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the tf.keras.layers.Layer class. For example, the following class implements a simplified version of the Dense layer:

```
class MyDense(tf.keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
```

```
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": tf.keras.activations.serialize(self.activation)}
```

Let’s walk through this code:

- The constructor takes all the hyperparameters as arguments (in this example, units and activation), and importantly it also takes a **kwargs argument. It calls the parent constructor, passing it the kwargs: this takes care of standard arguments such as input_shape, trainable, and name. Then it saves the hyperparameters as attributes, converting the activation argument to the appropriate activation function using the tf.keras.activations.get() function (it accepts functions, standard strings like "relu" or "swish", or simply None).
- The build() method’s role is to create the layer’s variables by calling the add_weight() method for each weight. The build() method is called the first time the layer is used. At that point, Keras will know the shape of this layer’s inputs, and it will pass it to the build() method, [10] which is often necessary to create some of the weights. For example, we need to know the number of neurons in the previous layer in order to create the connection weights matrix (i.e., the "kernel"): this corresponds to the size of the last dimension of the inputs. At the end of the build() method (and only at the end), you must call the parent’s build() method: this tells Keras that the layer is built (it just sets self.built = True).
- The call() method performs the desired operations. In this case, we compute the matrix multiplication of the inputs X and the layer’s kernel, we add the bias vector, and we apply the activation function to the

result, and this gives us the output of the layer.

- The get_config() method is just like in the previous custom classes. Note that we save the activation function's full configuration by calling tf.keras.activa tions.serialize().

You can now use a MyDense layer just like any other layer!

> **NOTE**
>
> Keras automatically infers the output shape, except when the layer is dynamic (as you will see shortly). In this (rare) case, you need to implement the compute_output_shape() method, which must return a TensorShape object.

To create a layer with multiple inputs (e.g., Concatenate), the argument to the call() method should be a tuple containing all the inputs. To create a layer with multiple outputs, the call() method should return the list of outputs. For example, the following toy layer takes two inputs and returns three outputs:

```
class MyMultiLayer(tf.keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return X1 + X2, X1 * X2, X1 / X2
```

This layer may now be used like any other layer, but of course only using the functional and subclassing APIs, not the sequential API (which only accepts layers with one input and one output).

If your layer needs to have a different behavior during training and during testing (e.g., if it uses Dropout or BatchNormalization layers), then you must add a training argument to the call() method and use this argument to decide what to do. For example, let's create a layer that adds Gaussian noise during training (for regularization) but does nothing during testing (Keras has a layer that does the same thing, tf.keras.layers.GaussianNoise):

```
class MyGaussianNoise(tf.keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
```

```
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=False):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X
```

With that, you can now build any custom layer you need! Now let’s look at how to create custom models.

Custom Models

We already looked at creating custom model classes in Chapter 10, when we discussed the subclassing API. [11] It's straightforward: subclass the tf.keras.Model class, create layers and variables in the constructor, and implement the call() method to do whatever you want the model to do. For example, suppose we want to build the model represented in Figure 12-3.

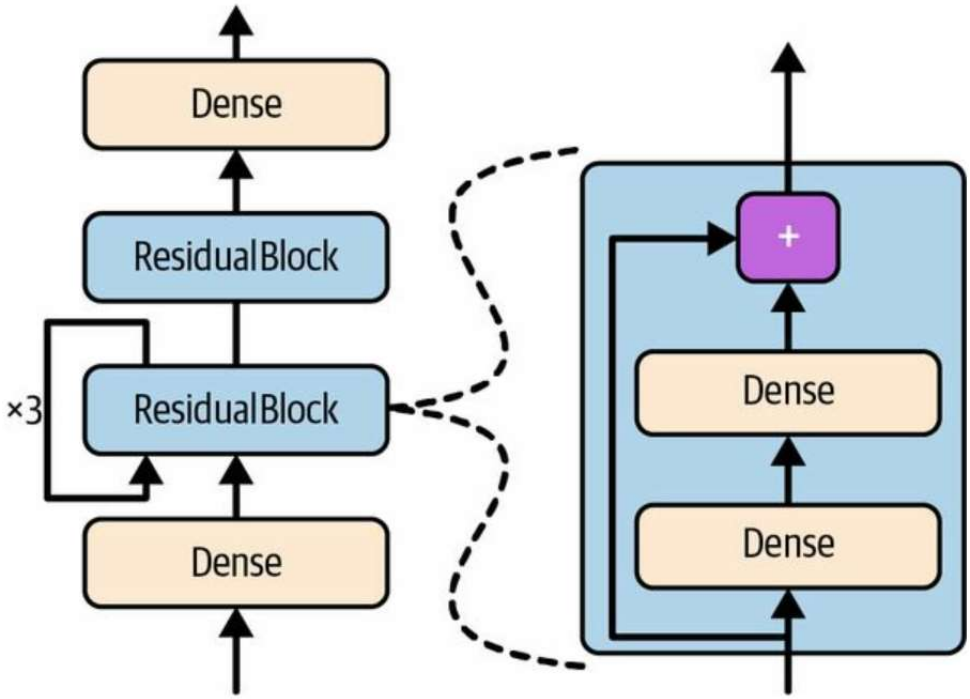

Figure 12-3. Custom model example: an arbitrary model with a custom ResidualBlock layer containing a skip connection

The inputs go through a first dense layer, then through a *residual block* composed of two dense layers and an addition operation (as you will see in Chapter 14, a residual block adds its inputs to its outputs), then through this same residual block three more times, then through a second residual block, and the final result goes through a dense output layer. Don't worry if this model does not make much sense; it's just an example to illustrate the fact

that you can easily build any kind of model you want, even one that contains loops and skip connections. To implement this model, it is best to first create a ResidualBlock layer, since we are going to create a couple of identical blocks (and we might want to reuse it in another model):

```
class ResidualBlock(tf.keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [tf.keras.layers.Dense(n_neurons, activation="relu",
                                             kernel_initializer="he_normal")
                       for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

This layer is a bit special since it contains other layers. This is handled transparently by Keras: it automatically detects that the hidden attribute contains trackable objects (layers in this case), so their variables are automatically added to this layer's list of variables. The rest of this class is self-explanatory. Next, let's use the subclassing API to define the model itself:

```
class ResidualRegressor(tf.keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = tf.keras.layers.Dense(30, activation="relu",
                                             kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = tf.keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

We create the layers in the constructor and use them in the call() method.

This model can then be used like any other model (compile it, fit it, evaluate it, and use it to make predictions). If you also want to be able to save the model using the save() method and load it using the tf.keras.models.load_model() function, you must implement the get_config() method (as we did earlier) in both the ResidualBlock class and the ResidualRegressor class. Alternatively, you can save and load the weights using the save_weights() and load_weights() methods.

The Model class is a subclass of the Layer class, so models can be defined and used exactly like layers. But a model has some extra functionalities, including of course its compile(), fit(), evaluate(), and predict() methods (and a few variants), plus the get_layer() method (which can return any of the model's layers by name or by index) and the save() method (and support for tf.keras.models.load_model() and tf.keras.models.clone_model()).

> **TIP**
>
> If models provide more functionality than layers, why not just define every layer as a model? Well, technically you could, but it is usually cleaner to distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train). The former should subclass the Layer class, while the latter should subclass the Model class.

With that, you can naturally and concisely build almost any model that you find in a paper, using the sequential API, the functional API, the subclassing API, or even a mix of these. "Almost" any model? Yes, there are still a few things that we need to look at: first, how to define losses or metrics based on model internals, and second, how to build a custom training loop.

Losses and Metrics Based on Model Internals

The custom losses and metrics we defined earlier were all based on the labels and the predictions (and optionally sample weights). There will be times when you want to define losses based on other parts of your model, such as the weights or activations of its hidden layers. This may be useful for regularization purposes or to monitor some internal aspect of your model.

To define a custom loss based on model internals, compute it based on any part of the model you want, then pass the result to the add_loss() method. For example, let's build a custom regression MLP model composed of a stack of five hidden layers plus an output layer. This custom model will also have an auxiliary output on top of the upper hidden layer. The loss associated with this auxiliary output will be called the *reconstruction loss* (see Chapter 17): it is the mean squared difference between the reconstruction and the inputs. By adding this reconstruction loss to the main loss, we will encourage the model to preserve as much information as possible through the hidden layers—even information that is not directly useful for the regression task itself. In practice, this loss sometimes improves generalization (it is a regularization loss). It is also possible to add a custom metric using the model's add_metric() method. Here is the code for this custom model with a custom reconstruction loss and a corresponding metric:

```
class ReconstructingRegressor(tf.keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [tf.keras.layers.Dense(30, activation="relu",
                                             kernel_initializer="he_normal")
                       for _ in range(5)]
        self.out = tf.keras.layers.Dense(output_dim)
        self.reconstruction_mean = tf.keras.metrics.Mean(
            name="reconstruction_error")

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = tf.keras.layers.Dense(n_inputs)

    def call(self, inputs, training=False):
```

```
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        if training:
            result = self.reconstruction_mean(recon_loss)
            self.add_metric(result)
        return self.out(Z)
```

Let’s go through this code:

- The constructor creates the DNN with five dense hidden layers and one dense output layer. We also create a Mean streaming metric to keep track of the reconstruction error during training.
- The build() method creates an extra dense layer that will be used to reconstruct the inputs of the model. It must be created here because its number of units must be equal to the number of inputs, and this number is unknown before the build() method is called. [12]
- The call() method processes the inputs through all five hidden layers, then passes the result through the reconstruction layer, which produces the reconstruction.
- Then the call() method computes the reconstruction loss (the mean squared difference between the reconstruction and the inputs), and adds it to the model’s list of losses using the add_loss() method. [13] Notice that we scale down the reconstruction loss by multiplying it by 0.05 (this is a hyperparameter you can tune). This ensures that the reconstruction loss does not dominate the main loss.
- Next, during training only, the call() method updates the reconstruction metric and adds it to the model so it can be displayed. This code example can actually be simplified by calling self.add_metric(recon_loss) instead: Keras will automatically track the mean for you.
- Finally, the call() method passes the output of the hidden layers to the

output layer and returns its output.

Both the total loss and the reconstruction loss will go down during training:

```
Epoch 1/5
363/363 [========] - 1s 820us/step - loss: 0.7640 - reconstruction_error: 1.2728
Epoch 2/5
363/363 [========] - 0s 809us/step - loss: 0.4584 - reconstruction_error: 0.6340
[...]
```

In most cases, everything we have discussed so far will be sufficient to implement whatever model you want to build, even with complex architectures, losses, and metrics. However, for some architectures, such as GANs (see Chapter 17), you will have to customize the training loop itself. Before we get there, we must look at how to compute gradients automatically in TensorFlow.

Computing Gradients Using Autodiff

To understand how to use autodiff (see Chapter 10 and Appendix B) to compute gradients automatically, let's consider a simple toy function:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

If you know calculus, you can analytically find that the partial derivative of this function with regard to w1 is 6 * w1 + 2 * w2. You can also find that its partial derivative with regard to w2 is 2 * w1. For example, at the point (w1, w2) = (5, 3), these partial derivatives are equal to 36 and 10, respectively, so the gradient vector at this point is (36, 10). But if this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be a virtually impossible task. One solution could be to compute an approximation of each partial derivative by measuring how much the function's output changes when you tweak the corresponding parameter by a tiny amount:

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Looks about right! This works rather well and is easy to implement, but it is just an approximation, and importantly you need to call f() at least once per parameter (not twice, since we could compute f(w1, w2) just once). Having to call f() at least once per parameter makes this approach intractable for large neural networks. So instead, we should use reverse-mode autodiff. TensorFlow makes this pretty simple:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
```

```
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

We first define two variables w1 and w2, then we create a tf.GradientTape context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result z with regard to both variables [w1, w2]. Let's take a look at the gradients that TensorFlow computed:

```
>>> gradients
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

Perfect! Not only is the result accurate (the precision is only limited by the floating-point errors), but the gradient() method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is incredibly efficient. It's like magic!

> **TIP**
>
> In order to save memory, only put the strict minimum inside the tf.GradientTape() block. Alternatively, pause recording by creating a with tape.stop_recording() block inside the tf.GradientTape() block.

The tape is automatically erased immediately after you call its gradient() method, so you will get an exception if you try to call gradient() twice:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)  # returns tensor 36.0
dz_dw2 = tape.gradient(z, w2)  # raises a RuntimeError!
```

If you need to call gradient() more than once, you must make the tape persistent and delete it each time you are done with it to free resources:

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)  # returns tensor 36.0
dz_dw2 = tape.gradient(z, w2)  # returns tensor 10.0, works fine now!
del tape
```

By default, the tape will only track operations involving variables, so if you try to compute the gradient of z with regard to anything other than a variable, the result will be None:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])  # returns [None, None]
```

However, you can force the tape to watch any tensors you like, to record every operation that involves them. You can then compute gradients with regard to these tensors, as if they were variables:

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])  # returns [tensor 36., tensor 10.]
```

This can be useful in some cases, like if you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little: the loss will be based on the gradient of the activations with regard to the inputs. Since the inputs are not variables, you'll need to tell the tape to watch them.

Most of the time a gradient tape is used to compute the gradients of a single value (usually the loss) with regard to a set of values (usually the model parameters). This is where reverse-mode autodiff shines, as it just needs to do one forward pass and one reverse pass to get all the gradients at once. If you try to compute the gradients of a vector, for example a vector containing

multiple losses, then TensorFlow will compute the gradients of the vector's sum. So if you ever need to get the individual gradients (e.g., the gradients of each loss with regard to the model parameters), you must call the tape's jacobian() method: it will perform reverse-mode autodiff once for each loss in the vector (all in parallel by default). It is even possible to compute second-order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives), but this is rarely needed in practice (see the "Computing Gradients Using Autodiff" section of this chapter's notebook for an example).

In some cases you may want to stop gradients from backpropagating through some part of your neural network. To do this, you must use the tf.stop_gradient() function. The function returns its inputs during the forward pass (like tf.identity()), but it does not let gradients through during backpropagation (it acts like a constant):

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2)  # the forward pass is not affected by stop_gradient()

gradients = tape.gradient(z, [w1, w2])  # returns [tensor 30., None]
```

Finally, you may occasionally run into some numerical issues when computing gradients. For example, if you compute the gradients of the square root function at $x = 10^{-50}$, the result will be infinite. In reality, the slope at that point is not infinite, but it's more than 32-bit floats can handle:

```
>>> x = tf.Variable(1e-50)
>>> with tf.GradientTape() as tape:
...     z = tf.sqrt(x)
...
>>> tape.gradient(z, [x])
[<tf.Tensor: shape=(), dtype=float32, numpy=inf>]
```

To solve this, it's often a good idea to add a tiny value to x (such as 10^{-6}) when computing its square root.

The exponential function is also a frequent source of headaches, as it grows extremely fast. For example, the way my_softplus() was defined earlier is not numerically stable. If you compute my_softplus(100.0), you will get infinity rather than the correct result (about 100). But it's possible to rewrite the function to make it numerically stable: the softplus function is defined as $\log(1 + \exp(z))$, which is also equal to $\log(1 + \exp(-|z|)) + \max(z, 0)$ (see the notebook for the mathematical proof) and the advantage of this second form is that the exponential term cannot explode. So, here's a better implementation of the my_softplus() function:

```
def my_softplus(z):
    return tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
```

In some rare cases, a numerically stable function may still have numerically unstable gradients. In such cases, you will have to tell TensorFlow which equation to use for the gradients, rather than letting it use autodiff. For this, you must use the @tf.cus tom_gradient decorator when defining the function, and return both the function's usual result and a function that computes the gradients. For example, let's update the my_softplus() function to also return a numerically stable gradients function:

```
@tf.custom_gradient
def my_softplus(z):
    def my_softplus_gradients(grads):  # grads = backprop'ed from upper layers
        return grads * (1 - 1 / (1 + tf.exp(z)))  # stable grads of softplus

    result = tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
    return result, my_softplus_gradients
```

If you know differential calculus (see the tutorial notebook on this topic), you can find that the derivative of $\log(1 + \exp(z))$ is $\exp(z) / (1 + \exp(z))$. But this form is not stable: for large values of z, it ends up computing infinity divided by infinity, which returns NaN. However, with a bit of algebraic manipulation, you can show that it's also equal to $1 – 1 / (1 + \exp(z))$, which *is* stable. The my_softplus_gradients() function uses this equation to compute the gradients. Note that this function will receive as input the gradients that were backpropagated so far, down to the my_softplus() function, and

according to the chain rule we must multiply them with this function's gradients.

Now when we compute the gradients of the my_softplus() function, we get the proper result, even for large input values.

Congratulations! You can now compute the gradients of any function (provided it is differentiable at the point where you compute it), even blocking backpropagation when needed, and write your own gradient functions! This is probably more flexibility than you will ever need, even if you build your own custom training loops. You'll see how to do that next.

Custom Training Loops

In some cases, the fit() method may not be flexible enough for what you need to do. For example, the Wide & Deep paper we discussed in Chapter 10 uses two different optimizers: one for the wide path and the other for the deep path. Since the fit() method only uses one optimizer (the one that we specify when compiling the model), implementing this paper requires writing your own custom loop.

You may also like to write custom training loops simply to feel more confident that they do precisely what you intend them to do (perhaps you are unsure about some details of the fit() method). It can sometimes feel safer to make everything explicit. However, remember that writing a custom training loop will make your code longer, more error-prone, and harder to maintain.

> **TIP**
>
> Unless you're learning or you really need the extra flexibility, you should prefer using the fit() method rather than implementing your own training loop, especially if you work in a team.

First, let's build a simple model. There's no need to compile it, since we will handle the training loop manually:

```
l2_reg = tf.keras.regularizers.l2(0.05)
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(30, activation="relu", kernel_initializer="he_normal",
                          kernel_regularizer=l2_reg),
    tf.keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Next, let's create a tiny function that will randomly sample a batch of instances from the training set (in Chapter 13 we will discuss the tf.data API, which offers a much better alternative):

```
def random_batch(X, y, batch_size=32):
```

```
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Let's also define a function that will display the training status, including the number of steps, the total number of steps, the mean loss since the start of the epoch (we will use the Mean metric to compute it), and other metrics:

```
def print_status_bar(step, total, loss, metrics=None):
    metrics = " - ".join([f"{m.name}: {m.result():.4f}"
                          for m in [loss] + (metrics or [])])
    end = "" if step < total else "\n"
    print(f"\r{step}/{total} - " + metrics, end=end)
```

This code is self-explanatory, unless you are unfamiliar with Python string formatting: {m.result():.4f} will format the metric's result as a float with four digits after the decimal point, and using \r (carriage return) along with end="" ensures that the status bar always gets printed on the same line.

With that, let's get down to business! First, we need to define some hyperparameters and choose the optimizer, the loss function, and the metrics (just the MAE in this example):

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
mean_loss = tf.keras.metrics.Mean(name="mean_loss")
metrics = [tf.keras.metrics.MeanAbsoluteError()]
```

And now we are ready to build the custom loop!

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
```

```
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)

        print_status_bar(step, n_steps, mean_loss, metrics)

    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

There's a lot going on in this code, so let's walk through it:

- We create two nested loops: one for the epochs, the other for the batches within an epoch.
- Then we sample a random batch from the training set.
- Inside the tf.GradientTape() block, we make a prediction for one batch, using the model as a function, and we compute the loss: it is equal to the main loss plus the other losses (in this model, there is one regularization loss per layer). Since the mean_squared_error() function returns one loss per instance, we compute the mean over the batch using tf.reduce_mean() (if you wanted to apply different weights to each instance, this is where you would do it). The regularization losses are already reduced to a single scalar each, so we just need to sum them (using tf.add_n(), which sums multiple tensors of the same shape and data type).
- Next, we ask the tape to compute the gradients of the loss with regard to each trainable variable—*not* all variables!—and we apply them to the optimizer to perform a gradient descent step.
- Then we update the mean loss and the metrics (over the current epoch), and we display the status bar.
- At the end of each epoch, we reset the states of the mean loss and the metrics.

If you want to apply gradient clipping (see Chapter 11), set the optimizer's

clipnorm or clipvalue hyperparameter. If you want to apply any other transformation to the gradients, simply do so before calling the apply_gradients() method. And if you want to add weight constraints to your model (e.g., by setting kernel_constraint or bias_constraint when creating a layer), you should update the training loop to apply these constraints just after apply_gradients(), like so:

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

> **WARNING**
>
> Don't forget to set training=True when calling the model in the training loop, especially if your model behaves differently during training and testing (e.g., if it uses BatchNormalization or Dropout). If it's a custom model, make sure to propagate the training argument to the layers that your model calls.

As you can see, there are quite a lot of things you need to get right, and it's easy to make a mistake. But on the bright side, you get full control.

Now that you know how to customize any part of your models [15] and training algorithms, let's see how you can use TensorFlow's automatic graph generation feature: it can speed up your custom code considerably, and it will also make it portable to any platform supported by TensorFlow (see Chapter 19).

TensorFlow Functions and Graphs

Back in TensorFlow 1, graphs were unavoidable (as were the complexities that came with them) because they were a central part of TensorFlow's API. Since TensorFlow 2 (released in 2019), graphs are still there, but not as central, and they're much (much!) simpler to use. To show just how simple, let's start with a trivial function that computes the cube of its input:

```
def cube(x):
    return x ** 3
```

We can obviously call this function with a Python value, such as an int or a float, or we can call it with a tensor:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Now, let's use tf.function() to convert this Python function to a *TensorFlow function*:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x7fbfe0c54d50>
```

This TF function can then be used exactly like the original Python function, and it will return the same result (but always as tensors):

```
>>> tf_cube(2)
<tf.Tensor: shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Under the hood, tf.function() analyzed the computations performed by the cube() function and generated an equivalent computation graph! As you can

see, it was rather painless (we will look at how this works shortly). Alternatively, we could have used tf.function as a decorator; this is actually more common:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

The original Python function is still available via the TF function's python_function attribute, in case you ever need it:

