



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. So 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:²⁷

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        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 `MCAlpha` `Dropout` 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 that it replaces the `Dropout` layers with `MCDropout`, then copy the existing model's weights to your new model.

In short, MC Dropout is a fantastic 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 regularization technique that is popular 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 ($\mathbf{w} \leftarrow \mathbf{w} r/\|\mathbf{w}\|_2$).

²⁷ 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`.

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:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=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_constraint`. 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	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping ($+\ell_2$ reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	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	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	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, fold the Batch Normalization layers into the previous layers, and possibly use a faster activation function such as leaky ReLU or just ReLU. 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” on page 685](#)). 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 Keras. 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. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it OK to initialize the bias terms to 0?
3. Name three advantages of the SELU activation function over ReLU.
4. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
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 ELU activation function.
 - b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with `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 in [Appendix A](#).

Custom Models and Training with TensorFlow

Up until now, we've used only TensorFlow's high-level API, `tf.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 `tf.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.



TensorFlow 2.0 (beta) was released in June 2019, making TensorFlow much easier to use. The first edition of this book used TF 1, while this edition uses TF 2.

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 could 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 popular Deep Learning library (in terms of citations in papers, adoption in companies, stars on GitHub, etc.). 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, then optimizing it (e.g., by pruning unused nodes), and finally 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 autodiff (see [Chapter 10](#) and [Appendix D](#)) 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 `tf.keras`,¹ 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).

¹ TensorFlow includes another Deep Learning API called the *Estimators API*, but the TensorFlow team recommends using `tf.keras` instead.



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.

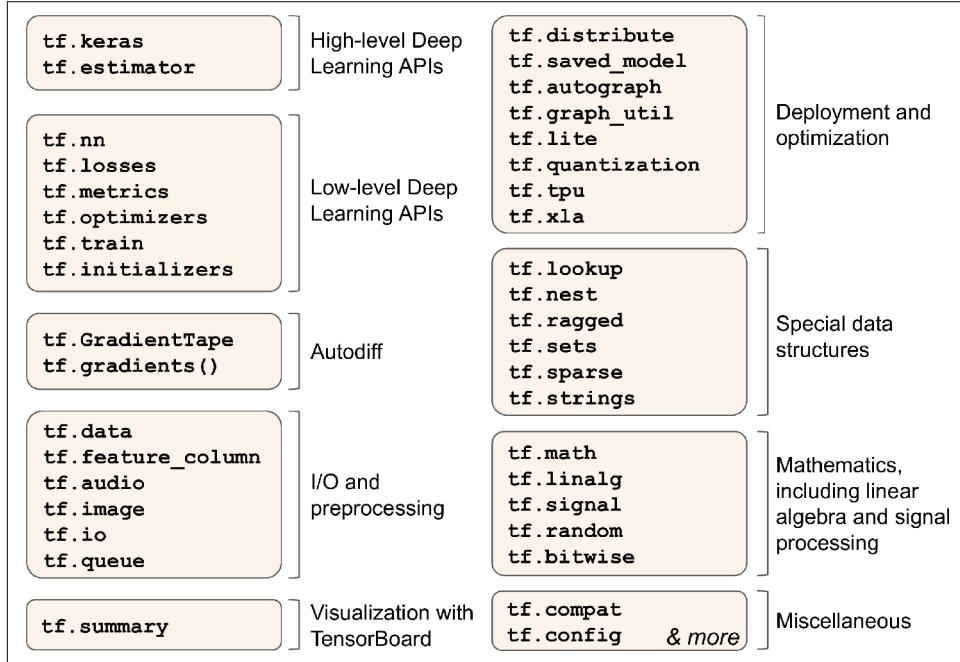


Figure 12-1. TensorFlow's Python API

At the lowest level, each TensorFlow operation (*op* for short) is implemented using highly efficient C++ code.² 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 operations³ (we will discuss how to use TensorFlow with GPUs or TPUs in Chapter 19).

TensorFlow's architecture is shown in Figure 12-2. Most of the time your code will use the high-level APIs (especially `tf.keras` and `tf.data`); but when you need more flexibility, you will use the lower-level Python API, handling tensors directly. Note that

² If you ever need to (but you probably won't), you can write your own operations using the C++ API.

³ To learn more about TPUs and how they work, check out <https://homl.info/tpus>.

APIs for other languages are also available. 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.

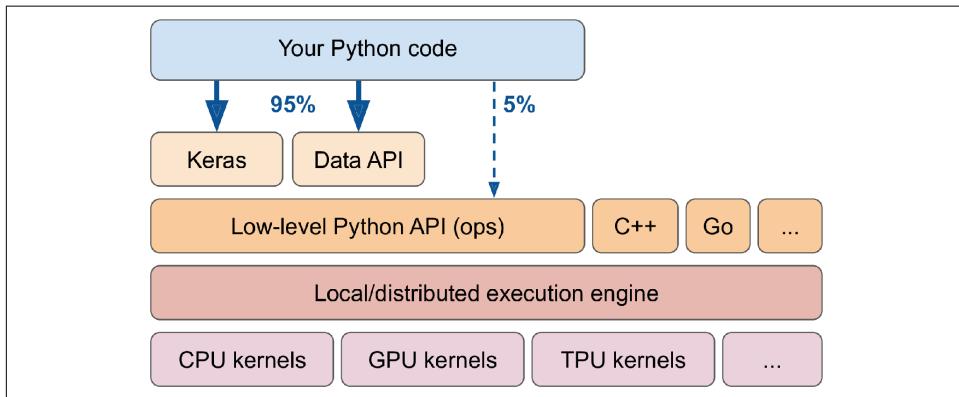


Figure 12-2. TensorFlow's architecture

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](#)). If you do not want to use the Python API, there are C++, Java, Go, and Swift APIs. There is even a JavaScript implementation called *TensorFlow.js* that makes it possible to run your models directly in your browser.

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.



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 <http://stackoverflow.com/> and tag your question with *tensorflow* and *python*. You can file bugs and feature requests through [GitHub](#). For general discussions, join the [Google group](#).

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 *TensorFlow*. 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:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # scalar
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

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

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indexing works much like in NumPy:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Most importantly, all sorts of tensor operations are available:

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
```

```

array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,   4.,   9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, 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.

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 often 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).



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 operations⁴ while preserving well-organized packages.

⁴ 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).