```
>>> tf_cube.python_function(2)
8
```

TensorFlow optimizes the computation graph, pruning unused nodes, simplifying expressions (e.g., 1 + 2 would get replaced with 3), and more. Once the optimized graph is ready, the TF function efficiently executes the operations in the graph, in the appropriate order (and in parallel when it can). As a result, a TF function will usually run much faster than the original Python function, especially if it performs complex computations. [16] Most of the time you will not really need to know more than that: when you want to boost a Python function, just transform it into a TF function. That's all!

Moreover, if you set jit_compile=True when calling tf.function(), then TensorFlow will use *accelerated linear algebra* (XLA) to compile dedicated kernels for your graph, often fusing multiple operations. For example, if your TF function calls tf.reduce_sum(a * b + c), then without XLA the function would first need to compute a * b and store the result in a temporary variable, then add c to that variable, and lastly call tf.reduce_sum() on the result. With XLA, the whole computation gets compiled into a single kernel, which will compute tf.reduce_sum(a * b + c) in one shot, without using any large temporary variable. Not only will this be much faster, it will also use dramatically less RAM.

When you write a custom loss function, a custom metric, a custom layer, or any other custom function and you use it in a Keras model (as we've done throughout this chapter), Keras automatically converts your function into a

TF function—no need to use tf.function(). So most of the time, the magic is 100% transparent. And if you want Keras to use XLA, you just need to set jit_compile=True when calling the compile() method. Easy!

> **TIP**
>
> You can tell Keras *not* to convert your Python functions to TF functions by setting dynamic=True when creating a custom layer or a custom model. Alternatively, you can set run_eagerly=True when calling the model's compile() method.

By default, a TF function generates a new graph for every unique set of input shapes and data types and caches it for subsequent calls. For example, if you call tf_cube(tf.constant(10)), a graph will be generated for int32 tensors of shape []. Then if you call tf_cube(tf.constant(20)), the same graph will be reused. But if you then call tf_cube(tf.constant([10, 20])), a new graph will be generated for int32 tensors of shape [2]. This is how TF functions handle polymorphism (i.e., varying argument types and shapes). However, this is only true for tensor arguments: if you pass numerical Python values to a TF function, a new graph will be generated for every distinct value: for example, calling tf_cube(10) and tf_cube(20) will generate two graphs.

> **WARNING**
>
> If you call a TF function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM (you must delete the TF function to release it). Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per layer. This allows TensorFlow to better optimize each variant of your model.

AutoGraph and Tracing

So how does TensorFlow generate graphs? It starts by analyzing the Python function's source code to capture all the control flow statements, such as for loops, while loops, and if statements, as well as break, continue, and return statements. This first step is called *AutoGraph*. The reason TensorFlow has to analyze the source code is that Python does not provide any other way to capture control flow statements: it offers magic methods like __add__() and __mul__() to capture operators like + and *, but there are no __while__() or __if__() magic methods. After analyzing the function's code, AutoGraph outputs an upgraded version of that function in which all the control flow statements are replaced by the appropriate TensorFlow operations, such as tf.while_loop() for loops and tf.cond() for if statements. For example, in Figure 12-4, AutoGraph analyzes the source code of the sum_squares() Python function, and it generates the tf__sum_squares() function. In this function, the for loop is replaced by the definition of the loop_body() function (containing the body of the original for loop), followed by a call to the for_stmt() function. This call will build the appropriate tf.while_loop() operation in the computation graph.

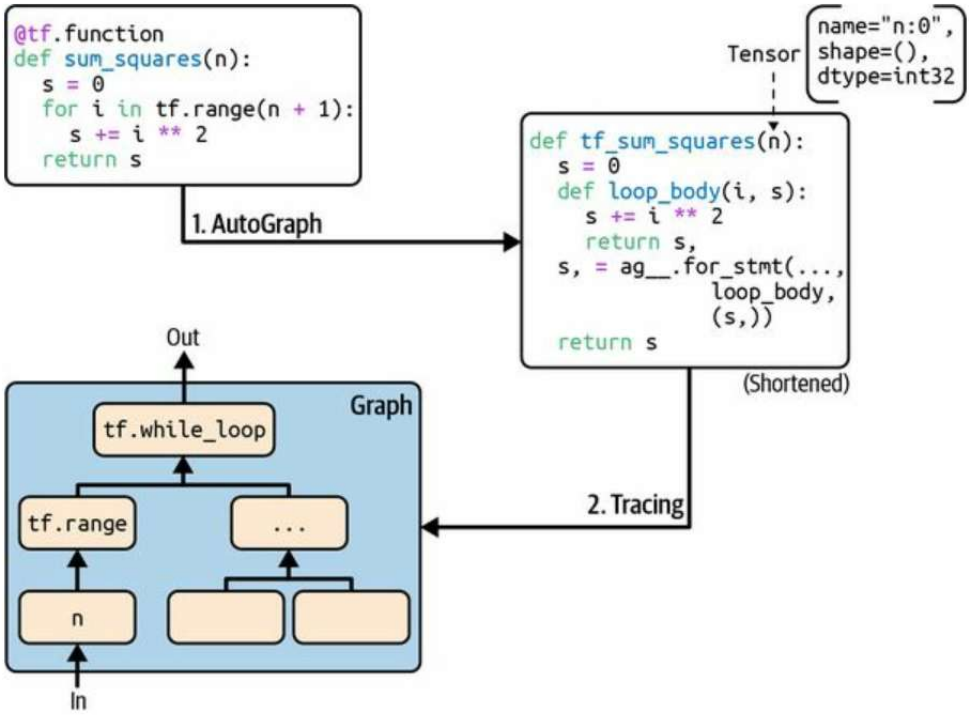

Figure 12-4. How TensorFlow generates graphs using AutoGraph and tracing

Next, TensorFlow calls this "upgraded" function, but instead of passing the argument, it passes a *symbolic tensor*—a tensor without any actual value, only a name, a data type, and a shape. For example, if you call sum_squares(tf.constant(10)), then the tf__sum_squares() function will be called with a symbolic tensor of type int32 and shape []. The function will run in *graph mode*, meaning that each TensorFlow operation will add a node in the graph to represent itself and its output tensor(s) (as opposed to the regular mode, called *eager execution*, or *eager mode*). In graph mode, TF operations do not perform any computations. Graph mode was the default mode in TensorFlow 1. In Figure 12-4, you can see the tf__sum_squares() function being called with a symbolic tensor as its argument (in this case, an int32 tensor of shape []) and the final graph being generated during tracing. The nodes represent operations, and the arrows represent tensors (both the generated function and the graph are simplified).

TIP

In order to view the generated function's source code, you can call tf.autograph.to_code(sum_squares.python_function). The code is not meant to be pretty, but it can sometimes help for debugging.

TF Function Rules

Most of the time, converting a Python function that performs TensorFlow operations into a TF function is trivial: decorate it with @tf.function or let Keras take care of it for you. However, there are a few rules to respect:

- If you call any external library, including NumPy or even the standard library, this call will run only during tracing; it will not be part of the graph. Indeed, a TensorFlow graph can only include TensorFlow constructs (tensors, operations, variables, datasets, and so on). So, make sure you use tf.reduce_sum() instead of np.sum(), tf.sort() instead of the built-in sorted() function, and so on (unless you really want the code to run only during tracing). This has a few additional implications:
  - If you define a TF function f(*x*) that just returns np.random.rand(), a random number will only be generated when the function is traced, so f(tf.constant(2.)) and f(tf.constant(3.)) will return the same random number, but f(tf.constant([2., 3.])) will return a different one. If you replace np.random.rand() with tf.random.uniform([]), then a new random number will be generated upon every call, since the operation will be part of the graph.
  - If your non-TensorFlow code has side effects (such as logging something or updating a Python counter), then you should not expect those side effects to occur every time you call the TF function, as they will only occur when the function is traced.
  - You can wrap arbitrary Python code in a tf.py_function() operation, but doing so will hinder performance, as TensorFlow will not be able to do any graph optimization on this code. It will also reduce portability, as the graph will only run on platforms where Python is available (and where the right libraries are installed).
- You can call other Python functions or TF functions, but they should follow the same rules, as TensorFlow will capture their operations in the

computation graph. Note that these other functions do not need to be decorated with @tf.function.

- If the function creates a TensorFlow variable (or any other stateful TensorFlow object, such as a dataset or a queue), it must do so upon the very first call, and only then, or else you will get an exception. It is usually preferable to create variables outside of the TF function (e.g., in the build() method of a custom layer). If you want to assign a new value to the variable, make sure you call its assign() method instead of using the = operator.
- The source code of your Python function should be available to TensorFlow. If the source code is unavailable (for example, if you define your function in the Python shell, which does not give access to the source code, or if you deploy only the compiled **.pyc* Python files to production), then the graph generation process will fail or have limited functionality.
- TensorFlow will only capture for loops that iterate over a tensor or a tf.data.Dataset (see Chapter 13). Therefore, make sure you use for i in tf.range(*x*) rather than for i in range(*x*), or else the loop will not be captured in the graph. Instead, it will run during tracing. (This may be what you want if the for loop is meant to build the graph; for example, to create each layer in a neural network.)
- As always, for performance reasons, you should prefer a vectorized implementation whenever you can, rather than using loops.

It's time to sum up! In this chapter we started with a brief overview of TensorFlow, then we looked at TensorFlow's low-level API, including tensors, operations, variables, and special data structures. We then used these tools to customize almost every component in the Keras API. Finally, we looked at how TF functions can boost performance, how graphs are generated using AutoGraph and tracing, and what rules to follow when you write TF functions (if you would like to open the black box a bit further and explore the generated graphs, you will find technical details in Appendix D).

In the next chapter, we will look at how to efficiently load and preprocess data with TensorFlow.

Exercises

1. How would you describe TensorFlow in a short sentence? What are its main features? Can you name other popular deep learning libraries?
2. Is TensorFlow a drop-in replacement for NumPy? What are the main differences between the two?
3. Do you get the same result with tf.range(10) and tf.constant(np.ara nge(10))?
4. Can you name six other data structures available in TensorFlow, beyond regular tensors?
5. You can define a custom loss function by writing a function or by subclassing the tf.keras.losses.Loss class. When would you use each option?
6. Similarly, you can define a custom metric in a function or as a subclass of tf.keras.metrics.Metric. When would you use each option?
7. When should you create a custom layer versus a custom model?
8. What are some use cases that require writing your own custom training loop?
9. Can custom Keras components contain arbitrary Python code, or must they be convertible to TF functions?
10. What are the main rules to respect if you want a function to be convertible to a TF function?
11. When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?
12. Implement a custom layer that performs *layer normalization* (we will use this type of layer in Chapter 15):

a. The build() method should define two trainable weights **α** and **β**, both of shape input_shape[-1:] and data type tf.float32. **α** should be initialized with 1s, and **β** with 0s.

b. The call() method should compute the mean μ and standard deviation σ of each instance's features. For this, you can use tf.nn.moments(inputs, axes=-1, keepdims=True), which returns the mean μ and the variance σ^2 of all instances (compute the square root of the variance to get the standard deviation). Then the function should compute and return $\boldsymbol{\alpha} \otimes (\mathbf{X} - \mu)/(\sigma + \varepsilon) + \boldsymbol{\beta}$, where $\otimes$ represents itemwise multiplication (*) and ε is a smoothing term (a small constant to avoid division by zero, e.g., 0.001).

c. Ensure that your custom layer produces the same (or very nearly the same) output as the tf.keras.layers.LayerNormalization layer.

13. Train a model using a custom training loop to tackle the Fashion MNIST dataset (see Chapter 10):

a. Display the epoch, iteration, mean training loss, and mean accuracy over each epoch (updated at each iteration), as well as the validation loss and accuracy at the end of each epoch.

b. Try using a different optimizer with a different learning rate for the upper layers and the lower layers.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

---

1 However, Facebook's PyTorch library is currently more popular in academia: more papers cite PyTorch than TensorFlow or Keras. Moreover, Google's JAX library is gaining momentum, especially in academia.

2 TensorFlow includes another deep learning API called the *estimators API*, but it is now deprecated.

3 If you ever need to (but you probably won't), you can write your own operations using the C++ API.

4 To learn more about TPUs and how they work, check out *https://homl.info/tpus*.

5 A notable exception is tf.math.log(), which is commonly used but doesn't have a tf.log() alias, as it might be confused with logging.

6 It would not be a good idea to use a weighted mean: if you did, then two instances with the same weight but in different batches would have a different impact on training, depending on the total weight of each batch.

7 The {**x, [...]} syntax was added in Python 3.5, to merge all the key/value pairs from dictionary x into another dictionary. Since Python 3.9, you can use the nicer x | y syntax instead (where x and y are two dictionaries).

8 However, the Huber loss is seldom used as a metric—the MAE or MSE is generally preferred.

9 This class is for illustration purposes only. A simpler and better implementation would just subclass the tf.keras.metrics.Mean class; see the "Streaming Metrics" section of this chapter's notebook for an example.

10 The Keras API calls this argument input_shape, but since it also includes the batch dimension, I prefer to call it batch_input_shape.

11 The name "subclassing API" in Keras usually refers only to the creation of custom models by subclassing, although many other things can be created by subclassing, as you've seen in this chapter.

12 Due to TensorFlow issue #46858, the call to super().build() may fail in this case, unless the issue was fixed by the time you read this. If not, you need to replace this line with self.built = True.

13 You can also call add_loss() on any layer inside the model, as the model recursively gathers losses from all of its layers.

14 If the tape goes out of scope, for example when the function that used it returns, Python's garbage collector will delete it for you.

15 With the exception of optimizers, as very few people ever customize these; see the "Custom Optimizers" section in the notebook for an example.

16 However, in this trivial example, the computation graph is so small that there is nothing at all to optimize, so tf_cube() actually runs much slower than cube().

Chapter 13. Loading and Preprocessing Data with TensorFlow

In Chapter 2, you saw that loading and preprocessing data is an important part of any machine learning project. You used Pandas to load and explore the (modified) California housing dataset—which was stored in a CSV file—and you applied Scikit-Learn's transformers for preprocessing. These tools are quite convenient, and you will probably be using them often, especially when exploring and experimenting with data.

However, when training TensorFlow models on large datasets, you may prefer to use TensorFlow's own data loading and preprocessing API, called *tf.data*. It is capable of loading and preprocessing data extremely efficiently, reading from multiple files in parallel using multithreading and queuing, shuffling and batching samples, and more. Plus, it can do all of this on the fly—it loads and preprocesses the next batch of data across multiple CPU cores, while your GPUs or TPUs are busy training the current batch of data.

The tf.data API lets you handle datasets that don't fit in memory, and it allows you to make full use of your hardware resources, thereby speeding up training. Off the shelf, the tf.data API can read from text files (such as CSV files), binary files with fixed-size records, and binary files that use TensorFlow's TFRecord format, which supports records of varying sizes.

TFRecord is a flexible and efficient binary format usually containing protocol buffers (an open source binary format). The tf.data API also has support for reading from SQL databases. Moreover, many open source extensions are available to read from all sorts of data sources, such as Google's BigQuery service (see *https://tensorflow.org/io*).

Keras also comes with powerful yet easy-to-use preprocessing layers that can

be embedded in your models: this way, when you deploy a model to production, it will be able to ingest raw data directly, without you having to add any additional preprocessing code. This eliminates the risk of mismatch between the preprocessing code used during training and the preprocessing code used in production, which would likely cause *training/serving skew*. And if you deploy your model in multiple apps coded in different programming languages, you won't have to reimplement the same preprocessing code multiple times, which also reduces the risk of mismatch.

As you will see, both APIs can be used jointly—for example, to benefit from the efficient data loading offered by tf.data and the convenience of the Keras preprocessing layers.

In this chapter, we will first cover the tf.data API and the TFRecord format. Then we will explore the Keras preprocessing layers and how to use them with the tf.data API. Lastly, we will take a quick look at a few related libraries that you may find useful for loading and preprocessing data, such as TensorFlow Datasets and TensorFlow Hub. So, let's get started!

The tf.data API

The whole tf.data API revolves around the concept of a tf.data.Dataset: this represents a sequence of data items. Usually you will use datasets that gradually read data from disk, but for simplicity let's create a dataset from a simple data tensor using tf.data.Dataset.from_tensor_slices():

```
>>> import tensorflow as tf
>>> X = tf.range(10)  # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

The from_tensor_slices() function takes a tensor and creates a tf.data.Dataset whose elements are all the slices of X along the first dimension, so this dataset contains 10 items: tensors 0, 1, 2, …, 9. In this case we would have obtained the same dataset if we had used tf.data.Dataset.range(10) (except the elements would be 64-bit integers instead of 32-bit integers).

You can simply iterate over a dataset's items like this:

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

> **NOTE**
>
> The tf.data API is a streaming API: you can very efficiently iterate through a dataset's items, but the API is not designed for indexing or slicing.

A dataset may also contain tuples of tensors, or dictionaries of name/tensor pairs, or even nested tuples and dictionaries of tensors. When slicing a tuple,

a dictionary, or a nested structure, the dataset will only slice the tensors it contains, while preserving the tuple/dictionary structure. For example:

```
>>> X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}
>>> dataset = tf.data.Dataset.from_tensor_slices(X_nested)
>>> for item in dataset:
...     print(item)
...
{'a': (<tf.Tensor: [...]=1>, <tf.Tensor: [...]=4>), 'b': <tf.Tensor: [...]=7>}
{'a': (<tf.Tensor: [...]=2>, <tf.Tensor: [...]=5>), 'b': <tf.Tensor: [...]=8>}
{'a': (<tf.Tensor: [...]=3>, <tf.Tensor: [...]=6>), 'b': <tf.Tensor: [...]=9>}
```

Chaining Transformations

Once you have a dataset, you can apply all sorts of transformations to it by calling its transformation methods. Each method returns a new dataset, so you can chain transformations like this (this chain is illustrated in Figure 13-1):

```
>>> dataset = tf.data.Dataset.from_tensor_slices(tf.range(10))
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

In this example, we first call the repeat() method on the original dataset, and it returns a new dataset that repeats the items of the original dataset three times. Of course, this will not copy all the data in memory three times! If you call this method with no arguments, the new dataset will repeat the source dataset forever, so the code that iterates over the dataset will have to decide when to stop.

Then we call the batch() method on this new dataset, and again this creates a new dataset. This one will group the items of the previous dataset in batches of seven items.

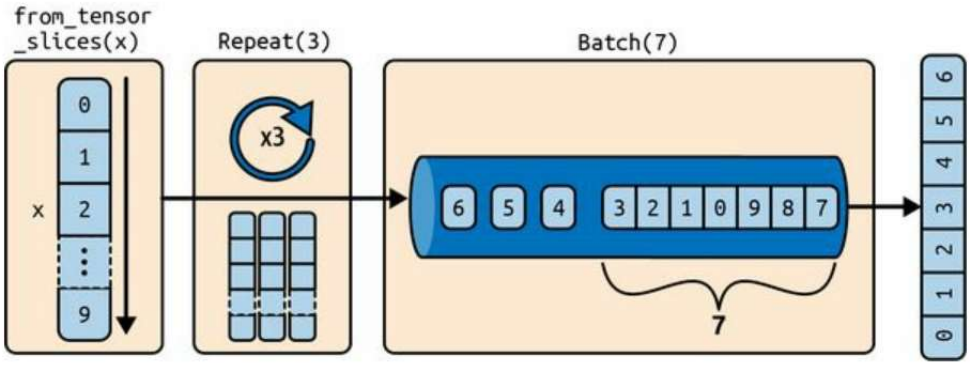

Figure 13-1. Chaining dataset transformations

Finally, we iterate over the items of this final dataset. The batch() method had to output a final batch of size two instead of seven, but you can call batch() with drop_remainder=True if you want it to drop this final batch, such that all batches have the exact same size.

> **WARNING**
>
> The dataset methods do *not* modify datasets—they create new ones. So make sure to keep a reference to these new datasets (e.g., with dataset = ...), or else nothing will happen.

You can also transform the items by calling the map() method. For example, this creates a new dataset with all batches multiplied by two:

```
>>> dataset = dataset.map(lambda x: x * 2)  # x is a batch
>>> for item in dataset:
...     print(item)
...
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int32)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
[...]
```

This map() method is the one you will call to apply any preprocessing to your data. Sometimes this will include computations that can be quite intensive, such as reshaping or rotating an image, so you will usually want to spawn multiple threads to speed things up. This can be done by setting the

num_parallel_calls argument to the number of threads to run, or to tf.data.AUTOTUNE. Note that the function you pass to the map() method must be convertible to a TF function (see Chapter 12).

It is also possible to simply filter the dataset using the filter() method. For example, this code creates a dataset that only contains the batchs whose sum is greater than 50:

```
>>> dataset = dataset.filter(lambda x: tf.reduce_sum(x) > 50)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
```

You will often want to look at just a few items from a dataset. You can use the take() method for that:

```
>>> for item in dataset.take(2):
...     print(item)
...
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
```

Shuffling the Data

As we discussed in Chapter 4, gradient descent works best when the instances in the training set are independent and identically distributed (IID). A simple way to ensure this is to shuffle the instances, using the shuffle() method. It will create a new dataset that will start by filling up a buffer with the first items of the source dataset. Then, whenever it is asked for an item, it will pull one out randomly from the buffer and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it will continue to pull out items randomly from the buffer until it is empty. You must specify the buffer size, and it is important to make it large enough, or else shuffling will not be very effective. [1] Just don't exceed the amount of RAM you have, though even if you have plenty of it, there's no need to go beyond the dataset's size. You can provide a random seed if you want the same random order every time you run your program. For example, the following code creates and displays a dataset containing the integers 0 to 9, repeated twice, shuffled using a buffer of size 4 and a random seed of 42, and batched with a batch size of 7:

```
>>> dataset = tf.data.Dataset.range(10).repeat(2)
>>> dataset = dataset.shuffle(buffer_size=4, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([3 0 1 6 2 5 7], shape=(7,), dtype=int64)
tf.Tensor([8 4 1 9 4 2 3], shape=(7,), dtype=int64)
tf.Tensor([7 5 0 8 9 6], shape=(6,), dtype=int64)
```

> **TIP**
>
> If you call repeat() on a shuffled dataset, by default it will generate a new order at every iteration. This is generally a good idea, but if you prefer to reuse the same order at each iteration (e.g., for tests or debugging), you can set reshuffle_each_itera tion=False when calling shuffle().

For a large dataset that does not fit in memory, this simple shuffling-buffer

approach may not be sufficient, since the buffer will be small compared to the dataset. One solution is to shuffle the source data itself (for example, on Linux you can shuffle text files using the shuf command). This will definitely improve shuffling a lot! Even if the source data is shuffled, you will usually want to shuffle it some more, or else the same order will be repeated at each epoch, and the model may end up being biased (e.g., due to some spurious patterns present by chance in the source data's order). To shuffle the instances some more, a common approach is to split the source data into multiple files, then read them in a random order during training. However, instances located in the same file will still end up close to each other. To avoid this you can pick multiple files randomly and read them simultaneously, interleaving their records. Then on top of that you can add a shuffling buffer using the shuffle() method. If this sounds like a lot of work, don't worry: the tf.data API makes all this possible in just a few lines of code. Let's go over how you can do this.

Interleaving Lines from Multiple Files

First, suppose you've loaded the California housing dataset, shuffled it (unless it was already shuffled), and split it into a training set, a validation set, and a test set. Then you split each set into many CSV files that each look like this (each row contains eight input features plus the target median house value):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul…,AveOccup,Lat…,Long…,MedianHouseValue
3.5214,15.0,3.050,1.107,1447.0,1.606,37.63,-122.43,1.442
5.3275,5.0,6.490,0.991,3464.0,3.443,33.69,-117.39,1.687
3.1,29.0,7.542,1.592,1328.0,2.251,38.44,-122.98,1.621
[...]
```

Let's also suppose train_filepaths contains the list of training filepaths (and you also have valid_filepaths and test_filepaths):

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv', ...]
```

Alternatively, you could use file patterns; for example, train_filepaths = "datasets/housing/my_train_*.csv". Now let's create a dataset containing only these filepaths:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

By default, the list_files() function returns a dataset that shuffles the filepaths. In general this is a good thing, but you can set shuffle=False if you do not want that for some reason.

Next, you can call the interleave() method to read from five files at a time and interleave their lines. You can also skip the first line of each file—which is the header row—using the skip() method):

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
```

```
    cycle_length=n_readers)
```

The interleave() method will create a dataset that will pull five filepaths from the filepath_dataset, and for each one it will call the function you gave it (a lambda in this example) to create a new dataset (in this case a TextLineDataset). To be clear, at this stage there will be seven datasets in all: the filepath dataset, the interleave dataset, and the five TextLineDatasets created internally by the interleave dataset. When you iterate over the interleave dataset, it will cycle through these five TextLineDatasets, reading one line at a time from each until all datasets are out of items. Then it will fetch the next five filepaths from the filepath_dataset and interleave them the same way, and so on until it runs out of filepaths. For interleaving to work best, it is preferable to have files of identical length; otherwise the end of the longest file will not be interleaved.

By default, interleave() does not use parallelism; it just reads one line at a time from each file, sequentially. If you want it to actually read files in parallel, you can set the interleave() method's num_parallel_calls argument to the number of threads you want (recall that the map() method also has this argument). You can even set it to tf.data.AUTOTUNE to make TensorFlow choose the right number of threads dynamically based on the available CPU. Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line)
...
tf.Tensor(b'4.5909,16.0,[...],33.63,-117.71,2.418', shape=(), dtype=string)
tf.Tensor(b'2.4792,24.0,[...],34.18,-118.38,2.0', shape=(), dtype=string)
tf.Tensor(b'4.2708,45.0,[...],37.48,-122.19,2.67', shape=(), dtype=string)
tf.Tensor(b'2.1856,41.0,[...],32.76,-117.12,1.205', shape=(), dtype=string)
tf.Tensor(b'4.1812,52.0,[...],33.73,-118.31,3.215', shape=(), dtype=string)
```

These are the first rows (ignoring the header row) of five CSV files, chosen randomly. Looks good!

NOTE

It’s possible to pass a list of filepaths to the TextLineDataset constructor: it will go through each file in order, line by line. If you also set the num_parallel_reads argument to a number greater than one, then the dataset will read that number of files in parallel and interleave their lines (without having to call the interleave() method). However, it will *not* shuffle the files, nor will it skip the header lines.

Preprocessing the Data

Now that we have a housing dataset that returns each instance as a tensor containing a byte string, we need to do a bit of preprocessing, including parsing the strings and scaling the data. Let's implement a couple custom functions that will perform this preprocessing:

```
X_mean, X_std = [...]  # mean and scale of each feature in the training set
n_inputs = 8

def parse_csv_line(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    return tf.stack(fields[:-1]), tf.stack(fields[-1:])

def preprocess(line):
    x, y = parse_csv_line(line)
    return (x - X_mean) / X_std, y
```

Let's walk through this code:

- First, the code assumes that we have precomputed the mean and standard deviation of each feature in the training set. X_mean and X_std are just 1D tensors (or NumPy arrays) containing eight floats, one per input feature. This can be done using a Scikit-Learn StandardScaler on a large enough random sample of the dataset. Later in this chapter, we will use a Keras preprocessing layer instead.
- The parse_csv_line() function takes one CSV line and parses it. To help with that, it uses the tf.io.decode_csv() function, which takes two arguments: the first is the line to parse, and the second is an array containing the default value for each column in the CSV file. This array (defs) tells TensorFlow not only the default value for each column, but also the number of columns and their types. In this example, we tell it that all feature columns are floats and that missing values should default to zero, but we provide an empty array of type tf.float32 as the default value for the last column (the target): the array tells TensorFlow that this

column contains floats, but that there is no default value, so it will raise an exception if it encounters a missing value.

- The tf.io.decode_csv() function returns a list of scalar tensors (one per column), but we need to return a 1D tensor array. So we call tf.stack() on all tensors except for the last one (the target): this will stack these tensors into a 1D array. We then do the same for the target value: this makes it a 1D tensor array with a single value, rather than a scalar tensor. The tf.io.decode_csv() function is done, so it returns the input features and the target.
- Finally, the custom preprocess() function just calls the parse_csv_line() function, scales the input features by subtracting the feature means and then dividing by the feature standard deviations, and returns a tuple containing the scaled features and the target.

Let's test this preprocessing function:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([2.782], dtype=float32)>)
```

Looks good! The preprocess() function can convert an instance from a byte string to a nice scaled tensor, with its corresponding label. We can now use the dataset's map() method to apply the preprocess() function to each sample in the dataset.

Putting Everything Together

To make the code more reusable, let's put together everything we have discussed so far into another helper function; it will create and return a dataset that will efficiently load California housing data from multiple CSV files, preprocess it, shuffle it, and batch it (see Figure 13-2):

```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                       n_parse_threads=5, shuffle_buffer_size=10_000, seed=42,
                       batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size, seed=seed)
    return dataset.batch(batch_size).prefetch(1)
```

Note that we use the prefetch() method on the very last line. This is important for performance, as you will see now.

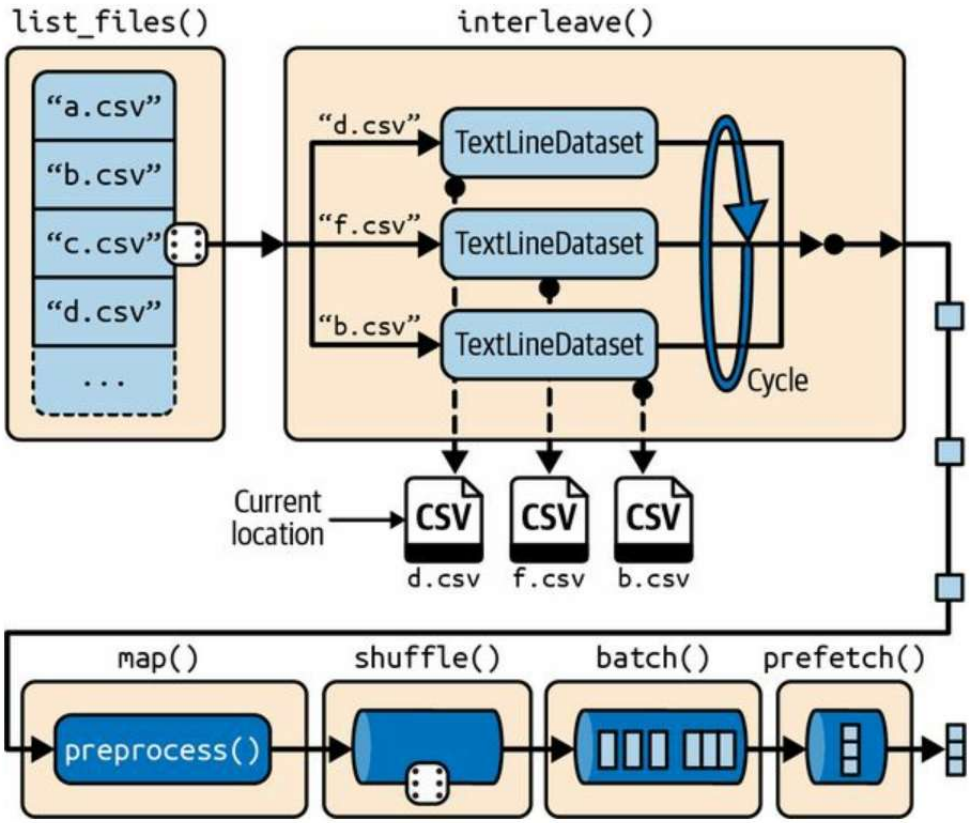

Figure 13-2. Loading and preprocessing data from multiple CSV files

Prefetching

By calling prefetch(1) at the end of the custom csv_reader_dataset() function, we are creating a dataset that will do its best to always be one batch ahead. [2] In other words, while our training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready (e.g., reading the data from disk and preprocessing it). This can improve performance dramatically, as is illustrated in Figure 13-3.

If we also ensure that loading and preprocessing are multithreaded (by setting num_parallel_calls when calling interleave() and map()), we can exploit multiple CPU cores and hopefully make preparing one batch of data shorter than running a training step on the GPU: this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU [3]), and training will run much faster.

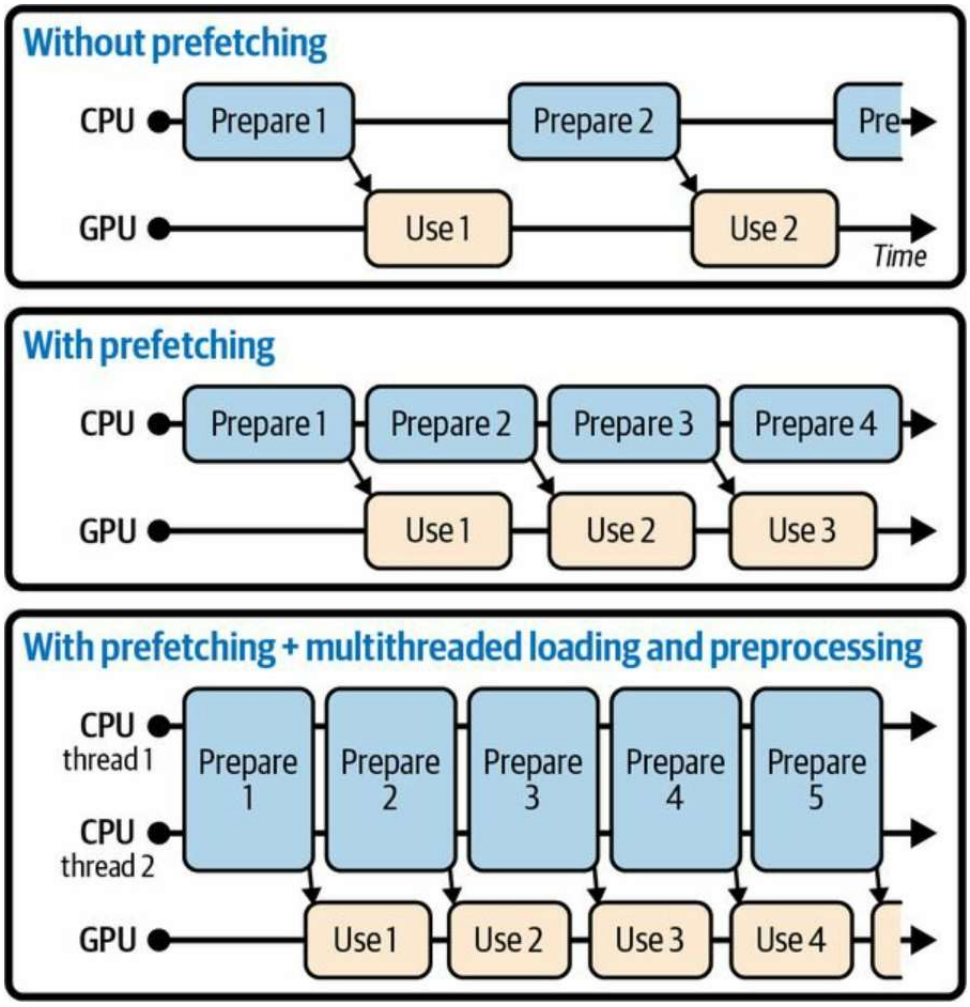

Figure 13-3. With prefetching, the CPU and the GPU work in parallel: as the GPU works on one batch, the CPU works on the next

> **TIP**
>
> If you plan to purchase a GPU card, its processing power and its memory size are of course very important (in particular, a large amount of RAM is crucial for large computer vision or natural language processing models). Just as important for good performance is the GPU's *memory bandwidth*; this is the number of gigabytes of data it can get into or out of its RAM per second.

If the dataset is small enough to fit in memory, you can significantly speed up training by using the dataset's cache() method to cache its content to RAM. You should generally do this after loading and preprocessing the data, but before shuffling, repeating, batching, and prefetching. This way, each instance will only be read and preprocessed once (instead of once per epoch), but the data will still be shuffled differently at each epoch, and the next batch will still be prepared in advance.

You have now learned how to build efficient input pipelines to load and preprocess data from multiple text files. We have discussed the most common dataset methods, but there are a few more you may want to look at, such as concatenate(), zip(), window(), reduce(), shard(), flat_map(), apply(), unbatch(), and padded_batch(). There are also a few more class methods, such as from_generator() and from_ten sors(), which create a new dataset from a Python generator or a list of tensors, respectively. Please check the API documentation for more details. Also note that there are experimental features available in tf.data.experimental, many of which will likely make it to the core API in future releases (e.g., check out the CsvDataset class, as well as the make_csv_dataset() method, which takes care of inferring the type of each column).

Using the Dataset with Keras

Now we can use the custom csv_reader_dataset() function we wrote earlier to create a dataset for the training set, and for the validation set and the test set. The training set will be shuffled at each epoch (note that the validation set and the test set will also be shuffled, even though we don't really need that):

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

Now you can simply build and train a Keras model using these datasets. When you call the model's fit() method, you pass train_set instead of X_train, y_train, and pass validation_data=valid_set instead of validation_data=(X_valid, y_valid). The fit() method will take care of repeating the training dataset once per epoch, using a different random order at each epoch:

```
model = tf.keras.Sequential([...])
model.compile(loss="mse", optimizer="sgd")
model.fit(train_set, validation_data=valid_set, epochs=5)
```

Similarly, you can pass a dataset to the evaluate() and predict() methods:

```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3)  # pretend we have 3 new samples
y_pred = model.predict(new_set)  # or you could just pass a NumPy array
```

Unlike the other sets, the new_set will usually not contain labels. If it does, as is the case here, Keras will ignore them. Note that in all these cases, you can still use NumPy arrays instead of datasets if you prefer (but of course they need to have been loaded and preprocessed first).

If you want to build your own custom training loop (as discussed in Chapter 12), you can just iterate over the training set, very naturally:

```
n_epochs = 5
```

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_set:
        [...]  # perform one gradient descent step
```

In fact, it is even possible to create a TF function (see Chapter 12) that trains the model for a whole epoch. This can really speed up training:

```
@tf.function
def train_one_epoch(model, optimizer, loss_fn, train_set):
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
for epoch in range(n_epochs):
    print("\rEpoch {}/{}".format(epoch + 1, n_epochs), end="")
    train_one_epoch(model, optimizer, loss_fn, train_set)
```

In Keras, the steps_per_execution argument of the compile() method lets you define the number of batches that the fit() method will process during each call to the tf.function it uses for training. The default is just 1, so if you set it to 50 you will often see a significant performance improvement. However, the on_batch_*() methods of Keras callbacks will only be called every 50 batches.

Congratulations, you now know how to build powerful input pipelines using the tf.data API! However, so far we've been using CSV files, which are common, simple, and convenient but not really efficient, and do not support large or complex data structures (such as images or audio) very well. So, let's see how to use TFRecords instead.

> **TIP**
>
> If you are happy with CSV files (or whatever other format you are using), you do not *have* to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful

when the bottleneck during training is loading and parsing the data.

The TFRecord Format

The TFRecord format is TensorFlow's preferred format for storing large amounts of data and reading it efficiently. It is a very simple binary format that just contains a sequence of binary records of varying sizes (each record is comprised of a length, a CRC checksum to check that the length was not corrupted, then the actual data, and finally a CRC checksum for the data). You can easily create a TFRecord file using the tf.io.TFRecordWriter class:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

And you can then use a tf.data.TFRecordDataset to read one or more TFRecord files:

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

This will output:

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

> **TIP**
>
> By default, a TFRecordDataset will read files one by one, but you can make it read multiple files in parallel and interleave their records by passing the constructor a list of filepaths and setting num_parallel_reads to a number greater than one. Alternatively, you could obtain the same result by using list_files() and interleave() as we did earlier to read multiple CSV files.

Compressed TFRecord Files

It can sometimes be useful to compress your TFRecord files, especially if they need to be loaded via a network connection. You can create a compressed TFRecord file by setting the options argument:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    f.write(b"Compress, compress, compress!")
```

When reading a compressed TFRecord file, you need to specify the compression type:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

A Brief Introduction to Protocol Buffers

Even though each record can use any binary format you want, TFRecord files usually contain serialized protocol buffers (also called *protobufs*). This is a portable, extensible, and efficient binary format developed at Google back in 2001 and made open source in 2008; protobufs are now widely used, in particular in gRPC, Google's remote procedure call system. They are defined using a simple language that looks like this:

```
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

This protobuf definition says we are using version 3 of the protobuf format, and it specifies that each Person object [4] may (optionally) have a name of type string, an id of type int32, and zero or more email fields, each of type string. The numbers 1, 2, and 3 are the field identifiers: they will be used in each record's binary representation. Once you have a definition in a *.proto* file, you can compile it. This requires protoc, the protobuf compiler, to generate access classes in Python (or some other language). Note that the protobuf definitions you will generally use in TensorFlow have already been compiled for you, and their Python classes are part of the TensorFlow library, so you will not need to use protoc. All you need to know is how to *use* protobuf access classes in Python. To illustrate the basics, let's look at a simple example that uses the access classes generated for the Person protobuf (the code is explained in the comments):

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
```

```
'Al'
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
'a@b.com'
>>> person.email.append("c@d.com")  # add an email address
>>> serialized = person.SerializeToString()  # serialize person to a byte string
>>> serialized
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(serialized)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

In short, we import the Person class generated by protoc, we create an instance and play with it, visualizing it and reading and writing some fields, then we serialize it using the SerializeToString() method. This is the binary data that is ready to be saved or transmitted over the network. When reading or receiving this binary data, we can parse it using the ParseFromString() method, and we get a copy of the object that was serialized. [5]

You could save the serialized Person object to a TFRecord file, then load and parse it: everything would work fine. However, ParseFromString() is not a TensorFlow operation, so you couldn't use it in a preprocessing function in a tf.data pipeline (except by wrapping it in a tf.py_function() operation, which would make the code slower and less portable, as you saw in Chapter 12). However, you could use the tf.io.decode_proto() function, which can parse any protobuf you want, provided you give it the protobuf definition (see the notebook for an example). That said, in practice you will generally want to use instead the predefined protobufs for which TensorFlow provides dedicated parsing operations. Let's look at these predefined protobufs now.

TensorFlow Protobufs

The main protobuf typically used in a TFRecord file is the Example protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats, or a list of integers. Here is the protobuf definition (from TensorFlow's source code):

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

The definitions of BytesList, FloatList, and Int64List are straightforward enough. Note that [packed = true] is used for repeated numerical fields, for a more efficient encoding. A Feature contains either a BytesList, a FloatList, or an Int64List. A Features (with an s) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an Example contains only a Features object.

> **NOTE**
>
> Why was Example even defined, since it contains no more than a Features object? Well, TensorFlow's developers may one day decide to add more fields to it. As long as the new Example definition still contains the features field, with the same ID, it will be backward compatible. This extensibility is one of the great features of protobufs.

Here is how you could create a tf.train.Example representing the same person

as earlier:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

The code is a bit verbose and repetitive, but you could easily wrap it inside a small helper function. Now that we have an Example protobuf, we can serialize it by calling its SerializeToString() method, then write the resulting data to a TFRecord file. Let's write it five times to pretend we have several contacts:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    for _ in range(5):
        f.write(person_example.SerializeToString())
```

Normally you would write much more than five Examples! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an Example protobuf for each instance, serializes them, and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

Now that we have a nice TFRecord file containing several serialized Examples, let's try to load it.

Loading and Parsing Examples

To load the serialized Example protobufs, we will use a tf.data.TFRecordDataset once again, and we will parse each Example using tf.io.parse_single_example(). It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a tf.io.FixedLenFeature descriptor indicating the feature's shape, type, and default value, or a tf.io.VarLenFeature descriptor indicating only the type if the length of the feature's list may vary (such as for the "emails" feature).

The following code defines a description dictionary, then creates a TFRecordDataset and applies a custom preprocessing function to it to parse each serialized Example protobuf that this dataset contains:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

def parse(serialized_example):
    return tf.io.parse_single_example(serialized_example, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse)
for parsed_example in dataset:
    print(parsed_example)
```

The fixed-length features are parsed as regular tensors, but the variable-length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using tf.sparse.to_dense(), but in this case it is simpler to just access its values:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

Instead of parsing examples one by one using tf.io.parse_single_example(), you may want to parse them batch by batch using tf.io.parse_example():

```
def parse(serialized_examples):
    return tf.io.parse_example(serialized_examples, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(2).map(parse)
for parsed_examples in dataset:
    print(parsed_examples)  # two examples at a time
```

Lastly, a BytesList can contain any binary data you want, including any serialized object. For example, you can use tf.io.encode_jpeg() to encode an image using the JPEG format and put this binary data in a BytesList. Later, when your code reads the TFRecord, it will start by parsing the Example, then it will need to call tf.io.decode_jpeg() to parse the data and get the original image (or you can use tf.io.decode_image(), which can decode any BMP, GIF, JPEG, or PNG image). You can also store any tensor you want in a BytesList by serializing the tensor using tf.io.serialize_tensor() then putting the resulting byte string in a BytesList feature. Later, when you parse the TFRecord, you can parse this data using tf.io.parse_tensor(). See this chapter's notebook at *https://homl.info/colab3* for examples of storing images and tensors in a TFRecord file.

As you can see, the Example protobuf is quite flexible, so it will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is represented as a list of words. There may be some contextual data too, such as the document's author, title, and publication date. TensorFlow's SequenceExample protobuf is designed for such use cases.

Handling Lists of Lists Using the SequenceExample Protobuf

Here is the definition of the SequenceExample protobuf:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

A SequenceExample contains a Features object for the contextual data and a FeatureLists object that contains one or more named FeatureList objects (e.g., a FeatureList named "content" and another named "comments"). Each FeatureList contains a list of Feature objects, each of which may be a list of byte strings, a list of 64-bit integers, or a list of floats (in this example, each Feature would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a SequenceExample, serializing it, and parsing it is similar to building, serializing, and parsing an Example, but you must use tf.io.parse_single_sequence_example() to parse a single SequenceExample or tf.io.parse_sequence_example() to parse a batch. Both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the preceding example), you may want to convert them to ragged tensors using tf.RaggedTensor.from_sparse() (see the notebook for the full code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Now that you know how to efficiently store, load, parse, and preprocess the data using the tf.data API, TFRecords, and protobufs, it’s time to turn our attention to the Keras preprocessing layers.

Keras Preprocessing Layers

Preparing your data for a neural network typically requires normalizing the numerical features, encoding the categorical features and text, cropping and resizing images, and more. There are several options for this:

- The preprocessing can be done ahead of time when preparing your training data files, using any tools you like, such as NumPy, Pandas, or Scikit-Learn. You will need to apply the exact same preprocessing steps in production, to ensure your production model receives preprocessed inputs similar to the ones it was trained on.
- Alternatively, you can preprocess your data on the fly while loading it with tf.data, by applying a preprocessing function to every element of a dataset using that dataset's map() method, as we did earlier in this chapter. Again, you will need to apply the same preprocessing steps in production.
- One last approach is to include preprocessing layers directly inside your model so it can preprocess all the input data on the fly during training, then use the same preprocessing layers in production. The rest of this chapter will look at this last approach.

Keras offers many preprocessing layers that you can include in your models: they can be applied to numerical features, categorical features, images, and text. We'll go over the numerical and categorical features in the next sections, as well as basic text preprocessing, and we will cover image preprocessing in Chapter 14 and more advanced text preprocessing in Chapter 16.

The Normalization Layer

As we saw in Chapter 10, Keras provides a Normalization layer that we can use to standardize the input features. We can either specify the mean and variance of each feature when creating the layer or—more simply—pass the training set to the layer's adapt() method before fitting the model, so the layer can measure the feature means and variances on its own before training:

```
norm_layer = tf.keras.layers.Normalization()
model = tf.keras.models.Sequential([
    norm_layer,
    tf.keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
norm_layer.adapt(X_train)  # computes the mean and variance of every feature
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=5)
```

> **TIP**
>
> The data sample passed to the adapt() method must be large enough to be representative of your dataset, but it does not have to be the full training set: for the Normalization layer, a few hundred instances randomly sampled from the training set will generally be sufficient to get a good estimate of the feature means and variances.

Since we included the Normalization layer inside the model, we can now deploy this model to production without having to worry about normalization again: the model will just handle it (see Figure 13-4). Fantastic! This approach completely eliminates the risk of preprocessing mismatch, which happens when people try to maintain different preprocessing code for training and production but update one and forget to update the other. The production model then ends up receiving data preprocessed in a way it doesn't expect. If they're lucky, they get a clear bug. If not, the model's accuracy just silently degrades.

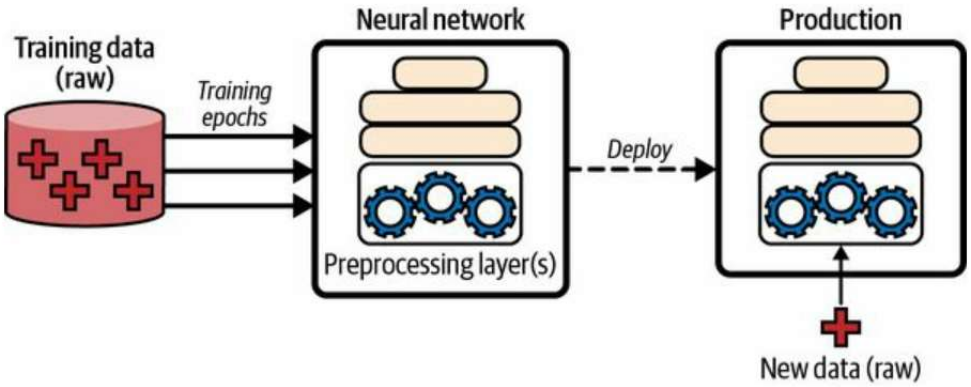

Figure 13-4. Including preprocessing layers inside a model

Including the preprocessing layer directly in the model is nice and straightforward, but it will slow down training (only very slightly in the case of the Normalization layer): indeed, since preprocessing is performed on the fly during training, it happens once per epoch. We can do better by normalizing the whole training set just once before training. To do this, we can use the Normalization layer in a standalone fashion (much like a Scikit-Learn StandardScaler):

```
norm_layer = tf.keras.layers.Normalization()
norm_layer.adapt(X_train)
X_train_scaled = norm_layer(X_train)
X_valid_scaled = norm_layer(X_valid)
```

Now we can train a model on the scaled data, this time without a Normalization layer:

```
model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
model.fit(X_train_scaled, y_train, epochs=5,
          validation_data=(X_valid_scaled, y_valid))
```

Good! This should speed up training a bit. But now the model won’t preprocess its inputs when we deploy it to production. To fix this, we just need to create a new model that wraps both the adapted Normalization layer and the model we just trained. We can then deploy this final model to

production, and it will take care of both preprocessing its inputs and making predictions (see Figure 13-5):

```
final_model = tf.keras.Sequential([norm_layer, model])
X_new = X_test[:3]  # pretend we have a few new instances (unscaled)
y_pred = final_model(X_new)  # preprocesses the data and makes predictions
```

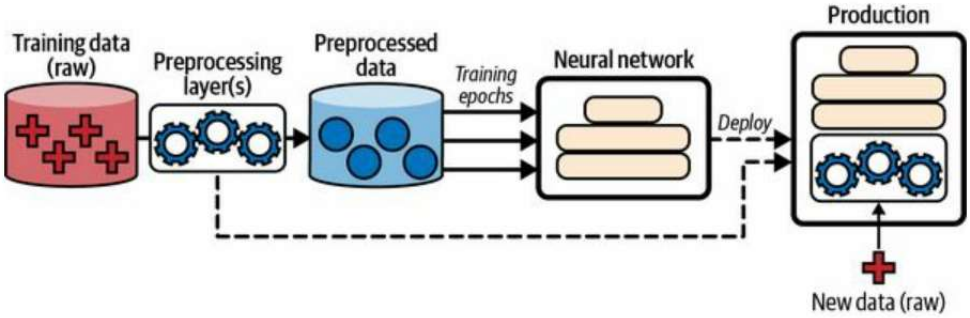

Figure 13-5. Preprocessing the data just once before training using preprocessing layers, then deploying these layers inside the final model

Now we have the best of both worlds: training is fast because we only preprocess the data once before training begins, and the final model can preprocess its inputs on the fly without any risk of preprocessing mismatch.

Moreover, the Keras preprocessing layers play nicely with the tf.data API. For example, it's possible to pass a tf.data.Dataset to a preprocessing layer's adapt() method. It's also possible to apply a Keras preprocessing layer to a tf.data.Dataset using the dataset's map() method. For example, here's how you could apply an adapted Normalization layer to the input features of each batch in a dataset:

```
dataset = dataset.map(lambda X, y: (norm_layer(X), y))
```

Lastly, if you ever need more features than the Keras preprocessing layers provide, you can always write your own Keras layer, just like we discussed in Chapter 12. For example, if the Normalization layer didn't exist, you could get a similar result using the following custom layer:

```
import numpy as np
```

```
class MyNormalization(tf.keras.layers.Layer):
    def adapt(self, X):
        self.mean_ = np.mean(X, axis=0, keepdims=True)
        self.std_ = np.std(X, axis=0, keepdims=True)

    def call(self, inputs):
        eps = tf.keras.backend.epsilon()  # a small smoothing term
        return (inputs - self.mean_) / (self.std_ + eps)
```

Next, let’s look at another Keras preprocessing layer for numerical features: the Discretization layer.

The Discretization Layer

The Discretization layer's goal is to transform a numerical feature into a categorical feature by mapping value ranges (called bins) to categories. This is sometimes useful for features with multimodal distributions, or with features that have a highly non-linear relationship with the target. For example, the following code maps a numerical age feature to three categories, less than 18, 18 to 50 (not included), and 50 or over:

```
>>> age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])
>>> discretize_layer = tf.keras.layers.Discretization(bin_boundaries=[18., 50.])
>>> age_categories = discretize_layer(age)
>>> age_categories
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[0],[2],[2],[1],[1],[0]])>
```

In this example, we provided the desired bin boundaries. If you prefer, you can instead provide the number of bins you want, then call the layer's adapt() method to let it find the appropriate bin boundaries based on the value percentiles. For example, if we set num_bins=3, then the bin boundaries will be located at the values just below the 33rd and 66th percentiles (in this example, at the values 10 and 37):

```
>>> discretize_layer = tf.keras.layers.Discretization(num_bins=3)
>>> discretize_layer.adapt(age)
>>> age_categories = discretize_layer(age)
>>> age_categories
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[1],[2],[2],[1],[2],[0]])>
```

Category identifiers such as these should generally not be passed directly to a neural network, as their values cannot be meaningfully compared. Instead, they should be encoded, for example using one-hot encoding. Let's look at how to do this now.

The CategoryEncoding Layer

When there are only a few categories (e.g., less than a dozen or two), then one-hot encoding is often a good option (as discussed in Chapter 2). To do this, Keras provides the CategoryEncoding layer. For example, let's one-hot encode the age_cate gories feature we just created:

```
>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)
>>> onehot_layer(age_categories)
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]], dtype=float32)>
```

If you try to encode more than one categorical feature at a time (which only makes sense if they all use the same categories), the CategoryEncoding class will perform *multi-hot encoding* by default: the output tensor will contain a 1 for each category present in *any* input feature. For example:

```
>>> two_age_categories = np.array([[1, 0], [2, 2], [2, 0]])
>>> onehot_layer(two_age_categories)
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 0.],
       [0., 0., 1.],
       [1., 0., 1.]], dtype=float32)>
```

If you believe it's useful to know how many times each category occurred, you can set output_mode="count" when creating the CategoryEncoding layer, in which case the output tensor will contain the number of occurrences of each category. In the preceding example, the output would be the same except for the second row, which would become [0., 0., 2.].

Note that both multi-hot encoding and count encoding lose information, since it's not possible to know which feature each active category came from. For example, both [0, 1] and [1, 0] are encoded as [1., 1., 0.]. If you want to avoid

this, then you need to one-hot encode each feature separately and concatenate the outputs. This way, [0, 1] would get encoded as [1., 0., 0., 0., 1., 0.] and [1, 0] would get encoded as [0., 1., 0., 1., 0., 0.]. You can get the same result by tweaking the category identifiers so they don't overlap. For example:

```
>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3 + 3)
>>> onehot_layer(two_age_categories + [0, 3])  # adds 3 to the second feature
<tf.Tensor: shape=(3, 6), dtype=float32, numpy=
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0.]], dtype=float32)>
```

In this output, the first three columns correspond to the first feature, and the last three correspond to the second feature. This allows the model to distinguish the two features. However, it also increases the number of features fed to the model, and thereby requires more model parameters. It's hard to know in advance whether a single multi-hot encoding or a per-feature one-hot encoding will work best: it depends on the task, and you may need to test both options.

Now you can encode categorical integer features using one-hot or multi-hot encoding. But what about categorical text features? For this, you can use the StringLookup layer.

The StringLookup Layer

Let’s use the Keras StringLookup layer to one-hot encode a cities feature:

```
>>> cities = ["Auckland", "Paris", "Paris", "San Francisco"]
>>> str_lookup_layer = tf.keras.layers.StringLookup()
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[1], [3], [3], [0]])>
```

We first create a StringLookup layer, then we adapt it to the data: it finds that there are three distinct categories. Then we use the layer to encode a few cities. They are encoded as integers by default. Unknown categories get mapped to 0, as is the case for “Montreal” in this example. The known categories are numbered starting at 1, from the most frequent category to the least frequent.

Conveniently, if you set output_mode="one_hot" when creating the StringLookup layer, it will output a one-hot vector for each category, instead of an integer:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]], dtype=float32)>
```

> **TIP**
>
> Keras also includes an IntegerLookup layer that acts much like the StringLookup layer but takes integers as input, rather than strings.

If the training set is very large, it may be convenient to adapt the layer to just a random subset of the training set. In this case, the layer’s adapt() method

may miss some of the rarer categories. By default, it would then map them all to category 0, making them indistinguishable by the model. To reduce this risk (while still adapting the layer only on a subset of the training set), you can set num_oov_indices to an integer greater than 1. This is the number of out-of-vocabulary (OOV) buckets to use: each unknown category will get mapped pseudorandomly to one of the OOV buckets, using a hash function modulo the number of OOV buckets. This will allow the model to distinguish at least some of the rare categories. For example:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(num_oov_indices=5)
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Foo"], ["Bar"], ["Baz"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[5], [7], [4], [3], [4]])>
```

Since there are five OOV buckets, the first known category's ID is now 5 ("Paris"). But "Foo", "Bar", and "Baz" are unknown, so they each get mapped to one of the OOV buckets. "Bar" gets its own dedicated bucket (with ID 3), but sadly "Foo" and "Baz" happen to be mapped to the same bucket (with ID 4), so they remain indistinguishable by the model. This is called a *hashing collision*. The only way to reduce the risk of collision is to increase the number of OOV buckets. However, this will also increase the total number of categories, which will require more RAM and extra model parameters once the categories are one-hot encoded. So, don't increase that number too much.

This idea of mapping categories pseudorandomly to buckets is called the *hashing trick*. Keras provides a dedicated layer which does just that: the Hashing layer.

The Hashing Layer

For each category, the Keras Hashing layer computes a hash, modulo the number of buckets (or “bins”). The mapping is entirely pseudorandom, but stable across runs and platforms (i.e., the same category will always be mapped to the same integer, as long as the number of bins is unchanged). For example, let’s use the Hashing layer to encode a few cities:

```
>>> hashing_layer = tf.keras.layers.Hashing(num_bins=10)
>>> hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[0], [1], [9], [1]])>
```

The benefit of this layer is that it does not need to be adapted at all, which may sometimes be useful, especially in an out-of-core setting (when the dataset is too large to fit in memory). However, we once again get a hashing collision: “Tokyo” and “Montreal” are mapped to the same ID, making them indistinguishable by the model. So, it’s usually preferable to stick to the StringLookup layer.

Let’s now look at another way to encode categories: trainable embeddings.

Encoding Categorical Features Using Embeddings

An embedding is a dense representation of some higher-dimensional data, such as a category, or a word in a vocabulary. If there are 50,000 possible categories, then one-hot encoding would produce a 50,000-dimensional sparse vector (i.e., containing mostly zeros). In contrast, an embedding would be a comparatively small dense vector; for example, with just 100 dimensions.

In deep learning, embeddings are usually initialized randomly, and they are then trained by gradient descent, along with the other model parameters. For example, the "NEAR BAY" category in the California housing dataset could be represented initially by a random vector such as [0.131, 0.890], while the "NEAR OCEAN" category might be represented by another random vector such as [0.631, 0.791]. In this example, we use 2D embeddings, but the number of dimensions is a hyperparameter you can tweak.

Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories in this case, gradient descent will certainly end up pushing them closer together, while it will tend to move them away from the "INLAND" category's embedding (see Figure 13-6). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories. This is called *representation learning* (you will see other types of representation learning in Chapter 17).

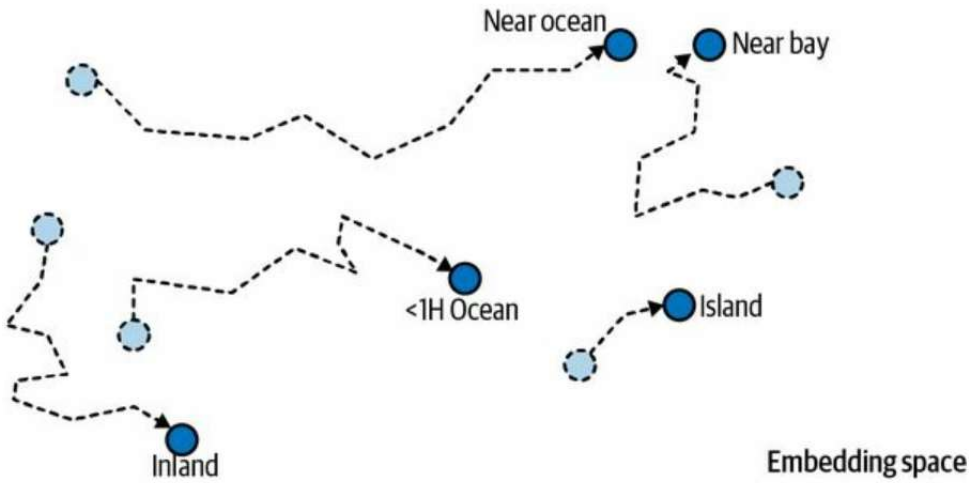

Figure 13-6. Embeddings will gradually improve during training

WORD EMBEDDINGS

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own.

The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks. But things really took off in 2013, when Tomáš Mikolov and other Google researchers published a paper[6] describing an efficient technique to learn word embeddings using neural networks, significantly outperforming previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a neural network to predict the words near any given word and obtained astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as *France*, *Spain*, and *Italy* ended up clustered together.

It’s not just about proximity, though: word embeddings were also

organized along meaningful axes in the embedding space. Here is a famous example: if you compute *King – Man + Woman* (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word *Queen* (see Figure 13-7). In other words, the word embeddings encode the concept of gender! Similarly, you can compute *Madrid – Spain + France*, and the result is close to *Paris*, which seems to show that the notion of capital city was also encoded in the embeddings.

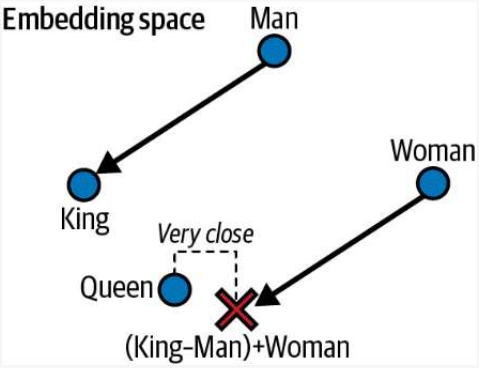

Figure 13-7. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Unfortunately, word embeddings sometimes capture our worst biases. For example, although they correctly learn that *Man is to King as Woman is to Queen*, they also seem to learn that *Man is to Doctor as Woman is to Nurse*: quite a sexist bias! To be fair, this particular example is probably exaggerated, as was pointed out in a 2019 paper[7] by Malvina Nissim et al. Nevertheless, ensuring fairness in deep learning algorithms is an important and active research topic.

Keras provides an Embedding layer, which wraps an *embedding matrix*: this matrix has one row per category and one column per embedding dimension. By default, it is initialized randomly. To convert a category ID to an embedding, the Embedding layer just looks up and returns the row that corresponds to that category. That's all there is to it! For example, let's initialize an Embedding layer with five rows and 2D embeddings, and use it

to encode some categories:

```
>>> tf.random.set_seed(42)
>>> embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)
>>> embedding_layer(np.array([2, 4, 2]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.04663396,  0.01846724],
       [-0.02736737, -0.02768031],
       [-0.04663396,  0.01846724]], dtype=float32)>
```

As you can see, category 2 gets encoded (twice) as the 2D vector [-0.04663396, 0.01846724], while category 4 gets encoded as [-0.02736737, -0.02768031]. Since the layer is not trained yet, these encodings are just random.

> **WARNING**
>
> An Embedding layer is initialized randomly, so it does not make sense to use it outside of a model as a standalone preprocessing layer unless you initialize it with pretrained weights.

If you want to embed a categorical text attribute, you can simply chain a StringLookup layer and an Embedding layer, like this:

```
>>> tf.random.set_seed(42)
>>> ocean_prox = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
>>> str_lookup_layer = tf.keras.layers.StringLookup()
>>> str_lookup_layer.adapt(ocean_prox)
>>> lookup_and_embed = tf.keras.Sequential([
...     str_lookup_layer,
...     tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),
...                               output_dim=2)
... ])
...
>>> lookup_and_embed(np.array([["<1H OCEAN"], ["ISLAND"], ["<1H OCEAN"]]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.01896119,  0.02223358],
       [ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358]], dtype=float32)>
```

Note that the number of rows in the embedding matrix needs to be equal to the vocabulary size: that's the total number of categories, including the known categories plus the OOV buckets (just one by default). The vocabulary_size() method of the StringLookup class conveniently returns this number.

> **TIP**
>
> In this example we used 2D embeddings, but as a rule of thumb embeddings typically have 10 to 300 dimensions, depending on the task, the vocabulary size, and the size of your training set. You will have to tune this hyperparameter.

Putting everything together, we can now create a Keras model that can process a categorical text feature along with regular numerical features and learn an embedding for each category (as well as for each OOV bucket):

```
X_train_num, X_train_cat, y_train = [...]  # load the training set
X_valid_num, X_valid_cat, y_valid = [...]  # and the validation set

num_input = tf.keras.layers.Input(shape=[8], name="num")
cat_input = tf.keras.layers.Input(shape=[], dtype=tf.string, name="cat")
cat_embeddings = lookup_and_embed(cat_input)
encoded_inputs = tf.keras.layers.concatenate([num_input, cat_embeddings])
outputs = tf.keras.layers.Dense(1)(encoded_inputs)
model = tf.keras.models.Model(inputs=[num_input, cat_input], outputs=[outputs])
model.compile(loss="mse", optimizer="sgd")
history = model.fit((X_train_num, X_train_cat), y_train, epochs=5,
                    validation_data=((X_valid_num, X_valid_cat), y_valid))
```

This model takes two inputs: num_input, which contains eight numerical features per instance, plus cat_input, which contains a single categorical text input per instance. The model uses the lookup_and_embed model we created earlier to encode each ocean-proximity category as the corresponding trainable embedding. Next, it concatenates the numerical inputs and the embeddings using the concatenate() function to produce the complete encoded inputs, which are ready to be fed to a neural network. We could add any kind of neural network at this point, but for simplicity we just add a

single dense output layer, and then we create the Keras Model with the inputs and output we've just defined. Next we compile the model and train it, passing both the numerical and categorical inputs.

As you saw in Chapter 10, since the Input layers are named "num" and "cat", we could also have passed the training data to the fit() method using a dictionary instead of a tuple: {"num": X_train_num, "cat": X_train_cat}. Alternatively, we could have passed a tf.data.Dataset containing batches, each represented as ((X_batch_num, X_batch_cat), y_batch) or as ({"num": X_batch_num, "cat": X_batch_cat}, y_batch). And of course the same goes for the validation data.

> **NOTE**
>
> One-hot encoding followed by a Dense layer (with no activation function and no biases) is equivalent to an Embedding layer. However, the Embedding layer uses way fewer computations as it avoids many multiplications by zero—the performance difference becomes clear when the size of the embedding matrix grows. The Dense layer's weight matrix plays the role of the embedding matrix. For example, using one-hot vectors of size 20 and a Dense layer with 10 units is equivalent to using an Embedding layer with input_dim=20 and output_dim=10. As a result, it would be wasteful to use more embedding dimensions than the number of units in the layer that follows the Embedding layer.

OK, now that you have learned how to encode categorical features, it's time to turn our attention to text preprocessing.

Text Preprocessing

Keras provides a TextVectorization layer for basic text preprocessing. Much like the StringLookup layer, you must either pass it a vocabulary upon creation, or let it learn the vocabulary from some training data using the adapt() method. Let's look at an example:

```
>>> train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]
>>> text_vec_layer = tf.keras.layers.TextVectorization()
>>> text_vec_layer.adapt(train_data)
>>> text_vec_layer(["Be good!", "Question: be or be?"])
<tf.Tensor: shape=(2, 4), dtype=int64, numpy=
array([[2, 1, 0, 0],
       [6, 2, 1, 2]])>
```

The two sentences "Be good!" and "Question: be or be?" were encoded as [2, 1, 0, 0] and [6, 2, 1, 2], respectively. The vocabulary was learned from the four sentences in the training data: "be" = 2, "to" = 3, etc. To construct the vocabulary, the adapt() method first converted the training sentences to lowercase and removed punctuation, which is why "Be", "be", and "be?" are all encoded as "be" = 2. Next, the sentences were split on whitespace, and the resulting words were sorted by descending frequency, producing the final vocabulary. When encoding sentences, unknown words get encoded as 1s. Lastly, since the first sentence is shorter than the second, it was padded with 0s.

> **TIP**
>
> The TextVectorization layer has many options. For example, you can preserve the case and punctuation if you want, by setting standardize=None, or you can pass any standardization function you please as the standardize argument. You can prevent splitting by setting split=None, or you can pass your own splitting function instead. You can set the output_sequence_length argument to ensure that the output sequences all get cropped or padded to the desired length, or you can set ragged=True to get a ragged tensor instead of a regular tensor. Please check out the documentation for more options.

The word IDs must be encoded, typically using an Embedding layer: we will do this in Chapter 16. Alternatively, you can set the TextVectorization layer's output_mode argument to "multi_hot" or "count" to get the corresponding encodings. However, simply counting words is usually not ideal: words like "to" and "the" are so frequent that they hardly matter at all, whereas, rarer words such as "basketball" are much more informative. So, rather than setting output_mode to "multi_hot" or "count", it is usually preferable to set it to "tf_idf", which stands for *term-frequency* × *inverse-document-frequency* (TF-IDF). This is similar to the count encoding, but words that occur frequently in the training data are downweighted, and conversely, rare words are upweighted. For example:

```
>>> text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")
>>> text_vec_layer.adapt(train_data)
>>> text_vec_layer(["Be good!", "Question: be or be?"])
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472 , 0. , 0. , 0. , 0.        ],
       [0.96725637, 1.3862944 , 0. , 0. , 0. , 1.0986123 ]], dtype=float32)>
```

There are many TF-IDF variants, but the way the TextVectorization layer implements it is by multiplying each word count by a weight equal to $\log(1 + d / (f + 1))$, where d is the total number of sentences (a.k.a., documents) in the training data and f counts how many of these training sentences contain the given word. For example, in this case there are $d = 4$ sentences in the training data, and the word "be" appears in $f = 3$ of these. Since the word "be" occurs twice in the sentence "Question: be or be?", it gets encoded as $2 \times \log(1 + 4 / (1 + 3)) \approx 1.3862944$. The word "question" only appears once, but since it is a less common word, its encoding is almost as high: $1 \times \log(1 + 4 / (1 + 1)) \approx 1.0986123$. Note that the average weight is used for unknown words.

This approach to text encoding is straightforward to use and it can give fairly good results for basic natural language processing tasks, but it has several important limitations: it only works with languages that separate words with spaces, it doesn't distinguish between homonyms (e.g., "to bear" versus "teddy bear"), it gives no hint to your model that words like "evolution" and "evolutionary" are related, etc. And if you use multi-hot, count, or TF-IDF

encoding, then the order of the words is lost. So what are the other options?

One option is to use the TensorFlow Text library, which provides more advanced text preprocessing features than the TextVectorization layer. For example, it includes several subword tokenizers capable of splitting the text into tokens smaller than words, which makes it possible for the model to more easily detect that “evolution” and “evolutionary” have something in common (more on subword tokenization in Chapter 16).

Yet another option is to use pretrained language model components. Let’s look at this now.

Using Pretrained Language Model Components

The TensorFlow Hub library makes it easy to reuse pretrained model components in your own models, for text, image, audio, and more. These model components are called *modules*. Simply browse the TF Hub repository, find the one you need, and copy the code example into your project, and the module will be automatically downloaded and bundled into a Keras layer that you can directly include in your model. Modules typically contain both preprocessing code and pretrained weights, and they generally require no extra training (but of course, the rest of your model will certainly require training).

For example, some powerful pretrained language models are available. The most powerful are quite large (several gigabytes), so for a quick example let's use the nnlm-en-dim50 module, version 2, which is a fairly basic module that takes raw text as input and outputs 50-dimensional sentence embeddings. We'll import TensorFlow Hub and use it to load the module, then use that module to encode two sentences to vectors:[8]

```
>>> import tensorflow_hub as hub
>>> hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2")
>>> sentence_embeddings = hub_layer(tf.constant(["To be", "Not to be"]))
>>> sentence_embeddings.numpy().round(2)
array([[-0.25,  0.28,  0.01,  0.1 ,  [...] ,  0.05,  0.31],
       [-0.2 ,  0.2 , -0.08,  0.02,  [...] , -0.04,  0.15]], dtype=float32)
```

The hub.KerasLayer layer downloads the module from the given URL. This particular module is a *sentence encoder*: it takes strings as input and encodes each one as a single vector (in this case, a 50-dimensional vector). Internally, it parses the string (splitting words on spaces) and embeds each word using an embedding matrix that was pretrained on a huge corpus: the Google News 7B corpus (seven billion words long!). Then it computes the mean of all the word embeddings, and the result is the sentence embedding. [9]

You just need to include this hub_layer in your model, and you're ready to go. Note that this particular language model was trained on the English

language, but many other languages are available, as well as multilingual models.

Last but not least, the excellent open source Transformers library by Hugging Face also makes it easy to include powerful language model components inside your own models. You can browse the Hugging Face Hub, choose the model you want, and use the provided code examples to get started. It used to contain only language models, but it has now expanded to include image models and more.

We will come back to natural language processing in more depth in Chapter 16. Let's now look at Keras's image preprocessing layers.

Image Preprocessing Layers

The Keras preprocessing API includes three image preprocessing layers:

- tf.keras.layers.Resizing resizes the input images to the desired size. For example, Resizing(height=100, width=200) resizes each image to 100 × 200, possibly distorting the image. If you set crop_to_aspect_ratio=True, then the image will be cropped to the target image ratio, to avoid distortion.
- tf.keras.layers.Rescaling rescales the pixel values. For example, Rescal ing(scale=2/255, offset=-1) scales the values from 0 → 255 to –1 → 1.
- tf.keras.layers.CenterCrop crops the image, keeping only a center patch of the desired height and width.

For example, let's load a couple of sample images and center-crop them. For this, we will use Scikit-Learn's load_sample_images() function; this loads two color images, one of a Chinese temple and the other of a flower (this requires the Pillow library, which should already be installed if you are using Colab or if you followed the installation instructions):

```
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]
crop_image_layer = tf.keras.layers.CenterCrop(height=100, width=100)
cropped_images = crop_image_layer(images)
```

Keras also includes several layers for data augmentation, such as RandomCrop, RandomFlip, RandomTranslation, RandomRotation, RandomZoom, RandomHeight, RandomWidth, and RandomContrast. These layers are only active during training, and they randomly apply some transformation to the input images (their names are self-explanatory). Data augmentation will artificially increase the size of the training set, which often leads to improved performance, as long as the transformed images look like realistic (nonaugmented) images. We'll cover image processing more closely

in the next chapter.

> **NOTE**
>
> Under the hood, the Keras preprocessing layers are based on TensorFlow's low-level API. For example, the Normalization layer uses tf.nn.moments() to compute both the mean and variance, the Discretization layer uses tf.raw_ops.Bucketize(), CategoricalEncoding uses tf.math.bincount(), IntegerLookup and StringLookup use the tf.lookup package, Hashing and TextVectorization use several ops from the tf.strings package, Embedding uses tf.nn.embedding_lookup(), and the image preprocessing layers use the ops from the tf.image package. If the Keras preprocessing API isn't sufficient for your needs, you may occasionally need to use TensorFlow's low-level API directly.

Now let's look at another way to load data easily and efficiently in TensorFlow.

The TensorFlow Datasets Project

The TensorFlow Datasets (TFDS) project makes it very easy to load common datasets, from small ones like MNIST or Fashion MNIST to huge datasets like ImageNet (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, time series, and much more. You can visit *https://homl.info/tfds* to view the full list, along with a description of each dataset. You can also check out Know Your Data, which is a tool to explore and understand many of the datasets provided by TFDS.

TFDS is not bundled with TensorFlow, but if you are running on Colab or if you followed the installation instructions at *https://homl.info/install*, then it's already installed. You can then import tensorflow_datasets, usually as tfds, then call the tfds.load() function, which will download the data you want (unless it was already downloaded earlier) and return the data as a dictionary of datasets (typically one for training and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds

datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]
```

You can then apply any transformation you want (typically shuffling, batching, and prefetching), and you're ready to train your model. Here is a simple example:

```
for batch in mnist_train.shuffle(10_000, seed=42).batch(32).prefetch(1):
    images = batch["image"]
    labels = batch["label"]
    # [...] do something with the images and labels
```

> **TIP**
>
> The load() function can shuffle the files it downloads: just set shuffle_files=True.

However, this may be insufficient, so it's best to shuffle the training data some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing two elements (again, the features and the labels). You could transform the dataset using the map() method, like this:

```
mnist_train = mnist_train.shuffle(buffer_size=10_000, seed=42).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

But it's simpler to ask the load() function to do this for you by setting as_supervised=True (obviously this works only for labeled datasets).

Lastly, TFDS provides a convenient way to split the data using the split argument. For example, if you want to use the first 90% of the training set for training, the remaining 10% for validation, and the whole test set for testing, then you can set split=["train[:90%]", "train[90%:]", "test"]. The load() function will return all three sets. Here is a complete example, loading and splitting the MNIST dataset using TFDS, then using these sets to train and evaluate a simple Keras model:

```
train_set, valid_set, test_set = tfds.load(
    name="mnist",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
train_set = train_set.shuffle(buffer_size=10_000, seed=42).batch(32).prefetch(1)
valid_set = valid_set.batch(32).cache()
test_set = test_set.batch(32).cache()
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=5)
test_loss, test_accuracy = model.evaluate(test_set)
```

Congratulations, you've reached the end of this quite technical chapter! You may feel that it is a bit far from the abstract beauty of neural networks, but the fact is deep learning often involves large amounts of data, and knowing how to load, parse, and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at convolutional neural networks, which are among the most successful neural net architectures for image processing and many other applications.

Exercises

1. Why would you want to use the tf.data API?
2. What are the benefits of splitting a large dataset into multiple files?
3. During training, how can you tell that your input pipeline is the bottleneck? What can you do to fix it?
4. Can you save any binary data to a TFRecord file, or only serialized protocol buffers?
5. Why would you go through the hassle of converting all your data to the Example protobuf format? Why not use your own protobuf definition?
6. When using TFRecords, when would you want to activate compression? Why not do it systematically?
7. Data can be preprocessed directly when writing the data files, or within the tf.data pipeline, or in preprocessing layers within your model. Can you list a few pros and cons of each option?
8. Name a few common ways you can encode categorical integer features. What about text?
9. Load the Fashion MNIST dataset (introduced in Chapter 10); split it into a training set, a validation set, and a test set; shuffle the training set; and save each dataset to multiple TFRecord files. Each record should be a serialized Example protobuf with two features: the serialized image (use tf.io.serialize_tensor() to serialize each image), and the label. [10] Then use tf.data to create an efficient dataset for each set. Finally, use a Keras model to train these datasets, including a preprocessing layer to standardize each input feature. Try to make the input pipeline as efficient as possible, using TensorBoard to visualize profiling data.
10. In this exercise you will download a dataset, split it, create a tf.data.Dataset to load it and preprocess it efficiently, then build and

train a binary classification model containing an Embedding layer:

a. Download the Large Movie Review Dataset, which contains 50,000 movie reviews from the Internet Movie Database (IMDb). The data is organized in two directories, *train* and *test*, each containing a *pos* subdirectory with 12,500 positive reviews and a *neg* subdirectory with 12,500 negative reviews. Each review is stored in a separate text file. There are other files and folders (including preprocessed bag-of-words versions), but we will ignore them in this exercise.

b. Split the test set into a validation set (15,000) and a test set (10,000).

c. Use tf.data to create an efficient dataset for each set.

d. Create a binary classification model, using a TextVectorization layer to preprocess each review.

e. Add an Embedding layer and compute the mean embedding for each review, multiplied by the square root of the number of words (see Chapter 16). This rescaled mean embedding can then be passed to the rest of your model.

f. Train the model and see what accuracy you get. Try to optimize your pipelines to make training as fast as possible.

g. Use TFDS to load the same dataset more easily: tfds.load("imdb_reviews").

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

---

1 Imagine a sorted deck of cards on your left: suppose you just take the top three cards and shuffle them, then pick one randomly and put it to your right, keeping the other two in your hands. Take another card on your left, shuffle the three cards in your hands and pick one of them randomly, and put it on your right. When you are done going through all the cards like this, you will have a deck of cards on your right: do you think it will be perfectly shuffled?

2 In general, just prefetching one batch is fine, but in some cases you may need to prefetch a few

more. Alternatively, you can let TensorFlow decide automatically by passing tf.data.AUTOTUNE to prefetch().

3 But check out the experimental tf.data.experimental.prefetch_to_device() function, which can prefetch data directly to the GPU. Any TensorFlow function or class with experimental in its name may change without warning in future versions. If an experimental function fails, try removing the word experimental: it may have been moved to the core API. If not, then please check the notebook, as I will ensure it contains up-to-date code.

4 Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

5 This chapter contains the bare minimum you need to know about protobufs to use TFRecords. To learn more about protobufs, please visit *https://homl.info/protobuf*.

6 Tomáš Mikolov et al., "Distributed Representations of Words and Phrases and Their Compositionality", *Proceedings of the 26th International Conference on Neural Information Processing Systems* 2 (2013): 3111–3119.

7 Malvina Nissim et al., "Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor", arXiv preprint arXiv:1905.09866 (2019).

8 TensorFlow Hub is not bundled with TensorFlow, but if you are running on Colab or if you followed the installation instructions at *https://homl.info/install*, then it's already installed.

9 To be precise, the sentence embedding is equal to the mean word embedding multiplied by the square root of the number of words in the sentence. This compensates for the fact that the mean of *n* random vectors gets shorter as *n* grows.

10 For large images, you could use tf.io.encode_jpeg() instead. This would save a lot of space, but it would lose a bit of image quality.

Chapter 14. 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 10 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 Keras. 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 14-1, 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.

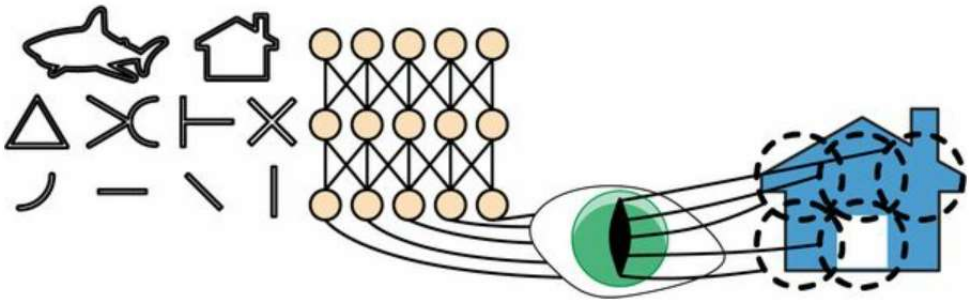

Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields

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 14-1, 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., 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 14-2). 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 common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

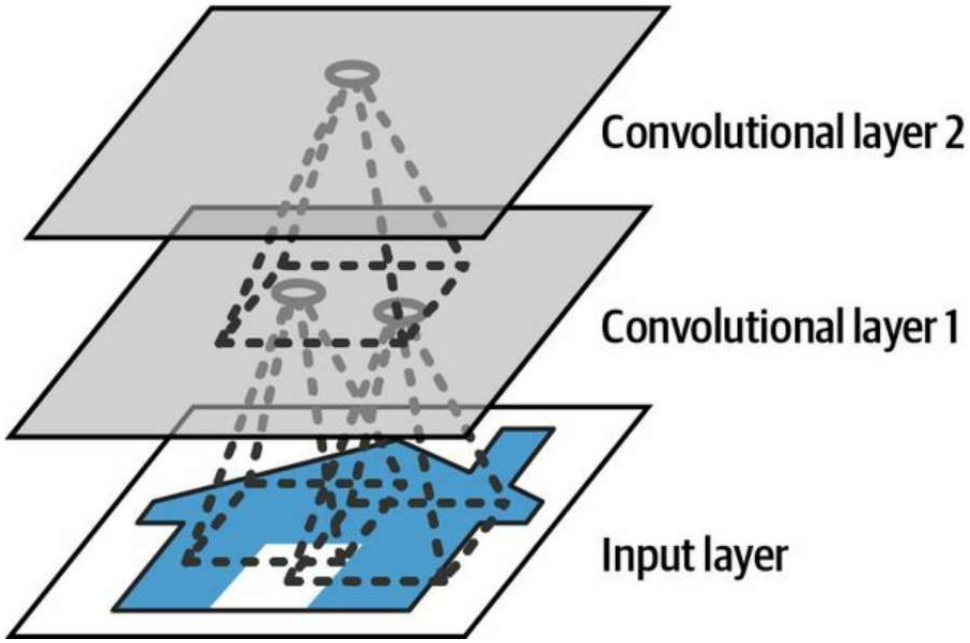

Figure 14-2. CNN layers with rectangular local receptive fields

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 14-3). 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 14-4. 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, but it does not have to be so). 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.

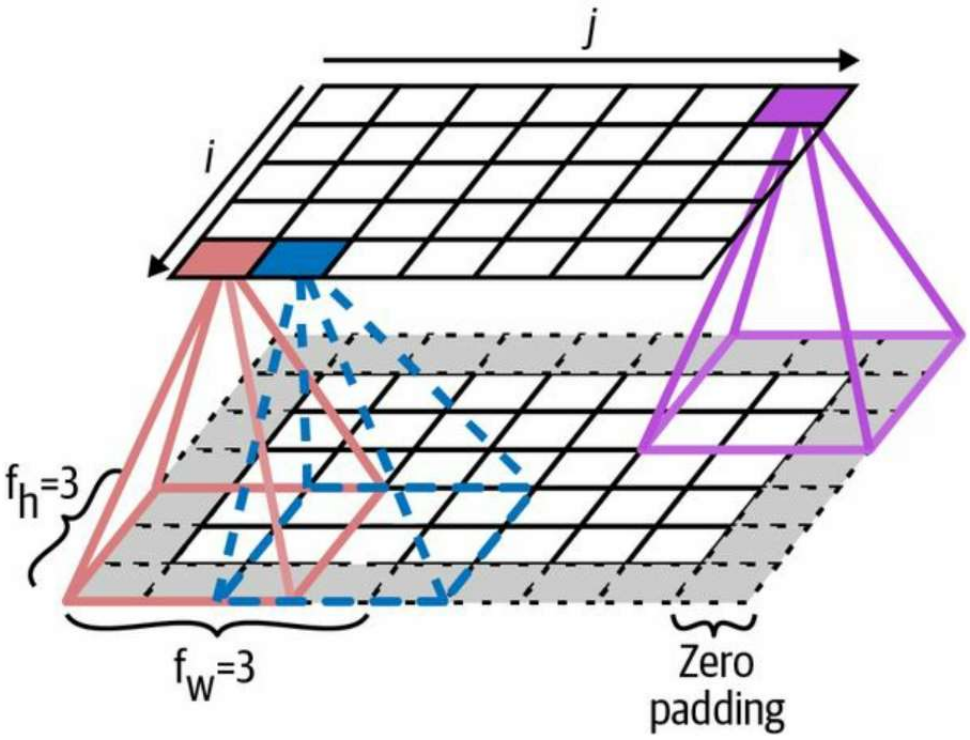

Figure 14-3. Connections between layers and zero padding

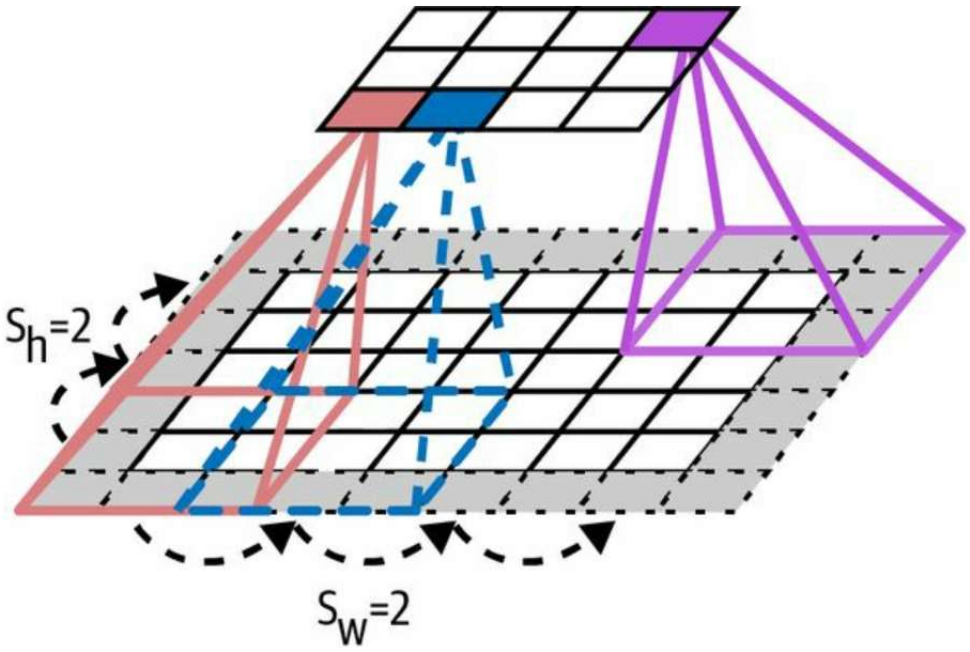

Figure 14-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 14-5 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.

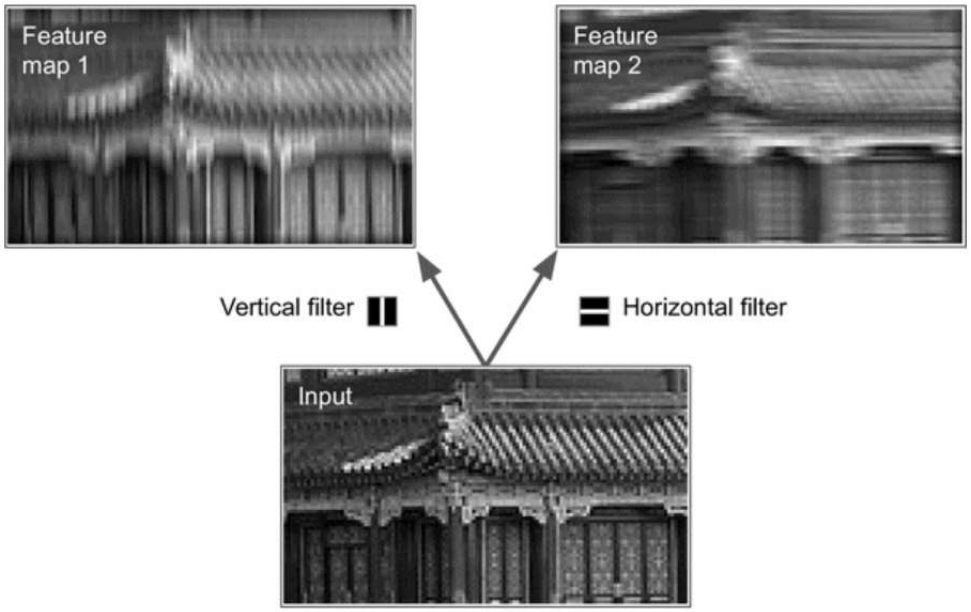

Figure 14-5. Applying two different filters to get two feature maps

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 14-5 (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.

Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D (see Figure 14-6). It has 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.

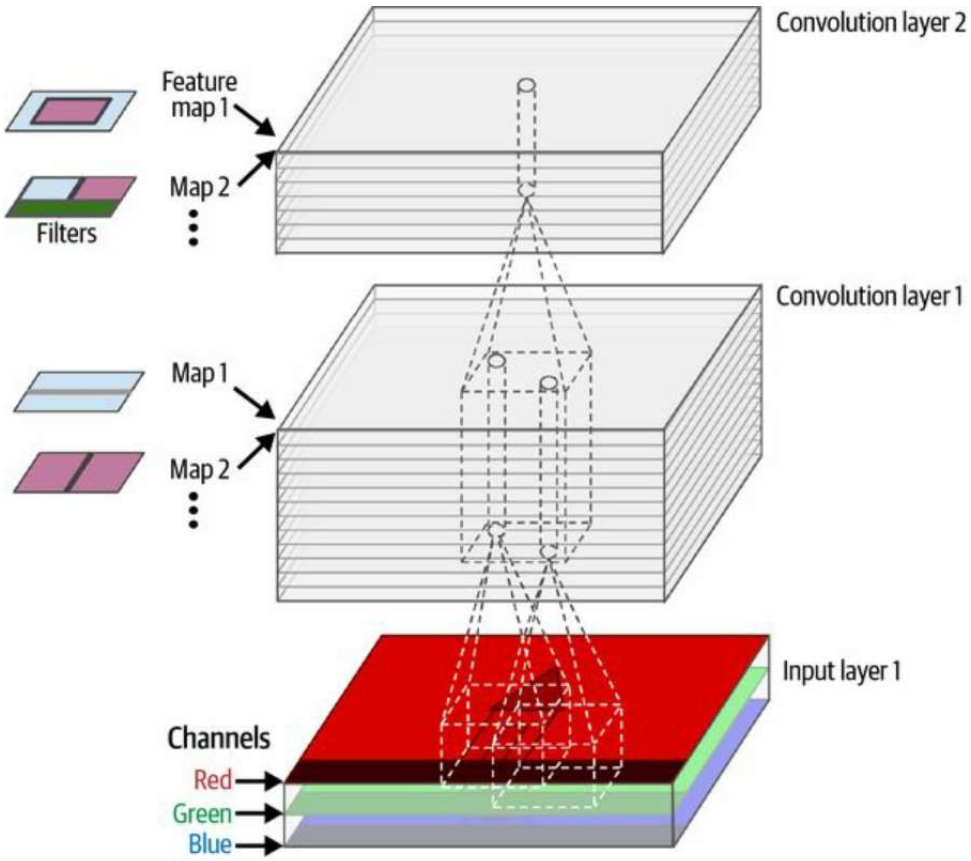

Figure 14-6. Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels; each convolutional layer outputs one feature map per filter

> **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 9, 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 14-1 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 14-1. Computing the output of a neuron in a convolutional layer

z i,j,k = b k + ∑ u=0 f h -1 ∑ v=0 f w -1 ∑ k'=0 f n ' -1 x i ' ,j ' ,k ' × w u,v,k ' ,k with i ' = i × s h + u j ' = j × s w + v

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 Keras.

Implementing Convolutional Layers with Keras

First, let's load and preprocess a couple of sample images, using Scikit-Learn's load_sample_image() function and Keras's CenterCrop and Rescaling layers (all of which were introduced in Chapter 13):

```
from sklearn.datasets import load_sample_images
import tensorflow as tf

images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

Let's look at the shape of the images tensor:

```
>>> images.shape
TensorShape([2, 70, 120, 3])
```

Yikes, it's a 4D tensor; we haven't seen this before! What do all these dimensions mean? Well, there are two sample images, which explains the first dimension. Then each image is 70 × 120, since that's the size we specified when creating the CenterCrop layer (the original images were 427 × 640). This explains the second and third dimensions. And lastly, each pixel holds one value per color channel, and there are three of them—red, green, and blue—which explains the last dimension.