Keras' Low-Level API

The Keras API has its own low-level API, located in `keras.backend`. It includes functions like `square()`, `exp()`, and `sqrt()`. In `tf.keras`, these functions generally just call the corresponding TensorFlow operations. If you want to write code that will be portable to other Keras implementations, you should use these Keras functions. However, they only cover a subset of all functions available in TensorFlow, so in this book we will use the TensorFlow operations directly. Here is a simple example using `keras.backend`, which is commonly named `K` for short:

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

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:

```
>>> 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)
```



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

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: you 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 (direct item assignment will not work) or by using the `scatter_update()` or `scatter_nd_update()` methods:

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



In practice you will rarely have to create variables manually, since Keras provides an `add_weight()` method that will take care of it for you, as we will 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 (please see the “Tensors and Operations” section in the notebook or [Appendix F](#) 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 size by default but can optionally be made dynamic. All tensors they contain must have the same shape and data type.

Ragged tensors (`tf.RaggedTensor`)

Represent static lists of lists of tensors, where every tensor has the same shape and data type. 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\xaa9"`). 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: simple First In, First Out (FIFO) queues (`FIFOQueue`), 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

Let's start by creating a custom loss function, which is a simple 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 not currently part of the official Keras API, but it is available in `tf.keras` (just use an instance of the `keras.losses.Huber` class). But let's pretend it's not there: implementing it is easy as pie! Just create a function that takes the labels and predictions as arguments, and use TensorFlow operations to compute every instance's loss:

```
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)
```



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

It is also preferable to return a tensor containing one loss per instance, rather than returning the mean loss. This way, Keras can apply class weights or sample weights when requested (see [Chapter 10](#)).

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

```
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 and use it to perform a Gradient Descent step. 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, as Keras saves the name of the function. Whenever 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 = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

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 = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

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

```

class HuberLoss(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}

```



The Keras API currently only specifies how to use subclassing to define layers, models, callbacks, and regularizers. If you build other components (such as losses, metrics, initializers, or constraints) using subclassing, they may not be portable to other Keras implementations. It's likely that the Keras API will be updated to specify subclassing for all these components as well.

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, it is `"sum_over_batch_size"`, which means that 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).⁵ 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 (note that the convenient `{**x}` syntax was added in Python 3.5).

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

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

⁵ 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.

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 = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

When you save a model, Keras calls the loss instance's `get_config()` method and saves the config as JSON in the HDF5 file. 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! That wasn't too hard, was it? Just as simple are custom activation functions, initializers, regularizers, and constraints. Let's look at these now.

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 very 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 `keras.activations.softplus()` or `tf.nn.softplus()`), a custom Glorot initializer (equivalent to `keras.initializers.glorot_normal()`), a custom ℓ_1 regularizer (equivalent to `keras.regularizers.l1(0.01)`), and a custom constraint that ensures weights are all positive (equivalent to `keras.constraints.nonneg()` or `tf.nn.relu()`):

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

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; for example:

```
layer = keras.layers.Dense(30, 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 `keras.regularizers.Regularizer`, `keras.constraints.Constraint`, `keras.initializers.Initializer`, or `keras.layers.Layer` (for any layer, including activation functions). Much like we did for the custom loss, here is a simple class for ℓ_1 regularization that saves its `factor` hyperparameter (this time we do not need to call the parent constructor or the `get_config()` method, as they are not defined by the parent class):

```
class MyL1Regularizer(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 we 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 where they are evaluated), and their gradients should not be 0 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 non-differentiable or have 0 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;⁶ 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"):

⁶ However, the Huber loss is seldom used as a metric (the MAE or MSE is preferred).

```
model.compile(loss="mse", optimizer="adam", 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 we 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 their ratio when requested. This is precisely what the `keras.metrics.Precision` class does:

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, 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: id=581780, 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 (note that we could also have passed sample weights). 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: id=581794, 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 create such a streaming metric, create a subclass of the `keras.metrics.Metric` class. Here is a simple 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 simply the mean Huber loss:

```
class HuberMetric(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):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        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:⁷

- 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).

⁷ This class is for illustration purposes only. A simpler and better implementation would just subclass the `keras.metrics.Mean` class; see the “Streaming metrics” section of the notebook for an example.



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 we have 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. In this case, you will need to create a custom layer. Or you may simply want to build a very repetitive architecture, containing identical blocks of layers repeated many times, and it would be convenient to treat each block of layers as a single layer. For example, if the model is a sequence of layers A, B, C, A, B, C, A, B, C, then you might want to define a custom layer D containing layers A, B, C, so your model would then simply be D, D, D. Let's see how to build custom layers.

First, some layers have no weights, such as `keras.layers.Flatten` or `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 `keras.layers.Lambda` layer. For example, the following layer will apply the exponential function to its inputs:

```
exponential_layer = 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`, `activation=keras.activations.exponential`, or simply `activation="exponential"`). 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.).

As you've probably guessed by now, to build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the `keras.layers.Layer` class. For example, the following class implements a simplified version of the `Dense` layer:

```

class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = 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")
        super().build(batch_input_shape) # must be at the end

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

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": 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 `keras.activations.get()` function (it accepts functions, standard strings like `"relu"` or `"selu"`, or simply `None`).⁸
- 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,⁹ 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

⁸ This function is specific to `tf.keras`. You could use `keras.layers.Activation` instead.

⁹ The Keras API calls this argument `input_shape`, but since it also includes the batch dimension, I prefer to call it `batch_input_shape`. Same for `compute_output_shape()`.

`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 `compute_output_shape()` method simply returns the shape of this layer's outputs. In this case, it is the same shape as the inputs, except the last dimension is replaced with the number of neurons in the layer. Note that in tf.keras, shapes are instances of the `tf.TensorShape` class, which you can convert to Python lists using `as_list()`.
- The `get_config()` method is just like in the previous custom classes. Note that we save the activation function's full configuration by calling `keras.activations.serialize()`.

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



You can generally omit the `compute_output_shape()` method, as tf.keras automatically infers the output shape, except when the layer is dynamic (as we will see shortly). In other Keras implementations, this method is either required or its default implementation assumes the output shape is the same as the input shape.

To create a layer with multiple inputs (e.g., `Concatenate`), the argument to the `call()` method should be a tuple containing all the inputs, and similarly the argument to the `compute_output_shape()` method should be a tuple containing each input's batch shape. To create a layer with multiple outputs, the `call()` method should return the list of outputs, and `compute_output_shape()` should return the list of batch output shapes (one per output). For example, the following toy layer takes two inputs and returns three outputs:

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

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1] # should probably handle broadcasting rules
```

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, `keras.layers.GaussianNoise`):

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

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

With that, you can now build any custom layer you need! Now let's create custom models.