Now let's create a 2D convolutional layer and feed it these images to see what comes out. For this, Keras provides a Convolution2D layer, alias Conv2D. Under the hood, this layer relies on TensorFlow's tf.nn.conv2d() operation. 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:

```
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
fmaps = conv_layer(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 (last dimension).

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

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

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 filters=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 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**
>
> The default option is surprisingly named padding="valid", which actually means no zero-padding at all! This name comes from the fact that in this case every neuron's receptive field lies strictly within *valid* positions inside the input (it does not go out of bounds). It's not a Keras naming quirk: everyone uses this odd 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 = tf.keras.layers.Conv2D(filters=32, kernel_size=7,
...                                     padding="same")
...
```

```
>>> fmaps = conv_layer(images)
>>> fmaps.shape
TensorShape([2, 70, 120, 32])
```

These two padding options are illustrated in Figure 14-7. For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.

If the stride is greater than 1 (in any direction), then the output size will not be equal to the input size, even if padding="same". For example, if you set strides=2 (or equivalently strides=(2, 2)), then the output feature maps will be 35 × 60: halved both vertically and horizontally. Figure 14-8 shows what happens when strides=2, with both padding options.

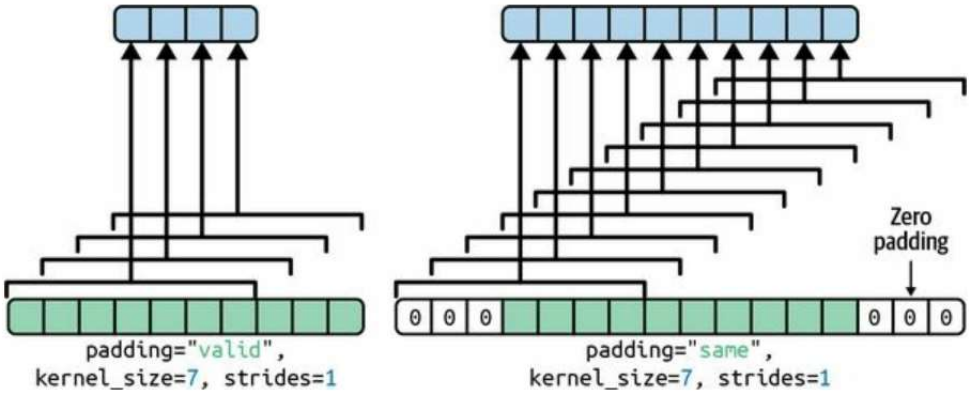

Figure 14-7. The two padding options, when strides=1

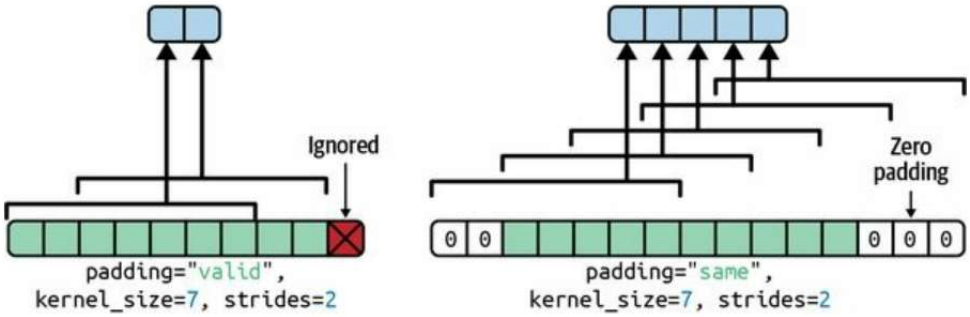

Figure 14-8. With strides greater than 1, the output is much smaller even when using "same" padding (and "valid" padding may ignore some inputs)

If you are curious, this is how the output size is computed:

- With padding="valid", if the width of the input is i_h, then the output width is equal to $(i_h - f_h + s_h) / s_h$, rounded down. Recall that f_h is the kernel width, and s_h is the horizontal stride. Any remainder in the division corresponds to ignored columns on the right side of the input image. The same logic can be used to compute the output height, and any ignored rows at the bottom of the image.
- With padding="same", the output width is equal to i_h / s_h, rounded up. To make this possible, the appropriate number of zero columns are padded to the left and right of the input image (an equal number if possible, or just one more on the right side). Assuming the output width is o_w, then the number of padded zero columns is $(o_w - 1) \times s_h + f_h - i_h$. Again, the same logic can be used to compute the output height and the number of padded rows.

Now let's look at the layer's weights (which were noted $w_{u, v, k', k}$ and b_k in Equation 14-1). Just like a Dense layer, a Conv2D layer holds all the layer's weights, including the kernels and biases. The kernels are initialized randomly, while the biases are initialized to zero. These weights are accessible as TF variables via the weights attribute, or as NumPy arrays via the get_weights() method:

```
>>> kernels, biases = conv_layer.get_weights()
>>> kernels.shape
(7, 7, 3, 32)
>>> biases.shape
(32,)
```

The kernels array is 4D, and its shape is [*kernel_height*, *kernel_width*, *input_channels*, *output_channels*]. The biases array 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).

Lastly, you will generally want to specify an activation function (such as ReLU) when creating a Conv2D layer, and also specify the corresponding kernel initializer (such as He initialization). This is for the same reason as for Dense 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.

As you can see, convolutional layers have quite a few hyperparameters: filters, kernel_size, padding, strides, activation, kernel_initializer, etc. 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.

Memory Requirements

Another challenge with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 200 5 × 5 filters, with stride 1 and "same" padding. If the input is a 150 × 100 RGB image (three channels), then the number of parameters is (5 × 5 × 3 + 1) × 200 = 15,200 (the + 1 corresponds to the bias terms), which is fairly small compared to a fully connected layer. [7] 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. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy 200 × 150 × 100 × 32 = 96 million bits (12 MB) of RAM. [8] And that's just for one instance—if a training batch contains 100 instances, then this 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 reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

> **TIP**
>
> If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, removing a few layers, using 16-bit floats instead of 32-bit floats, or distributing the CNN across multiple devices (you will see how to do this in Chapter 19).

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; all it does is aggregate the inputs using an aggregation function such as the max or mean. Figure 14-9 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2 × 2 *pooling kernel*, [9] 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 14-9, the input values are 1, 5, 3, 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).

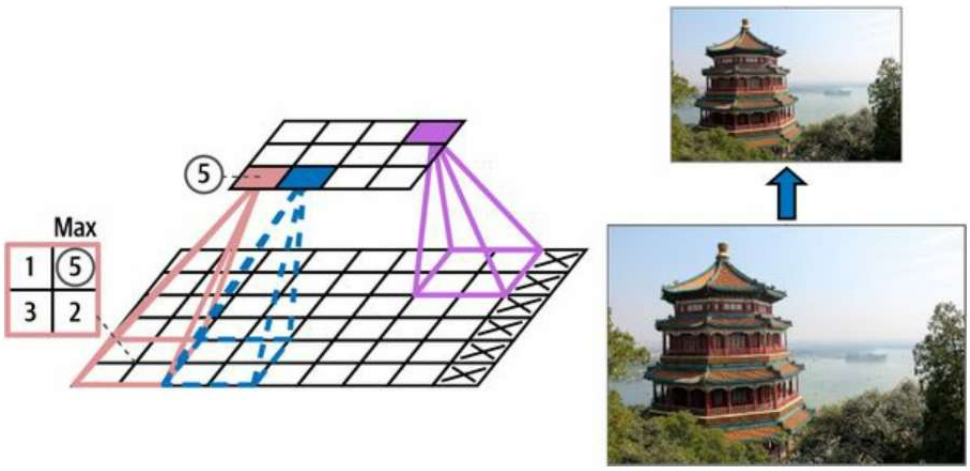

Figure 14-9. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

> **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 14-10. 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), simply 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.

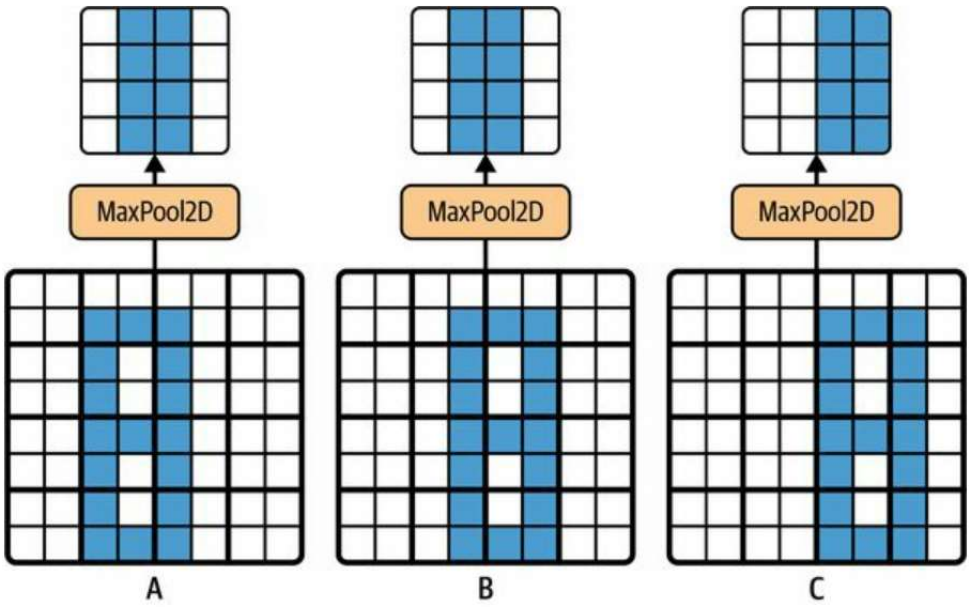

Figure 14-10. Invariance to small translations

Implementing Pooling Layers with Keras

The following code creates a MaxPooling2D layer, alias MaxPool2D, 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 "valid" padding (i.e., no padding at all):

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use AveragePooling2D, alias AvgPool2D, instead of 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 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 14-11), 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.

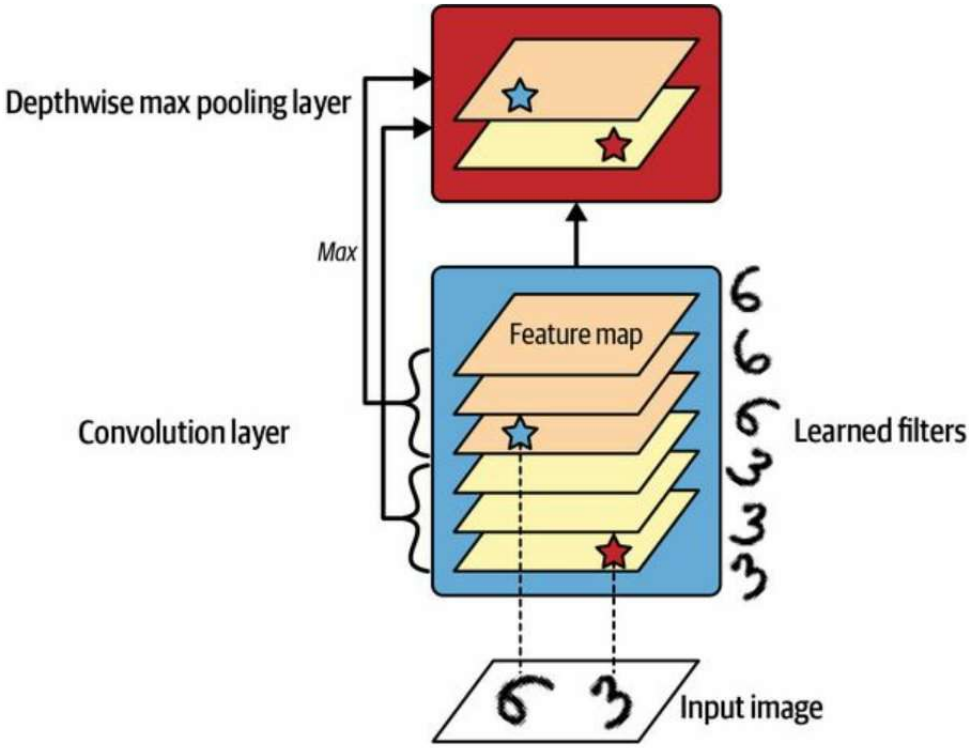

Figure 14-11. Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case)

Keras does not include a depthwise max pooling layer, but it's not too difficult to implement a custom layer for that:

```
class DepthPool(tf.keras.layers.Layer):
    def __init__(self, pool_size=2, **kwargs):
        super().__init__(**kwargs)
        self.pool_size = pool_size

    def call(self, inputs):
        shape = tf.shape(inputs)  # shape[-1] is the number of channels
        groups = shape[-1] // self.pool_size  # number of channel groups
        new_shape = tf.concat([shape[:-1], [groups, self.pool_size]], axis=0)
        return tf.reduce_max(tf.reshape(inputs, new_shape), axis=-1)
```

This layer reshapes its inputs to split the channels into groups of the desired size (pool_size), then it uses tf.reduce_max() to compute the max of each group. This implementation assumes that the stride is equal to the pool size,

which is generally what you want. Alternatively, you could use TensorFlow's tf.nn.max_pool() operation, and wrap in a Lambda layer to use it inside a Keras model, but sadly this op does not implement depthwise pooling for the GPU, only for the CPU.

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 (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just 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, simply use the GlobalAveragePooling2D class, alias GlobalAvgPool2D:

```
global_avg_pool = tf.keras.layers.GlobalAvgPool2D()
```

It's equivalent to the following Lambda layer, which computes the mean over the spatial dimensions (height and width):

```
global_avg_pool = tf.keras.layers.Lambda(
    lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

For example, if we apply this layer to the input images, we get the mean intensity of red, green, and blue for each image:

```
>>> global_avg_pool(images)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.64338624, 0.5971759 , 0.5824972 ],
       [0.76306933, 0.26011038, 0.10849128]], dtype=float32)>
```

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 14-12). 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).

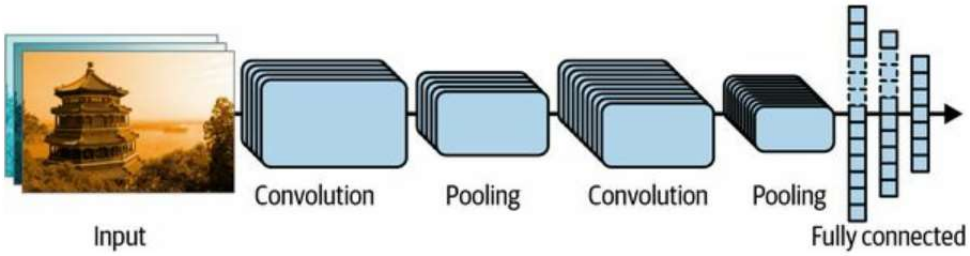

Figure 14-12. Typical CNN architecture

> **TIP**
>
> A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5 × 5 kernel, 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), usually with a stride of 2 or more. This will reduce 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 10):

```
from functools import partial
```

```
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                        activation="relu", kernel_initializer="he_normal")
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Let’s go through this code:

- We use the functools.partial() function (introduced in Chapter 11) to define DefaultConv2D, which acts just like Conv2D but with different default arguments: a small kernel size of 3, "same" padding, the ReLU activation function, and its corresponding He initializer.
- Next, we create the 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 sets input_shape=[28, 28, 1], because the images are 28 × 28 pixels, with a single color channel (i.e., grayscale). When you load the Fashion MNIST dataset, make sure each image has this shape: you may need to use np.reshape() or np.expanddims() to add the channels dimension. Alternatively, you could use a Reshape layer as the first layer in the model.
- We then add a max pooling layer that uses the default pool size of 2, so it divides each spatial dimension by a factor of 2.
- 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 and a dense output layer. Since it's a classification task with 10 classes, the output layer has 10 units, and it uses the softmax activation function. 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.

If you compile this model using the "sparse_categorical_crossentropy" loss and you fit the model to the Fashion MNIST training set, it should reach over 92% accuracy on the test set. It's not state of the art, but it is pretty good, and clearly much better than what we achieved with dense networks in Chapter 10.

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 top-five error rate for image classification —that is, the number of test images for which the system's top five predictions did *not* include the correct answer—fell from over 26% to less than 2.3% in just six years. 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). Along the way, we will also look at a few more architectures, including Xception, ResNeXt, DenseNet, MobileNet, CSPNet, and EfficientNet.

LeNet-5

The LeNet-5 architecture[10] 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 14-1.

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size
Out	Fully connected	–	10	–
F6	Fully connected	–	84	–
C5	Convolution	120	1 × 1	5 × 5
S4	Avg pooling	16	5 × 5	2 × 2
C3	Convolution	16	10 × 10	5 × 5
S2	Avg pooling	6	14 × 14	2 × 2
C1	Convolution	6	28 × 28	5 × 5
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. 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/colab3*. Yann LeCun's website also features great demos of LeNet-5 classifying digits.

AlexNet

The AlexNet CNN architecture[11] 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%! AlexaNet was 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 14-2 presents this architecture.

Table 14-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size
Out	Fully connected	–	1,000	–
F10	Fully connected	–	4,096	–
F9	Fully connected	–	4,096	–
S8	Max pooling	256	6 × 6	3 × 3
C7	Convolution	256	13 × 13	3 × 3
C6	Convolution	384	13 × 13	3 × 3
C5	Convolution	384	13 × 13	3 × 3
S4	Max pooling	256	13 × 13	3 × 3
C3	Convolution	256	27 × 27	5 × 5
S2	Max pooling	96	27 × 27	3 × 3
C1	Convolution	96	55 × 55	11 × 11
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 14-13). To do this, you can use Keras's data augmentation layers, introduced in Chapter 13 (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. To produce a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase your training set size.

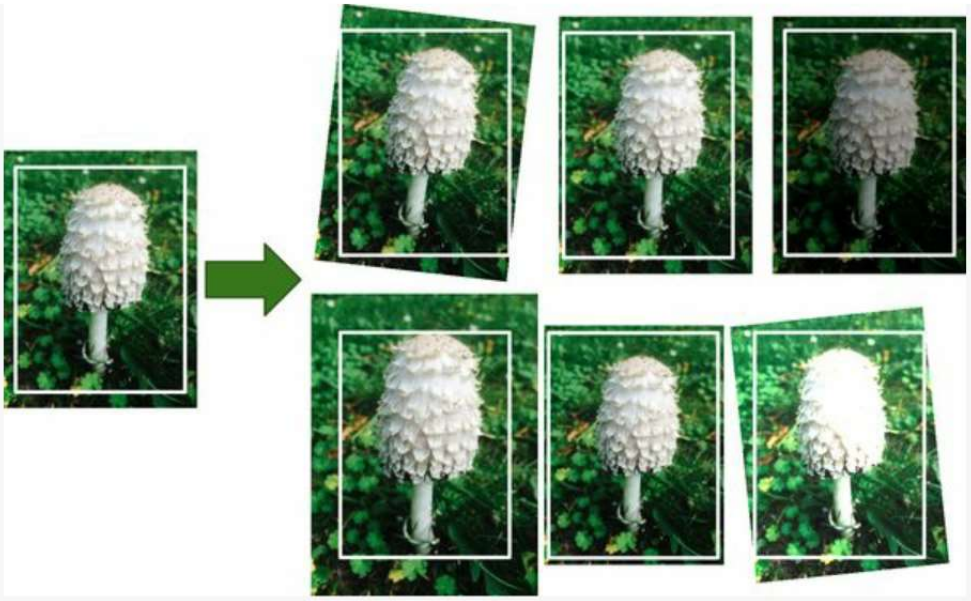

Figure 14-13. Generating new training instances from existing ones

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.

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, 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. Equation 14-2 shows how to apply LRN.

Equation 14-2. Local response normalization (LRN)

b i = a i k+α∑ j=j low j high a j 2 -β with j high = min i + r 2 , f n - 1 j low = max 0 , i - r 2

In this equation:

- b_i is the normalized output of the neuron located in feature map i, at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k, α, β, and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

For example, if $r = 2$ and a neuron has a strong activation, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as: $r = 5$, $\alpha = 0.0001$, $\beta = 0.75$, and $k = 2$. You can implement this step by using the tf.nn.local_response_normalization() function (which you can wrap in a Lambda layer if you want to use it in a Keras model).

A variant of AlexNet called *ZF Net*[12] 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, [13] 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 14-15). This was made possible by subnetworks called *inception modules*, [14] 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 14-14 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., to stack the feature maps from all four top convolutional layers). It can be implemented using Keras's Concatenate layer, using the default axis=-1.

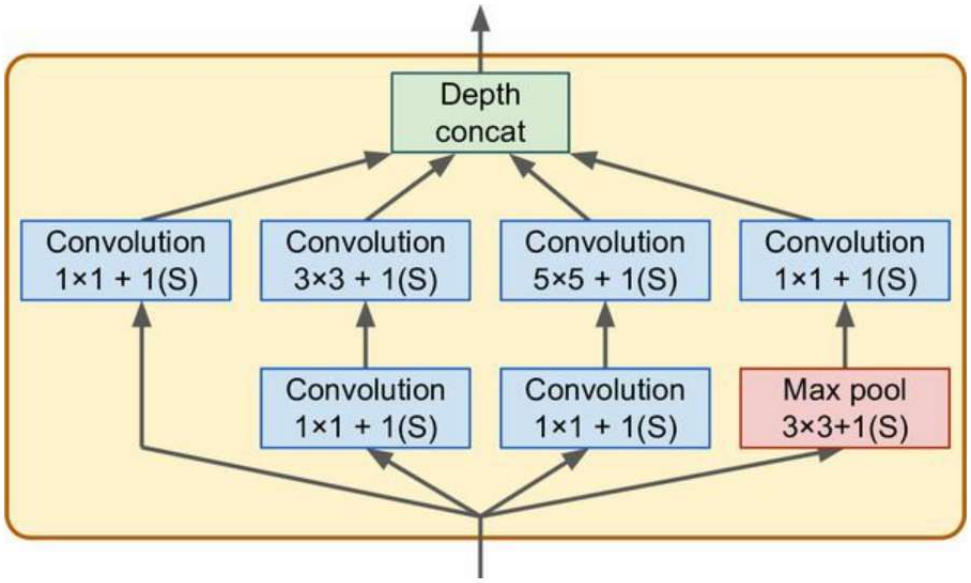

Figure 14-14. Inception module

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 14-15). 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 14-14). Note that all the convolutional layers use the ReLU activation function.