Custom Models

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

¹⁰ The name "Subclassing API" usually refers only to the creation of custom models by subclassing, although many other things can be created by subclassing, as we saw in this chapter.

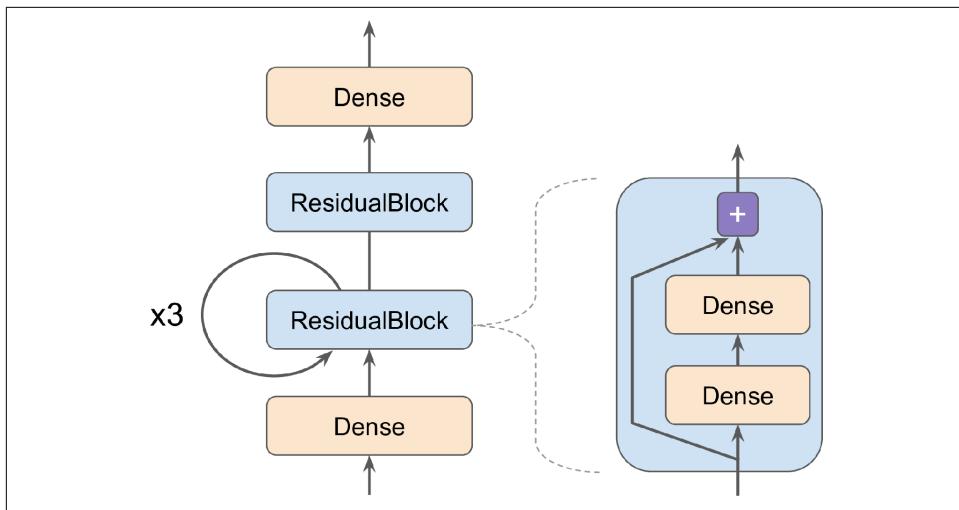


Figure 12-3. Custom model example: an arbitrary model with a custom `ResidualBlock` layer containing a skip connection

The inputs go through a first dense layer, then through a *residual block* composed of two dense layers and an addition operation (as we 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. Note that 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(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         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(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                         kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = 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 `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_layers()` method (which can return any of the model's layers by name or by index) and the `save()` method (and support for `keras.models.load_model()` and `keras.models.clone_model()`).



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 to 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). Here is the code for this custom model with a custom reconstruction loss:

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                         kernel_initializer="lecun_normal")
                      for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

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

    def call(self, inputs):
        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)
        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.
- The `build()` method creates an extra dense layer which 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.
- 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.¹¹ 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.
- Finally, the `call()` method passes the output of the hidden layers to the output layer and returns its output.

Similarly, you can add a custom metric based on model internals by computing it in any way you want, as long as the result is the output of a metric object. For example, you can create a `keras.metrics.Mean` object in the constructor, then call it in the `call()` method, passing it the `recon_loss`, and finally add it to the model by calling the model's `add_metric()` method. This way, when you train the model, Keras will display both the mean loss over each epoch (the loss is the sum of the main loss plus 0.05 times the reconstruction loss) and the mean reconstruction error over each epoch. Both will go down during training:

```
Epoch 1/5
11610/11610 [=====] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [=====] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

In over 99% of 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, in some rare cases you may need to customize the training loop

¹¹ You can also call `add_loss()` on any layer inside the model, as the model recursively gathers losses from all of its layers.

itself. Before we get there, we need to look at how to compute gradients automatically in TensorFlow.

Computing Gradients Using Autodiff

To understand how to use autodiff (see [Chapter 10](#) and [Appendix D](#)) 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 w_1 is $6 * w_1 + 2 * w_2$. You can also find that its partial derivative with regard to w_2 is $2 * w_1$. For example, at the point $(w_1, w_2) = (5, 3)$, 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 an almost 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:

```
>>> 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
10.00000003174137
```

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 $f(w_1, w_2)$ just once). Needing to call $f()$ at least once per parameter makes this approach intractable for large neural networks. So instead, we should use 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 w_1 and w_2 , 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 $[w_1, w_2]$. Let's take a look at the gradients that TensorFlow computed:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, 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!



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) # => tensor 36.0  
dz_dw2 = tape.gradient(z, w2) # 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:¹²

```
with tf.GradientTape(persistent=True) as tape:  
    z = f(w1, w2)  
  
dz_dw1 = tape.gradient(z, w1) # => tensor 36.0  
dz_dw2 = tape.gradient(z, w2) # => 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:

¹² 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.

```

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 would 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 with Autodiff” section of the 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) # same result as without 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 `my_softplus()` function for large inputs, the result will be NaN:

```

>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>

```

This is because computing the gradients of this function using autodiff leads to some numerical difficulties: due to floating-point precision errors, autodiff ends up computing infinity divided by infinity (which returns NaN). Fortunately, we can analytically find that the derivative of the softplus function is just $1 / (1 + 1 / \exp(x))$, which is numerically stable. Next, we can tell TensorFlow to use this stable function when computing the gradients of the `my_softplus()` function by decorating it with `@tf.custom_gradient` and making it return both its normal output and the function that computes the derivatives (note that it will receive as input the gradients that were backpropagated so far, down to the softplus function; and according to the chain rule, we should multiply them with this function's gradients):

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

Now when we compute the gradients of the `my_better_softplus()` function, we get the proper result, even for large input values (however, the main output still explodes because of the exponential; one workaround is to use `tf.where()` to return the inputs when they are large).

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, as we will see now.

Custom Training Loops

In some rare 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.



Unless 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. No need to compile it, since we will handle the training loop manually:

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                      kernel_regularizer=l2_reg),
    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 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 (i.e., we will use the `Mean` metric to compute it), and other metrics:

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

This code is self-explanatory, unless you are unfamiliar with Python string formatting: `{:.4f}` will format 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. In the notebook, the `print_status_bar()` function includes a progress bar, but you could use the handy `tqdm` library instead.

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 = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
```

```
mean_loss = keras.metrics.Mean()
metrics = [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 * batch_size, len(y_train), mean_loss, metrics)
            print_status_bar(len(y_train), len(y_train), 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 gradient 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 display the status bar again to make it look complete¹³ and to print a line feed, and we reset the states of the mean loss and the metrics.

If you set the optimizer’s `clipnorm` or `clipvalue` hyperparameter, it will take care of this for you. If you want to apply any other transformation to the gradients, simply do so before calling the `apply_gradients()` method.

If you 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()`:

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

Most importantly, this training loop does not handle layers that behave differently during training and testing (e.g., BatchNormalization or Dropout). To handle these, you need to call the model with `training=True` and make sure it propagates this to every layer that needs it.

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, so it’s your call.

Now that you know how to customize any part of your models¹⁴ 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

In TensorFlow 1, graphs were unavoidable (as were the complexities that came with them) because they were a central part of TensorFlow’s API. In TensorFlow 2, they 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
```

¹³ The truth is we did not process every single instance in the training set, because we sampled instances randomly: some were processed more than once, while others were not processed at all. Likewise, if the training set size is not a multiple of the batch size, we will miss a few instances. In practice that’s fine.

¹⁴ With the exception of optimizers, as very few people ever customize these; see the “Custom Optimizers” section in the notebook for an example.

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: id=18634148, 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 0x1546fc080>
```

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

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, 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 see 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.¹⁵ 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!

¹⁵ 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()`.

Moreover, 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 did throughout this chapter), Keras automatically converts your function into a TF Function—no need to use `tf.function()`. So most of the time, all this magic is 100% transparent.



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.



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.

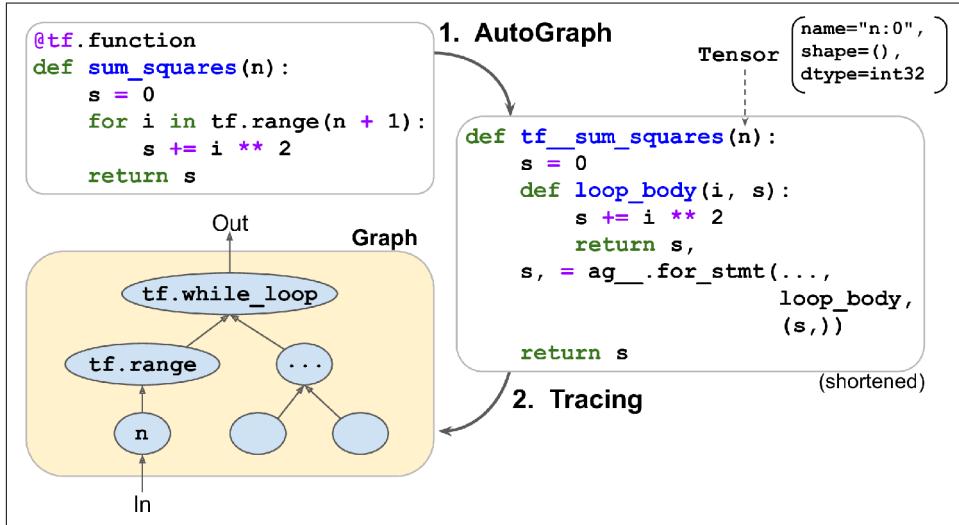


Figure 12-4. How TensorFlow generates graphs using AutoGraph and tracing

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. This should feel familiar if you know TensorFlow 1, as graph mode was the default mode. 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).



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 dataset. So 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 `tf.keras`. 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, for example to explore the generated graphs, you will find technical details in [Appendix G](#)).

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.arange(10))`?
4. Can you name six other data structures available in TensorFlow, beyond regular tensors?

5. A custom loss function can be defined by writing a function or by subclassing the `keras.losses.Loss` class. When would you use each option?
6. Similarly, a custom metric can be defined in a function or a subclass of `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 $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$, where \otimes represents itemwise multiplication (*) and ϵ is a smoothing term (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 `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 in [Appendix A](#).

Loading and Preprocessing Data with TensorFlow

So far we have used only datasets that fit in memory, but Deep Learning systems are often trained on very large datasets that will not fit in RAM. Ingesting a large dataset and preprocessing it efficiently can be tricky to implement with other Deep Learning libraries, but TensorFlow makes it easy thanks to the *Data API*: you just create a dataset object, and tell it where to get the data and how to transform it. TensorFlow takes care of all the implementation details, such as multithreading, queuing, batching, and prefetching. Moreover, the Data API works seamlessly with tf.keras!

Off the shelf, the 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 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.

Reading huge datasets efficiently is not the only difficulty: the data also needs to be preprocessed, usually normalized. Moreover, it is not always composed strictly of convenient numerical fields: there may be text features, categorical features, and so on. These need to be encoded, for example using one-hot encoding, bag-of-words encoding, or *embeddings* (as we will see, an embedding is a trainable dense vector that represents a category or token). One option to handle all this preprocessing is to write your own custom preprocessing layers. Another is to use the standard preprocessing layers provided by Keras.

In this chapter, we will cover the Data API, the TFRecord format, and how to create custom preprocessing layers and use the standard Keras ones. We will also take a quick look at a few related projects from TensorFlow's ecosystem:

TF Transform (tf.Transform)

Makes it possible to write a single preprocessing function that can be run in batch mode on your full training set, before training (to speed it up), and then exported to a TF Function and incorporated into your trained model so that once it is deployed in production it can take care of preprocessing new instances on the fly.

TF Datasets (TFDS)

Provides a convenient function to download many common datasets of all kinds, including large ones like ImageNet, as well as convenient dataset objects to manipulate them using the Data API.

So let's get started!

The Data API

The whole Data API revolves around the concept of a *dataset*: as you might suspect, 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 entirely in RAM using `tf.data.Dataset.from_tensor_slices()`:

```
>>> 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)`.

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(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