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.

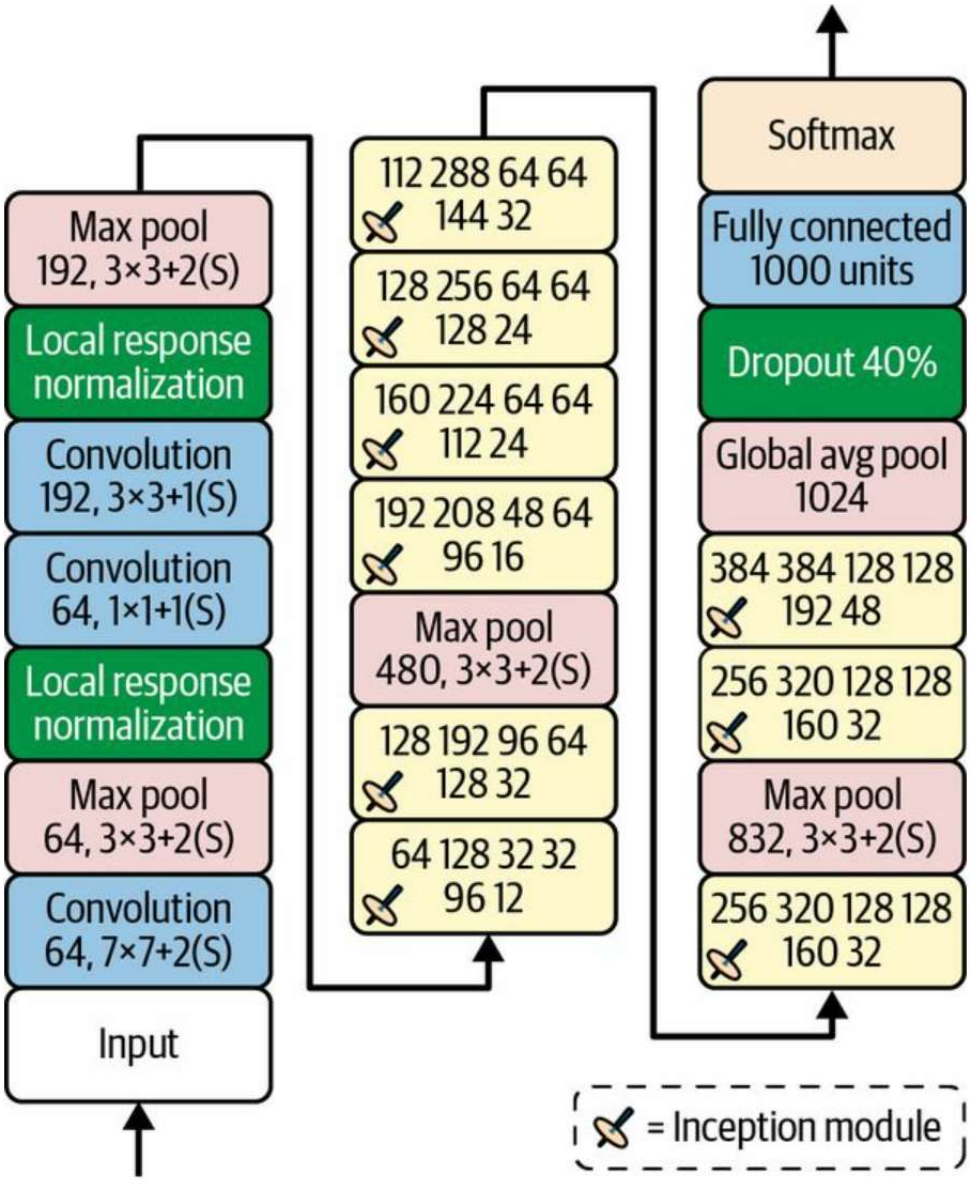

Figure 14-15. GoogLeNet architecture

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.

VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet, [15] 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.

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a Residual Network (ResNet)[16] 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 14-16).

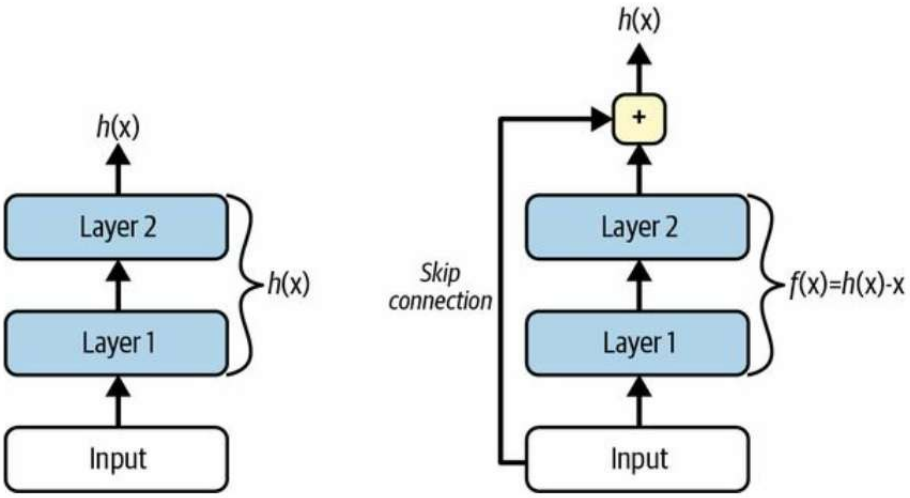

Figure 14-16. Residual learning

When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. 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 14-17). 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.

Now let's look at ResNet's architecture (see Figure 14-18). 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).

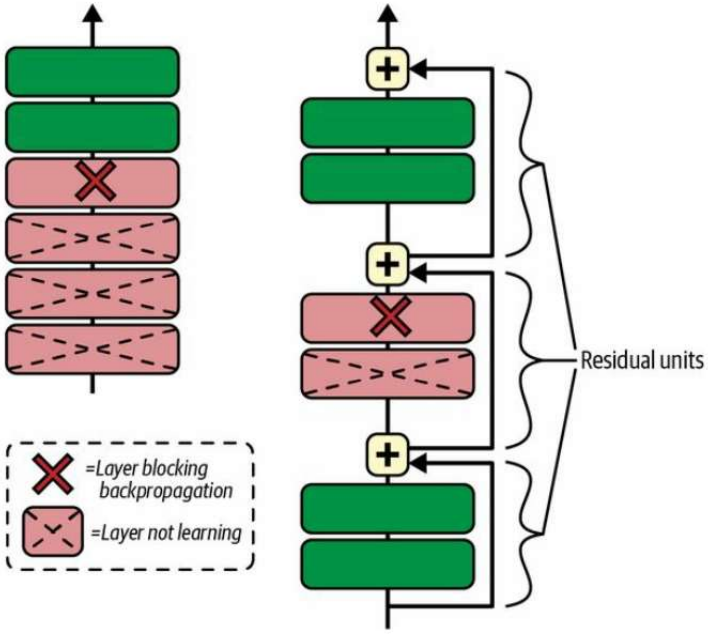

Figure 14-17. Regular deep neural network (left) and deep residual network (right)

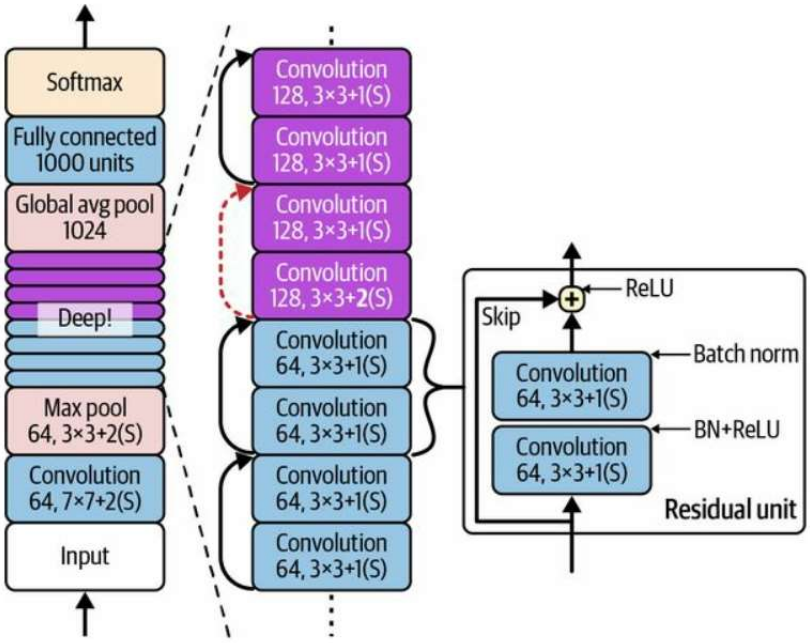

Figure 14-18. ResNet architecture

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 14-18). 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 14-19).

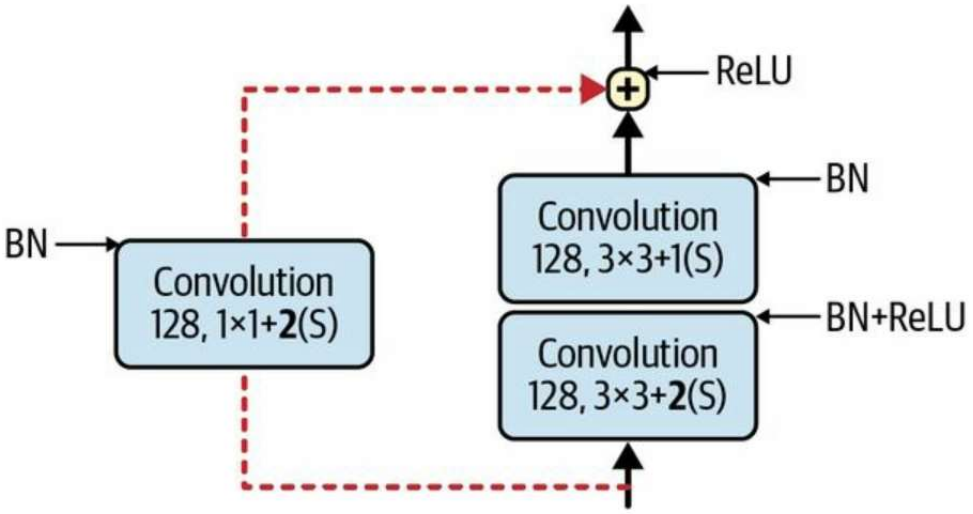

Figure 14-19. Skip connection when changing feature map size and depth

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) [17] 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.

> **NOTE**
>
> Google's Inception-v4[18] architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

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 × 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.

Xception

Another variant of the GoogLeNet architecture is worth noting: Xception[19] (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of 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 [20]). 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 14-20). 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.

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 14-20 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.

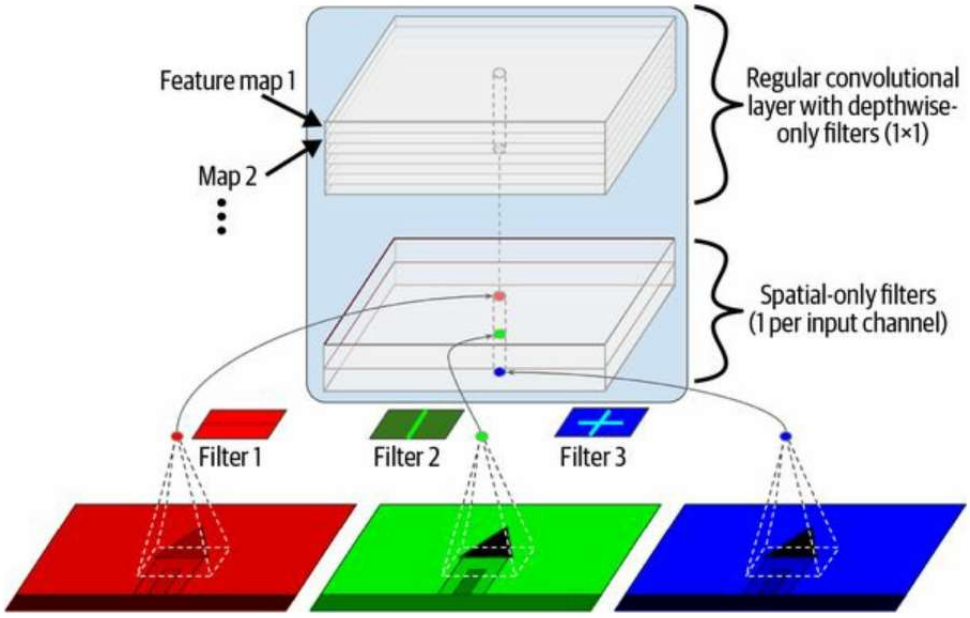

Figure 14-20. Depthwise separable convolutional layer

> **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). In Keras, just use SeparableConv2D instead of Conv2D: it's a drop-in replacement. Keras also offers a DepthwiseConv2D layer that implements the first part of a depthwise separable convolutional layer (i.e., applying one spatial filter per input feature map).

SENet

The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network (SENet). [21] 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 14-21.

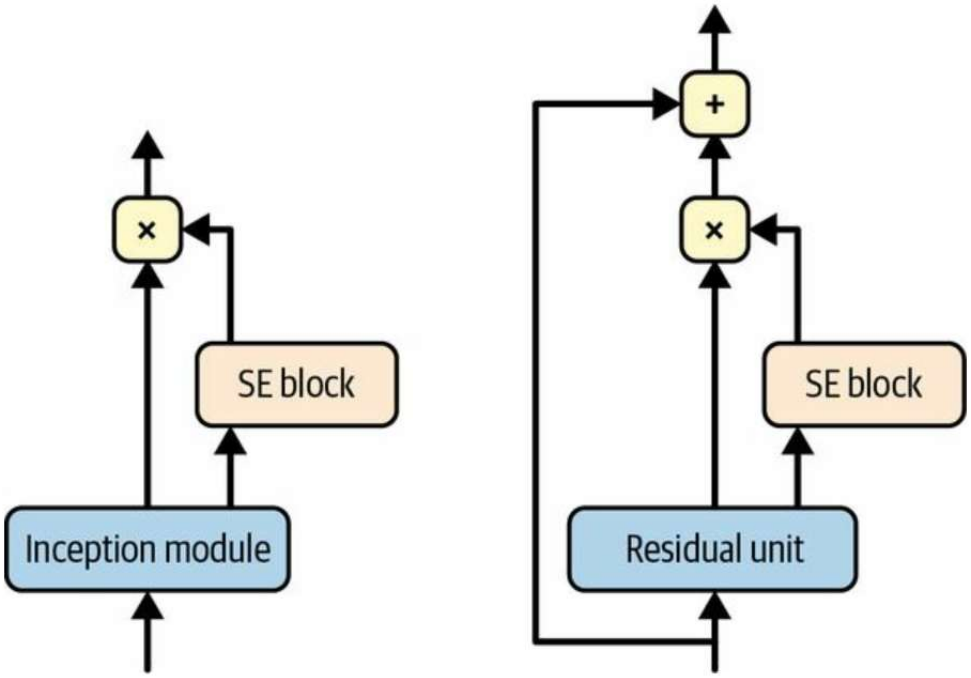

Figure 14-21. SE-Inception module (left) and SE-ResNet unit (right)

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 14-22. 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 irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

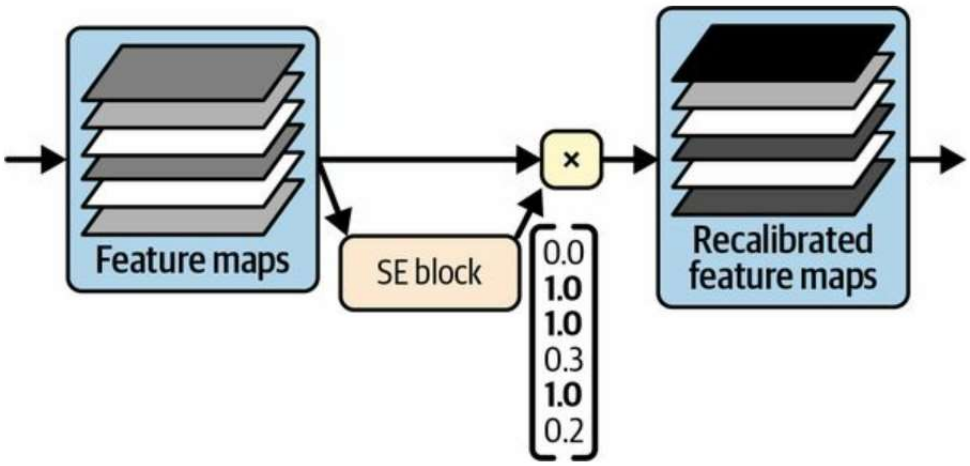

Figure 14-22. An SE block performs feature map recalibration

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 14-23).

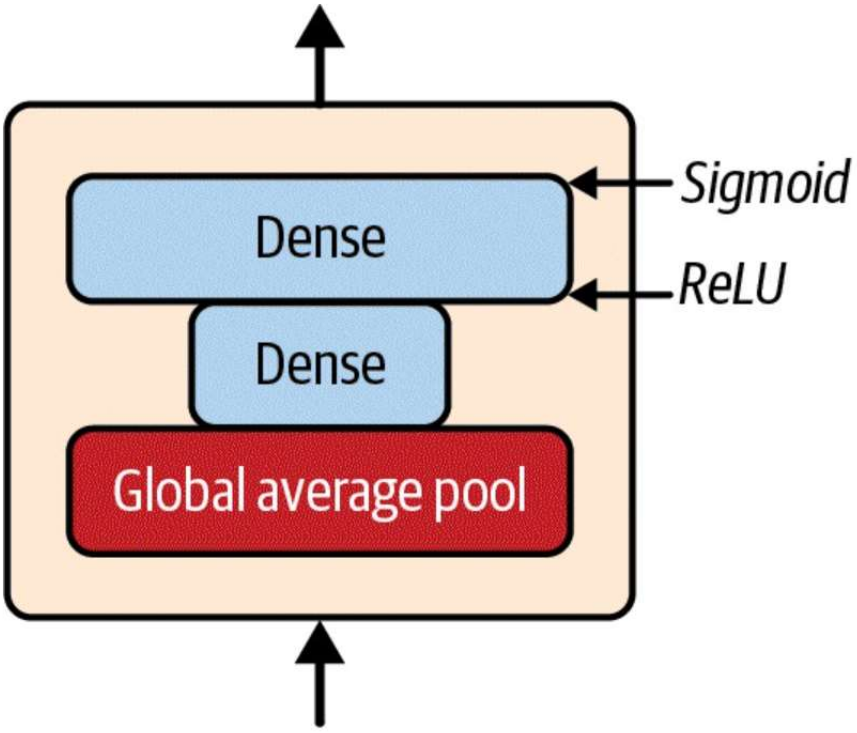

Figure 14-23. SE block architecture

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 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 17). 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:

ResNeXt[22]

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[23]

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 4 in a block takes as input the depthwise concatenation of the outputs of layers 1, 2, and 3 in that block. Dense blocks are separated by a few transition layers.

MobileNet[24]

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.

CSPNet[25]

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[26]

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.

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.

Choosing the Right CNN Architecture

With so many CNN architectures, how do you choose which one is best for your project? Well, it depends on what matters most to you: Accuracy? Model size (e.g., for deployment to a mobile device)? Inference speed on CPU? On GPU? Table 14-3 lists the best pretrained models currently available in Keras (you'll see how to use them later in this chapter), sorted by model size. You can find the full list at *https://keras.io/api/applications*. For each model, the table shows the Keras class name to use (in the tf.keras.applications package), the model's size in MB, the top-1 and top-5 validation accuracy on the ImageNet dataset, the number of parameters (millions), and the inference time on CPU and GPU in ms, using batches of 32 images on reasonably powerful hardware. [27] For each column, the best value is highlighted. As you can see, larger models are generally more accurate, but not always; for example, EfficientNetB2 outperforms InceptionV3 both in size and accuracy. I only kept InceptionV3 in the list because it is almost twice as fast as EfficientNetB2 on a CPU. Similarly, InceptionResNetV2 is fast on a CPU, and ResNet50V2 and ResNet101V2 are blazingly fast on a GPU.

Table 14-3. Pretrained models available in Keras

Class name	Size (MB)	Top-1 acc	Top-5 acc	Params
MobileNetV2	**14**	71.3%	90.1%	**3.5M**
MobileNet	16	70.4%	89.5%	4.3M
NASNetMobile	23	74.4%	91.9%	5.3M
EfficientNetB0	29	77.1%	93.3%	5.3M
EfficientNetB1	31	79.1%	94.4%	7.9M
EfficientNetB2	36	80.1%	94.9%	9.2M
EfficientNetB3	48	81.6%	95.7%	12.3M

EfficientNetB4	75	82.9%	96.4%	19.5M
InceptionV3	92	77.9%	93.7%	23.9M
ResNet50V2	98	76.0%	93.0%	25.6M
EfficientNetB5	118	83.6%	96.7%	30.6M
EfficientNetB6	166	84.0%	96.8%	43.3M
ResNet101V2	171	77.2%	93.8%	44.7M
InceptionResNetV2	215	80.3%	95.3%	55.9M
EfficientNetB7	256	**84.3%**	**97.0%**	66.7M

I hope you enjoyed this deep dive into the main CNN architectures! Now let’s see how to implement one of them using Keras.

Implementing a ResNet-34 CNN Using Keras

Most CNN architectures described so far can be implemented pretty naturally using Keras (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 Keras. First, we'll create a ResidualUnit layer:

```
DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="same", kernel_initializer="he_normal",
                        use_bias=False)

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

As you can see, this code matches Figure 14-19 pretty closely. In the constructor, we create all the layers we will need: the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left

(only needed if the stride is greater than 1). Then in the call() method, we make the inputs go through the main layers and the skip layers (if any), and we add both outputs and apply the activation function.

Now we can build a ResNet-34 using a Sequential model, since it's really just a long sequence of layers—we can treat each residual unit as a single layer now that we have the ResidualUnit class. The code closely matches Figure 14-18:

```
model = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224, 3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"),
])
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

The only tricky part in this code is the loop that adds the ResidualUnit layers to the model: 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 add the ResidualUnit, and finally we update prev_filters.

It is amazing that in about 40 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 the Keras API. Implementing the other CNN architectures is a bit longer, but not much harder. However, Keras comes with several of these architectures built in, so why not use them instead?

Using Pretrained Models from Keras

In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are readily available with a single line of code in the tf.keras.applications package.