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 = 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)
```

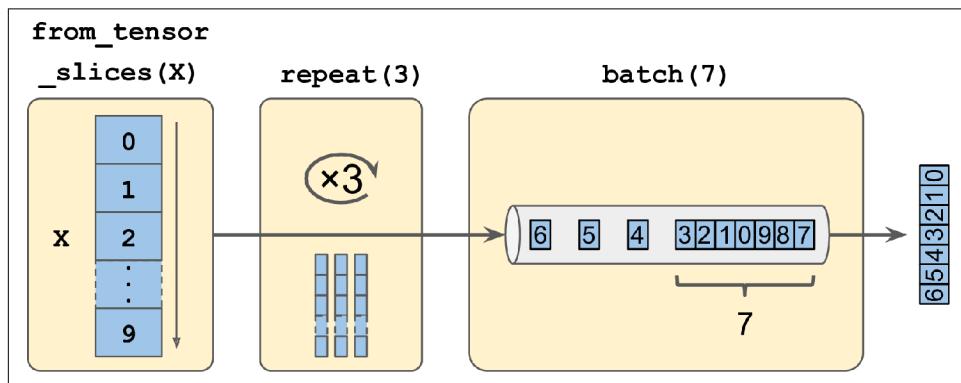


Figure 13-1. Chaining dataset transformations

In this example, we first call the `repeat()` method on the original dataset, and it returns a new dataset that will repeat 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. Finally, we iterate over the items of this final dataset. As you can see, the `batch()` method had to output a final batch of size two instead of seven, but you can call it with `drop_remainder=True` if you want it to drop this final batch so that all batches have the exact same size.



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 items doubled:

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

This function is the one you will call to apply any preprocessing you want 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: it's as simple as setting the `num_parallel_calls` argument. Note that the function you pass to the `map()` method must be convertible to a TF Function (see [Chapter 12](#)).

While the `map()` method applies a transformation to each item, the `apply()` method applies a transformation to the dataset as a whole. For example, the following code applies the `unbatch()` function to the dataset (this function is currently experimental, but it will most likely move to the core API in a future release). Each item in the new dataset will be a single-integer tensor instead of a batch of seven integers:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

It is also possible to simply filter the dataset using the `filter()` method:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

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(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Shuffling the Data

As you know, Gradient Descent works best when the instances in the training set are independent and identically distributed (see [Chapter 4](#)). 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 continues 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, and 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 3 times, shuffled using a buffer of size 5 and a random seed of 42, and batched with a batch size of 7:

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



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_iteration=False`.

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 all

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?

this sounds like a lot of work, don't worry: the Data API makes all this possible in just a few lines of code. Let's see how to do this.

Interleaving lines from multiple files

First, let's suppose that 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  
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442  
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687  
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621  
[...]
```

Let's also suppose `train_filepaths` contains the list of training file paths (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 file paths:

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

By default, the `list_files()` function returns a dataset that shuffles the file paths. 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 (skipping 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 file paths 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 we 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 get

the next five file paths from the `filepath_dataset` and interleave them the same way, and so on until it runs out of file paths.



For interleaving to work best, it is preferable to have files of identical length; otherwise the ends of the longest files 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 `num_parallel_calls` argument to the number of threads you want (note that the `map()` method also has this argument). You can even set it to `tf.data.experimental.AUTOTUNE` to make TensorFlow choose the right number of threads dynamically based on the available CPU (however, this is an experimental feature for now). Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

These are the first rows (ignoring the header row) of five CSV files, chosen randomly. Looks good! But as you can see, these are just byte strings; we need to parse them and scale the data.

Preprocessing the Data

Let's implement a small function that will perform this preprocessing:

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

def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    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.
- The `preprocess()` function takes one CSV line and starts by parsing it. For this 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 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 0, 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 `decode_csv()` function returns a list of scalar tensors (one per column), but we need to return 1D tensor arrays. 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).
- Finally, we scale the input features by subtracting the feature means and then dividing by the feature standard deviations, and we return 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: id=6227, 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: [...], numpy=array([2.782], dtype=float32)>)
```

Looks good! We can now apply the function to the dataset.

Putting Everything Together

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

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                      n_read_threads=None, shuffle_buffer_size=10000,
                      n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    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).repeat(repeat)
return dataset.batch(batch_size).prefetch(1)

```

Everything should make sense in this code, except the very last line (`prefetch(1)`), which is important for performance.

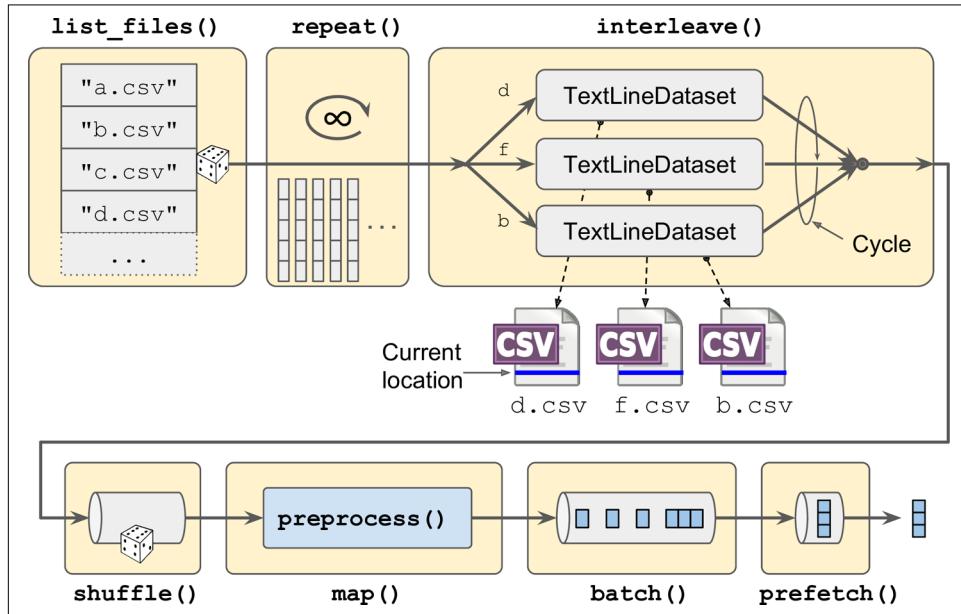


Figure 13-2. Loading and preprocessing data from multiple CSV files

Prefetching

By calling `prefetch(1)` at the end, 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 cores on the CPU and hopefully make preparing one batch of data shorter than running a training step on the GPU:

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.experimental.AUTOTUNE` (this is an experimental feature for now).

this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU³), and training will run much faster.