For example, you can load the ResNet-50 model, pretrained on ImageNet, with the following line of code:

```
model = tf.keras.applications.ResNet50(weights="imagenet")
```

That's all! This will create a ResNet-50 model and download weights pretrained on the ImageNet dataset. To use it, you first need to ensure that the images have the right size. A ResNet-50 model expects 224 × 224–pixel images (other models may expect other sizes, such as 299 × 299), so let's use Keras's Resizing layer (introduced in Chapter 13) to resize two sample images (after cropping them to the target aspect ratio):

```
images = load_sample_images()["images"]
images_resized = tf.keras.layers.Resizing(height=224, width=224,
                                          crop_to_aspect_ratio=True)(images)
```

The pretrained models assume that the images are preprocessed in a specific way. In some cases they may expect the inputs to be scaled from 0 to 1, or from –1 to 1, and so on. Each model provides a preprocess_input() function that you can use to preprocess your images. These functions assume that the original pixel values range from 0 to 255, which is the case here:

```
inputs = tf.keras.applications.resnet50.preprocess_input(images_resized)
```

Now we can use the pretrained model to make predictions:

```
>>> Y_proba = model.predict(inputs)
>>> Y_proba.shape
(2, 1000)
```

As usual, the output Y_proba is a matrix with one row per image and one column per class (in this case, there are 1,000 classes). If you want to display the top *K* predictions, including the class name and the estimated probability of each predicted class, use the decode_predictions() function. For each image, it returns an array containing the top *K* predictions, where each prediction is represented as an array containing the class identifier, [28] its name, and the corresponding confidence score:

```
top_K = tf.keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print(f"Image #{image_index}")
    for class_id, name, y_proba in top_K[image_index]:
        print(f"  {class_id} - {name:12s} {y_proba:.2%}")
```

The output looks like this:

```
Image #0
  n03877845 - palace       54.69%
  n03781244 - monastery    24.72%
  n02825657 - bell_cote    18.55%
Image #1
  n04522168 - vase         32.66%
  n11939491 - daisy        17.81%
  n03530642 - honeycomb    12.06%
```

The correct classes are palace and dahlia, so the model is correct for the first image but wrong for the second. However, that's because dahlia is not one of the 1,000 ImageNet classes. With that in mind, vase is a reasonable guess (perhaps the flower is in a vase?), and daisy is not a bad choice either, since dahlias and daisies are both from the same Compositae family.

As you can see, it is very easy to create a pretty good image classifier using a pretrained model. As you saw in Table 14-3, many other vision models are available in tf.keras.applications, from lightweight and fast models to large and accurate ones.

But what if you want to use an image classifier for classes of images that are not part of ImageNet? 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. For example, let's train a model to classify pictures of flowers, reusing a pretrained Xception model. First, we'll load the flowers dataset using TensorFlow Datasets (introduced in Chapter 13):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples  # 3670
class_names = info.features["label"].names  # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes  # 5
```

Note that you can get information about the dataset by setting with_info=True. Here, we get the dataset size and the names of the classes. Unfortunately, there is only a "train" dataset, no test set or validation set, so we need to split the training set. Let's call tfds.load() again, but this time taking the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

```
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

All three datasets contain individual images. We need to batch them, but first we need to ensure they all have the same size, or batching will fail. We can use a Resizing layer for this. We must also call the tf.keras.applications.xcep tion.preprocess_input() function to preprocess the images appropriately for the Xception model. Lastly, we'll also shuffle the training set and use prefetching:

```
batch_size = 32
```

```
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224, crop_to_aspect_ratio=True),
    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)
])
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
```

Now each batch contains 32 images, all of them 224 × 224 pixels, with pixel values ranging from –1 to 1. Perfect!

Since the dataset is not very large, a bit of data augmentation will certainly help. Let's create a data augmentation model that we will embed in our final model. During training, it will randomly flip the images horizontally, rotate them a little bit, and tweak the contrast:

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42),
    tf.keras.layers.RandomContrast(factor=0.2, seed=42)
])
```

> **TIP**
>
> The tf.keras.preprocessing.image.ImageDataGenerator class makes it easy to load images from disk and augment them in various ways: you can shift each image, rotate it, rescale it, flip it horizontally or vertically, shear it, or apply any transformation function you want to it. This is very convenient for simple projects. However, a tf.data pipeline is not much more complicated, and it's generally faster. Moreover, if you have a GPU and you include the preprocessing or data augmentation layers inside your model, they will benefit from GPU acceleration during training.

Next let's load an Xception model, pretrained on ImageNet. We exclude the top of the network by setting include_top=False. This excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer (feeding it the output of the base model), followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we wrap all this in a Keras Model:

```
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

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:

```
for layer in base_model.layers:
    layer.trainable = False
```

> **WARNING**
>
> Since our model uses the base model's layers directly, rather than the base_model object itself, setting base_model.trainable=False would have no effect.

Finally, we can compile the model and start training:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=3)
```

> **WARNING**
>
> If you are running in Colab, make sure the runtime is using a GPU: select Runtime → "Change runtime type", choose "GPU" in the "Hardware accelerator" drop-down menu, then click Save. It's possible to train the model without a GPU, but it will be terribly slow (minutes per epoch, as opposed to seconds).

After training the model for a few epochs, its validation accuracy should reach a bit over 80% and then stop improving. This means that the top layers are now pretty well trained, and we are ready to unfreeze some of the base model's top layers, then continue training. For example, let's unfreeze layers 56 and above (that's the start of residual unit 7 out of 14, as you can see if

you list the layer names):

```
for layer in base_model.layers[56:]:
    layer.trainable = True
```

Don’t forget to compile the model whenever you freeze or unfreeze layers. Also make sure to use a much lower learning rate to avoid damaging the pretrained weights:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

This model should reach around 92% accuracy on the test set, in just a few minutes of training (with a GPU). If you tune the hyperparameters, lower the learning rate, and train for quite a bit longer, you should be able to reach 95% to 97%. 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 10: to predict a bounding box around the object, a common approach is to predict the horizontal and vertical coordinates of the object's center, as well as its height and width. This means we have four numbers to predict. It does not require much change to the model; we just need to add a second dense output layer with four units (typically on top of the global average pooling layer), and it can be trained using the MSE loss:

```
base_model = tf.keras.applications.xception.Xception(weights="imagenet",
                                                     include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = tf.keras.layers.Dense(4)(avg)
model = tf.keras.Model(inputs=base_model.input,
                       outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2],  # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy"])
```

But now we have a problem: the flowers dataset does not have bounding boxes around the flowers. So, we need to add them ourselves. This is often one of the hardest and most costly parts of a machine learning project: getting the labels. 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 image labeling tool like VGG Image Annotator, LabelImg, OpenLabeler, or ImgLab, or perhaps a commercial tool like LabelBox or Supervisely. 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. Adriana Kovashka et al. wrote a very practical paper[29] about crowdsourcing in computer vision. I recommend you check it out, even if you do not plan to use crowdsourcing. If there are just a few hundred or a even a

couple 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.

Now let's suppose you've obtained the bounding boxes for every image in the flowers dataset (for now we will assume there is a single bounding box per image). You then need to create a dataset whose items will be batches of preprocessed images along with their class labels and their bounding boxes. Each item should be a tuple of the form (images, (class_labels, bounding_boxes)). Then you are ready to train your model!

> **TIP**
>
> The bounding boxes should be normalized so that the horizontal and vertical coordinates, as well as the height and width, all range from 0 to 1. Also, it is common to predict the square root of the height and width rather than the height and width directly: this way, a 10-pixel error for a large bounding box will not be penalized as much as a 10-pixel error for a small bounding box.

The MSE often works fairly well as a cost function 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): the area of overlap between the predicted bounding box and the target bounding box, divided by the area of their union (see Figure 14-24). In Keras, it is implemented by the tf.keras.metrics.MeanIoU class.

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)?

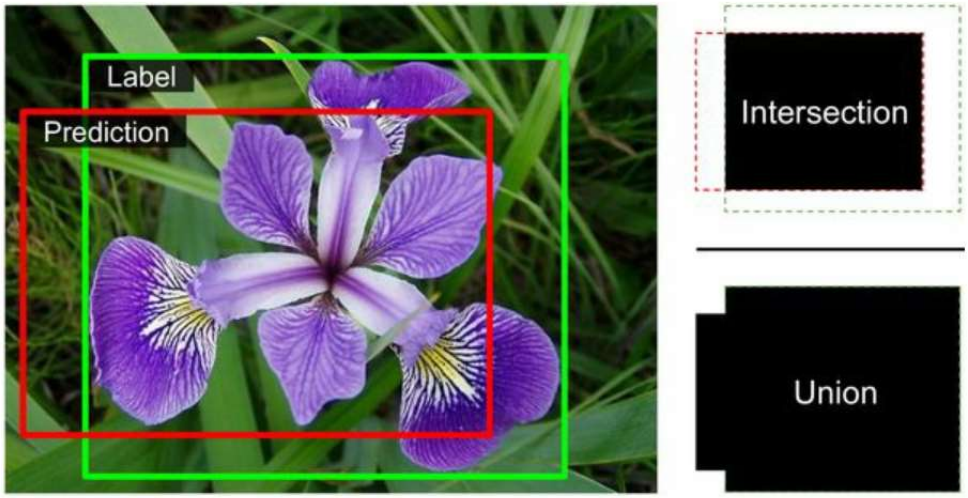

Figure 14-24. IoU metric for bounding boxes

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 14-25. 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.

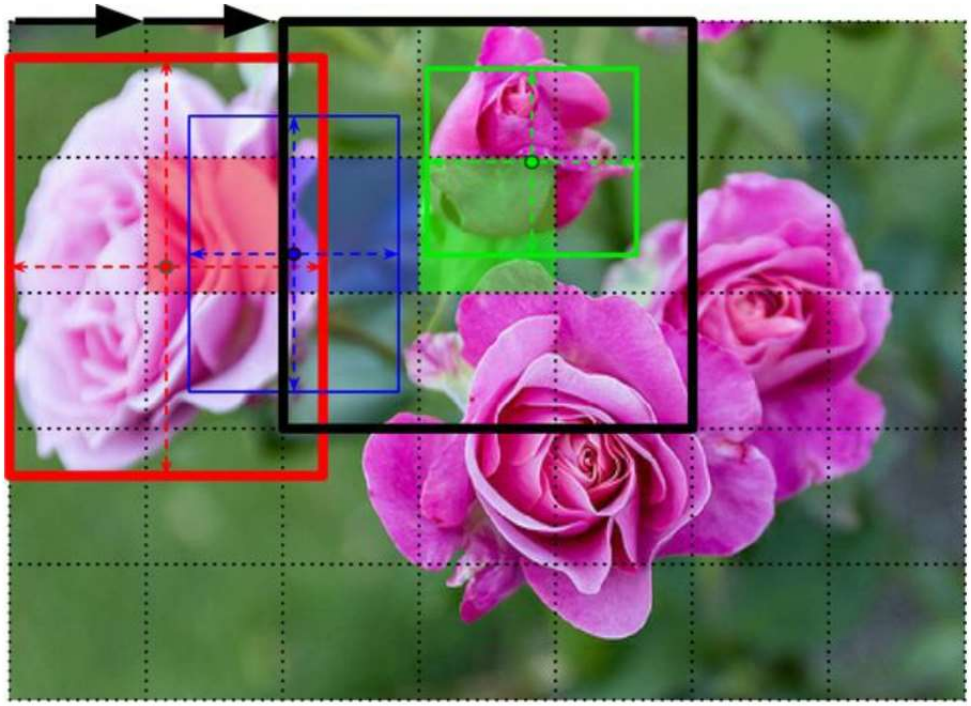

Figure 14-25. Detecting multiple objects by sliding a CNN across the image

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 larger 4 × 4 regions as well, to get even more bounding boxes.

This technique is fairly straightforward, but as you can see it will often detect the same object multiple times, at slightly different positions. Some postprocessing is needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression*. 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 14-25, 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

The idea of FCNs was first introduced in a 2015 paper[30] 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*, 1, 1, 200].

> **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 you 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 [31] (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. It was trained on 224 × 224 images, and it outputs 10 numbers:

- Outputs 0 to 4 are sent through the softmax activation function, and this gives the class probabilities (one per class).
- Output 5 is sent through the sigmoid activation function, and this gives the objectness score.
- Outputs 6 and 7 represent the bounding box's center coordinates; they also go through a sigmoid activation function to ensure they range from 0 to 1.
- Lastly, outputs 8 and 9 represent the bounding box's height and width; they do not go through any activation function to allow the bounding boxes to extend beyond the borders of the image.

We can now convert the CNN's dense layers to convolutional layers. In fact, we don't even need to retrain it; 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 14-26). If we feed the FCN a 448 × 448 image (see the right side of Figure 14-26), the bottleneck layer will now output 14 × 14 feature maps. [32] Since the dense output layer was replaced by a convolutional layer using 10 filters of size 7 × 7, with "valid" padding and stride 1, the output will be composed of 10 features 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 10 numbers (5 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.

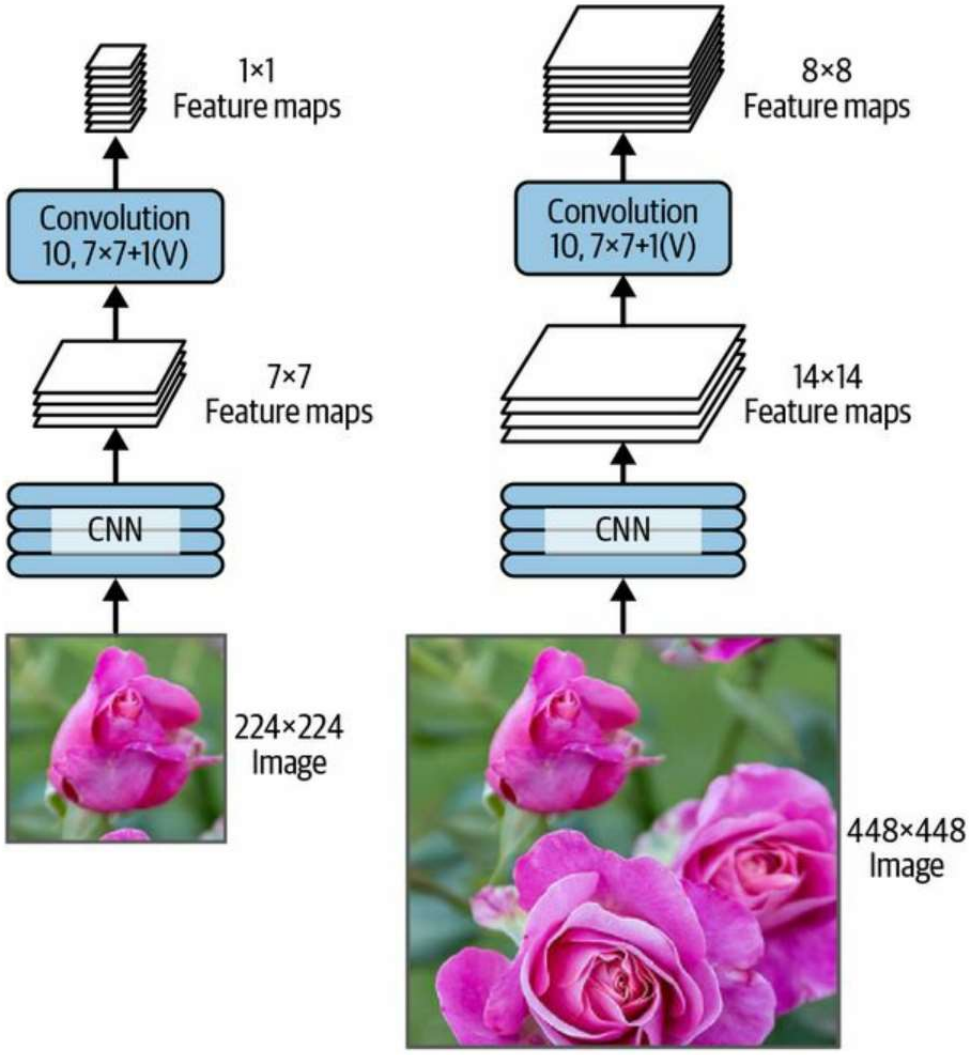

Figure 14-26. The same fully convolutional network processing a small image (left) and a large one (right)

You Only Look Once

YOLO is a fast and accurate object detection architecture proposed by Joseph Redmon et al. in a 2015 paper.[33] 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 postprocessing, 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 TensorFlow, Keras, PyTorch, and more. It was continuously improved over the years, with YOLOv2, YOLOv3, and YOLO9000 (again by Joseph Redmon et al.), YOLOv4 (by Alexey Bochkovskiy et al.), YOLOv5 (by Glenn Jocher), and PP-YOLO (by Xiang Long et al.).

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 YOLOv4-tiny, which is 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: 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 left 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). That's it!

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.

Many object detection models are available on TensorFlow Hub, often with pretrained weights, such as YOLOv5, [34] SSD, [35] Faster R-CNN, [36] and EfficentDet. [37]

SSD and EfficientDet are "look once" detection models, similar to YOLO.

EfficientDet is based on the EfficientNet convolutional architecture. Faster R-CNN is more complex: 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. The best place to start using these models is TensorFlow Hub's excellent object detection tutorial.

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 to or further away from the camera, 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. [38] 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.

> **TIP**
>
> There are a few DeepSORT implementations available on GitHub, including a TensorFlow implementation of YOLOv4 + DeepSORT: *https://github.com/theAIGuysCode/yolov4-deepsort*.

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 14-27. Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the right 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 will not be much more precise than that.

Figure 14-27. 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. 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 add up all the strides greater than 1), 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 *transposed convolutional layer*: [39] this is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 14-28). 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 14-28). 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 Keras, you can use the Conv2DTranspose 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).

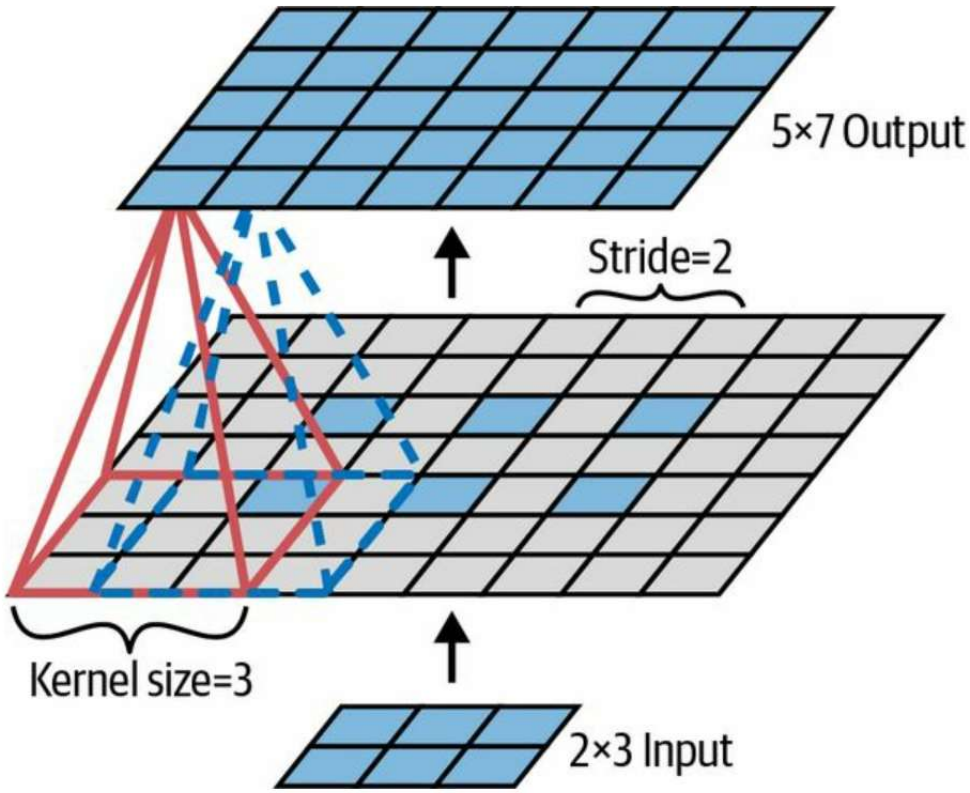

Figure 14-28. Upsampling using a transposed convolutional layer

OTHER KERAS CONVOLUTIONAL LAYERS

Keras also offers a few other kinds of convolutional layers:

tf.keras.layers.Conv1D

A convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as you will see in Chapter 15.

tf.keras.layers.Conv3D

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

dilation_rate

Setting the dilation_rate 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 price 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 14-29). 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*.

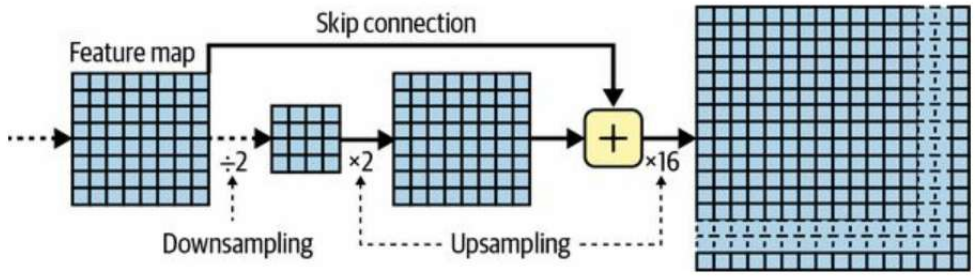

Figure 14-29. Skip layers recover some spatial resolution from lower layers

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[40] 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, but you also get a pixel mask that locates pixels in the bounding box that belong to the object. This model is available on TensorFlow Hub, pretrained on the COCO 2017 dataset. The field is moving fast, though so if you want to try the latest and greatest models, please check out the state-of-the-art section of *https://paperswithcode.com*.

As you can see, the field of deep computer vision is vast and fast-paced, with all sorts of architectures popping up every year. Almost all of them are 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 16). The progress made over the last decade 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 17), *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. When would you want to add a local response normalization layer?
6. Can you name the main innovations in AlexNet, as compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, Xception, and EfficientNet?
7. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?
8. What is the main technical difficulty of semantic segmentation?
9. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
10. 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.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).

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.

11. Go through TensorFlow's Style Transfer tutorial. This is a fun way to generate art using deep learning.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

---

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 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!

8 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 ≈ 11.44 MiB.

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

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

11 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.

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

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

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

15 Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", arXiv preprint arXiv:1409.1556 (2014).

16 Kaiming He et al., "Deep Residual Learning for Image Recognition", arXiv preprint arXiv:1512:03385 (2015).

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

18 Christian Szegedy et al., "Inception–v4, Inception-ResNet and the Impact of Residual Connections on Learning", arXiv preprint arXiv:1602.07261 (2016).

19 François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions", arXiv preprint arXiv:1610.02357 (2016).

20 This name can sometimes be ambiguous, since spatially separable convolutions are often called "separable convolutions" as well.

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

22 Saining Xie et al., "Aggregated Residual Transformations for Deep Neural Networks", arXiv preprint arXiv:1611.05431 (2016).

23 Gao Huang et al., "Densely Connected Convolutional Networks", arXiv preprint arXiv:1608.06993 (2016).

24 Andrew G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv preprint arxiv:1704.04861 (2017).

25 Chien-Yao Wang et al., "CSPNet: A New Backbone That Can Enhance Learning Capability of CNN", arXiv preprint arXiv:1911.11929 (2019).

26 Mingxing Tan and Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", arXiv preprint arXiv:1905.11946 (2019).

27 A 92-core AMD EPYC CPU with IBPB, 1.7 TB of RAM, and an Nvidia Tesla A100 GPU.

28 In the ImageNet dataset, each image is mapped to a word in the WordNet dataset: the class ID is just a WordNet ID.

29 Adriana Kovashka et al., “Crowdsourcing in Computer Vision”, *Foundations and Trends in Computer Graphics and Vision* 10, no. 3 (2014): 177–243.

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

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

32 This assumes we used only "same" padding in the network: "valid" padding would reduce the size of the feature maps. Moreover, 448 can be neatly divided by 2 several times until we reach 7, without any rounding error. If any layer uses a different stride than 1 or 2, then there may be some rounding error, so again the feature maps may end up being smaller.

33 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.

34 You can find YOLOv3, YOLOv4, and their tiny variants in the TensorFlow Models project at *https://homl.info/yolotf*.

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

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

37 Mingxing Tan et al., “EfficientDet: Scalable and Efficient Object Detection”, arXiv preprint arXiv:1911.09070 (2019).

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

39 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.

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