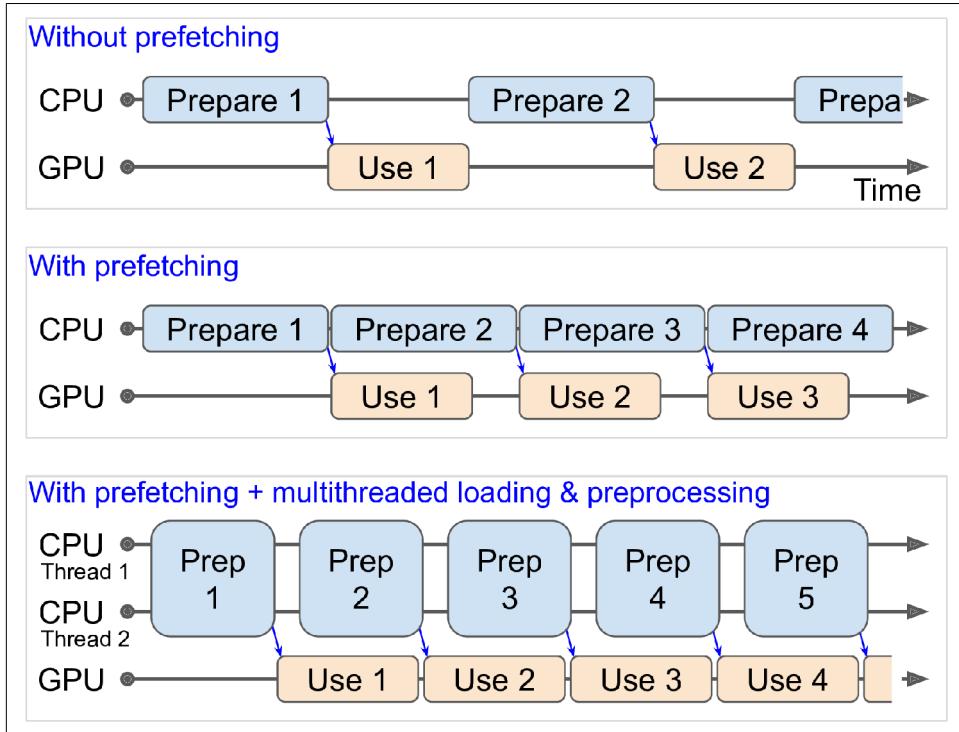


Figure 13-3. With prefetching, the CPU and the GPU work in parallel: as the GPU works on one batch, the CPU works on the next



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 computer vision). Just as important to get good performance is its *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

³ But check out the `tf.data.experimental.prefetch_to_device()` function, which can prefetch data directly to the GPU.

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 now know 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: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()`, and `padded_batch()`. There are also a couple more class methods: `from_generator()` and `from_tensors()`, 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 tf.keras

Now we can use the `csv_reader_dataset()` function to create a dataset for the training set. Note that we do not need to repeat it, as this will be taken care of by `tf.keras`. We also create datasets for the validation set and the test set:

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

And now we can simply build and train a Keras model using these datasets.⁴ All we need to do is pass the training and validation datasets to the `fit()` method, instead of `X_train`, `y_train`, `X_valid`, and `y_valid`:⁵

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

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

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) # pretend we have 3 new instances
model.predict(new_set) # a dataset containing new instances
```

Unlike the other sets, the `new_set` will usually not contain labels (if it does, Keras will ignore them). Note that in all these cases, you can still use NumPy arrays instead of

⁴ Support for datasets is specific to `tf.keras`; this will not work in other implementations of the Keras API.

⁵ The `fit()` method will take care of repeating the training dataset. Alternatively, you could call `repeat()` on the training dataset so that it repeats forever and specify the `steps_per_epoch` argument when calling the `fit()` method. This may be useful in some rare cases, for example if you want to use a shuffle buffer that crosses over epochs.

datasets if you want (but of course they need to have been loaded and preprocessed first).

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

```
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 performs the whole training loop:

```
@tf.function  
def train(model, optimizer, loss_fn, n_epochs, [...]):  
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])  
    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)  
            grads = tape.gradient(loss, model.trainable_variables)  
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Congratulations, you now know how to build powerful input pipelines using the Data API! However, so far we have used 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.



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)
```



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 setting `num_parallel_reads`. 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:
    [...]
```

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 definition says we are using version 3 of the protobuf format, and it specifies that each `Person` object⁶ 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 we will use have already been compiled for you, and their Python classes are part of TensorFlow, 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
>>> s = person.SerializeToString() # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # create a new Person
>>> person2.ParseFromString(s) # 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.⁷

We could save the serialized `Person` object to a TFRecord file, then we could load and parse it: everything would work fine. However, `SerializeToString()` and `ParseFrom`

⁶ Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

⁷ 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>.

`String()` are not TensorFlow operations (and neither are the other operations in this code), so they cannot be included in a TensorFlow Function (except by wrapping them in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in [Chapter 12](#)). Fortunately, TensorFlow does include special protobuf definitions for which it provides parsing operations.

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:

```
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.⁸ Here is how you could create a `tf.train.Example` representing the same person as earlier and write it to a TFRecord file:

```
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"])),
```

⁸ 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.

```
        "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 it's rather straightforward (and 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:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:  
    f.write(person_example.SerializeToString())
```

Normally you would write much more than one `Example`! 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 a serialized `Example`, 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()`. This is a TensorFlow operation, so it can be included in a TF Function. 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 it iterates over the TFRecord Dataset and parses the serialized `Example` protobuf 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),  
}  
  
for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):  
    parsed_example = tf.io.parse_single_example(serialized_example,  
                                              feature_description)
```

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'], [...])>
```

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()`.

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()`:

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)
```

As you can see, the `Example` protobuf 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, and parse data, the next step is to prepare it so that it can be fed to a neural network.

Preprocessing the Input Features

Preparing your data for a neural network requires converting all features into numerical features, generally normalizing them, and more. In particular, if your data contains categorical features or text features, they need to be converted to numbers. This can be done ahead of time when preparing your data files, using any tool you like (e.g., NumPy, pandas, or Scikit-Learn). Alternatively, you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier), or you can include a preprocessing layer directly in your model. Let's look at this last option now.

For example, here is how you can implement a standardization layer using a `Lambda` layer. For each feature, it subtracts the mean and divides by its standard deviation (plus a tiny smoothing term to avoid division by zero):

```
means = np.mean(X_train, axis=0, keepdims=True)  
stds = np.std(X_train, axis=0, keepdims=True)  
eps = keras.backend.epsilon()  
model = keras.models.Sequential([  
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),  
    [...] # other layers  
])
```

That's not too hard! However, you may prefer to use a nice self-contained custom layer (much like Scikit-Learn's `StandardScaler`), rather than having global variables like `means` and `stds` dangling around:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

Before you can use this standardization layer, you will need to adapt it to your dataset by calling the `adapt()` method and passing it a data sample. This will allow it to use the appropriate mean and standard deviation for each feature:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

This sample must be large enough to be representative of your dataset, but it does not have to be the full training set: in general, a few hundred randomly selected instances will suffice (however, this depends on your task). Next, you can use this preprocessing layer like a normal layer:

```
model = keras.Sequential()
model.add(std_layer)
[...] # create the rest of the model
model.compile([...])
model.fit([...])
```

If you are thinking that Keras should contain a standardization layer like this one, here's some good news for you: by the time you read this, the `keras.layers.Normalization` layer will probably be available. It will work very much like our custom Standardization layer: first, create the layer, then adapt it to your dataset by passing a data sample to the `adapt()` method, and finally use the layer normally.

Now let's look at categorical features. We will start by encoding them as one-hot vectors.

Encoding Categorical Features Using One-Hot Vectors

Consider the `ocean_proximity` feature in the California housing dataset we explored in [Chapter 2](#): it is a categorical feature with five possible values: "`<1H OCEAN`", "`INLAND`", "`NEAR OCEAN`", "`NEAR BAY`", and "`ISLAND`". We need to encode this feature before we feed it to a neural network. Since there are very few categories, we can use one-hot encoding. For this, we first need to map each category to its index (0 to 4), which can be done using a lookup table:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
```

```

table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)

```

Let's go through this code:

- We first define the *vocabulary*: this is the list of all possible categories.
- Then we create a tensor with the corresponding indices (0 to 4).
- Next, we create an initializer for the lookup table, passing it the list of categories and their corresponding indices. In this example, we already have this data, so we use a `KeyValueTensorInitializer`; but if the categories were listed in a text file (with one category per line), we would use a `TextFileInitializer` instead.
- In the last two lines we create the lookup table, giving it the initializer and specifying the number of *out-of-vocabulary* (oov) buckets. If we look up a category that does not exist in the vocabulary, the lookup table will compute a hash of this category and use it to assign the unknown category to one of the oov buckets. Their indices start after the known categories, so in this example the indices of the two oov buckets are 5 and 6.

Why use oov buckets? Well, if the number of categories is large (e.g., zip codes, cities, words, products, or users) and the dataset is large as well, or it keeps changing, then getting the full list of categories may not be convenient. One solution is to define the vocabulary based on a data sample (rather than the whole training set) and add some oov buckets for the other categories that were not in the data sample. The more unknown categories you expect to find during training, the more oov buckets you should use. Indeed, if there are not enough oov buckets, there will be collisions: different categories will end up in the same bucket, so the neural network will not be able to distinguish them (at least not based on this feature).

Now let's use the lookup table to encode a small batch of categorical features to one-hot vectors:

```

>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.]], dtype=float32)>

```

As you can see, "NEAR BAY" was mapped to index 3, the unknown category "DESERT" was mapped to one of the two oov buckets (at index 5), and "INLAND" was mapped to

index 1, twice. Then we used `tf.one_hot()` to one-hot encode these indices. Notice that we have to tell this function the total number of indices, which is equal to the vocabulary size plus the number of oov buckets. Now you know how to encode categorical features to one-hot vectors using TensorFlow!

Just like earlier, it wouldn't be too difficult to bundle all of this logic into a nice self-contained class. Its `adapt()` method would take a data sample and extract all the distinct categories it contains. It would create a lookup table to map each category to its index (including unknown categories using oov buckets). Then its `call()` method would use the lookup table to map the input categories to their indices. Well, here's more good news: by the time you read this, Keras will probably include a layer called `keras.layers.TextVectorization`, which will be capable of doing exactly that: its `adapt()` method will extract the vocabulary from a data sample, and its `call()` method will convert each category to its index in the vocabulary. You could add this layer at the beginning of your model, followed by a `Lambda` layer that would apply the `tf.one_hot()` function, if you want to convert these indices to one-hot vectors.

This may not be the best solution, though. The size of each one-hot vector is the vocabulary length plus the number of oov buckets. This is fine when there are just a few possible categories, but if the vocabulary is large, it is much more efficient to encode them using *embeddings* instead.



As a rule of thumb, if the number of categories is lower than 10, then one-hot encoding is generally the way to go (but your mileage may vary!). If the number of categories is greater than 50 (which is often the case when you use hash buckets), then embeddings are usually preferable. In between 10 and 50 categories, you may want to experiment with both options and see which one works best for your use case.

Encoding Categorical Features Using Embeddings

An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the "NEAR BAY" category 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, 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 13-4](#)). 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* (we will see other types of representation learning in [Chapter 17](#)).

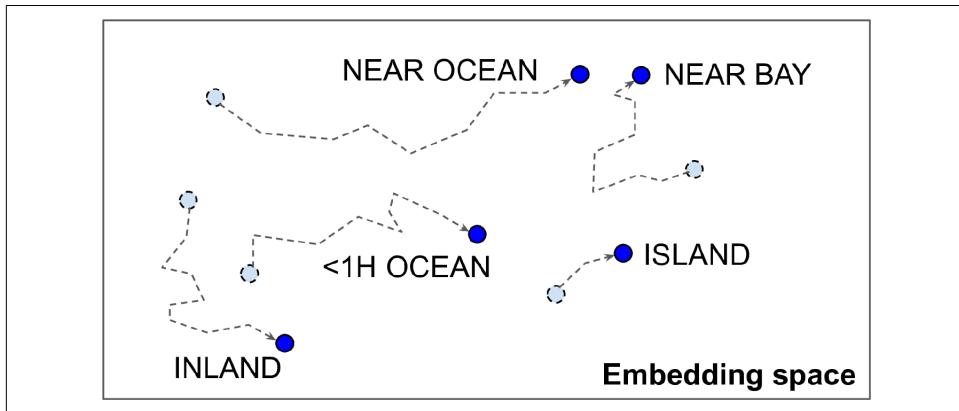


Figure 13-4. Embeddings will gradually improve during training

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](#)⁹ 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 13-5](#)). In other words, the word embeddings encode the concept of gender!

⁹ Tomas Mikolov et al., “Distributed Representations of Words and Phrases and Their Compositionalities,” *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013): 3111–3119.

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.

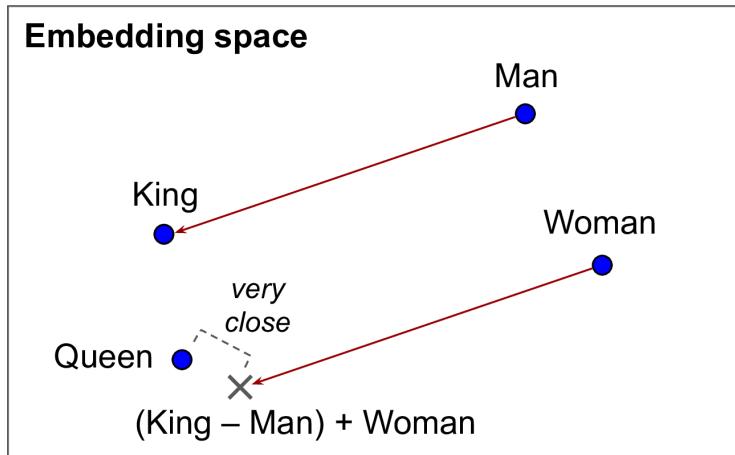


Figure 13-5. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

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 paper¹⁰](#) by Malvina Nissim et al. Nevertheless, ensuring fairness in Deep Learning algorithms is an important and active research topic.

Let's look at how we could implement embeddings manually, to understand how they work (then we will use a simple Keras layer instead). First, we need to create an *embedding matrix* containing each category's embedding, initialized randomly; it will have one row per category and per oov bucket, and one column per embedding dimension:

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

¹⁰ Malvina Nissim et al., “Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor,” arXiv preprint arXiv:1905.09866 (2019).

In this example we are using 2D embeddings, but as a rule of thumb embeddings typically have 10 to 300 dimensions, depending on the task and the vocabulary size (you will have to tune this hyperparameter).

This embedding matrix is a random 6×2 matrix, stored in a variable (so it can be tweaked by Gradient Descent during training):

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Now let's encode the same batch of categorical features as earlier, but this time using these embeddings:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

The `tf.nn.embedding_lookup()` function looks up the rows in the embedding matrix, at the given indices—that's all it does. For example, the lookup table says that the "INLAND" category is at index 1, so the `tf.nn.embedding_lookup()` function returns the embedding at row 1 in the embedding matrix (twice): $[0.3528825, 0.46448255]$.

Keras provides a `keras.layers.Embedding` layer that handles the embedding matrix (trainable, by default); when the layer is created it initializes the embedding matrix randomly, and then when it is called with some category indices it returns the rows at those indices in the embedding matrix:

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
...                                         output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

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

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

This model takes two inputs: a regular input containing eight numerical features per instance, plus a categorical input (containing one categorical feature per instance). It uses a `Lambda` layer to look up each category's index, then it looks up the embeddings for these indices. Next, it concatenates the embeddings and the regular inputs in order to give the encoded inputs, which are ready to be fed to a neural network. We could add any kind of neural network at this point, but we just add a dense output layer, and we create the Keras model.

When the `keras.layers.TextVectorization` layer is available, you can call its `adapt()` method to make it extract the vocabulary from a data sample (it will take care of creating the lookup table for you). Then you can add it to your model, and it will perform the index lookup (replacing the `Lambda` layer in the previous code example).



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 (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.

Now let's look a bit more closely at the Keras preprocessing layers.

Keras Preprocessing Layers

The TensorFlow team is working on providing a set of standard [Keras preprocessing layers](#). They will probably be available by the time you read this; however, the API may change slightly by then, so please refer to the notebook for this chapter if anything behaves unexpectedly. This new API will likely supersede the existing Feature

Columns API, which is harder to use and less intuitive (if you want to learn more about the Feature Columns API anyway, please check out the notebook for this chapter).

We already discussed two of these layers: the `keras.layers.Normalization` layer that will perform feature standardization (it will be equivalent to the `Standardization` layer we defined earlier), and the `TextVectorization` layer that will be capable of encoding each word in the inputs into its index in the vocabulary. In both cases, you create the layer, you call its `adapt()` method with a data sample, and then you use the layer normally in your model. The other preprocessing layers will follow the same pattern.

The API will also include a `keras.layers.Discretization` layer that will chop continuous data into different bins and encode each bin as a one-hot vector. For example, you could use it to discretize prices into three categories, (low, medium, high), which would be encoded as [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively. Of course this loses a lot of information, but in some cases it can help the model detect patterns that would otherwise not be obvious when just looking at the continuous values.



The `Discretization` layer will not be differentiable, and it should only be used at the start of your model. Indeed, the model's preprocessing layers will be frozen during training, so their parameters will not be affected by Gradient Descent, and thus they do not need to be differentiable. This also means that you should not use an `Embedding` layer directly in a custom preprocessing layer, if you want it to be trainable: instead, it should be added separately to your model, as in the previous code example.

It will also be possible to chain multiple preprocessing layers using the `PreprocessingStage` class. For example, the following code will create a preprocessing pipeline that will first normalize the inputs, then discretize them (this may remind you of Scikit-Learn pipelines). After you adapt this pipeline to a data sample, you can use it like a regular layer in your models (but again, only at the start of the model, since it contains a nondifferentiable preprocessing layer):

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

The `TextVectorization` layer will also have an option to output word-count vectors instead of word indices. For example, if the vocabulary contains three words, say ["and", "basketball", "more"], then the text "more and more" will be mapped to the vector [1, 0, 2]: the word "and" appears once, the word "basketball" does not appear at all, and the word "more" appears twice. This text representation is called a

bag of words, since it completely loses the order of the words. Common words like "and" will have a large value in most texts, even though they are usually the least interesting (e.g., in the text "more and more basketball" the word "basketball" is clearly the most important, precisely because it is not a very frequent word). So, the word counts should be normalized in a way that reduces the importance of frequent words. A common way to do this is to divide each word count by the log of the total number of training instances in which the word appears. This technique is called *Term-Frequency × Inverse-Document-Frequency* (TF-IDF). For example, let's imagine that the words "and", "basketball", and "more" appear respectively in 200, 10, and 100 text instances in the training set: in this case, the final vector will be $[1/\log(200), 0/\log(10), 2/\log(100)]$, which is approximately equal to $[0.19, 0., 0.43]$. The `TextVectorization` layer will (likely) have an option to perform TF-IDF.



If the standard preprocessing layers are insufficient for your task, you will still have the option to create your own custom preprocessing layer, much like we did earlier with the `Standardization` class. Create a subclass of the `keras.layers.PreprocessingLayer` class with an `adapt()` method, which should take a `data_sample` argument and optionally an `extra reset_state` argument: if `True`, then the `adapt()` method should reset any existing state before computing the new state; if `False`, it should try to update the existing state.

As you can see, these Keras preprocessing layers will make preprocessing much easier! Now, whether you choose to write your own preprocessing layers or use Keras's (or even use the Feature Columns API), all the preprocessing will be done on the fly. During training, however, it may be preferable to perform preprocessing ahead of time. Let's see why we'd want to do that and how we'd go about it.

TF Transform

If preprocessing is computationally expensive, then handling it before training rather than on the fly may give you a significant speedup: the data will be preprocessed just once per instance *before* training, rather than once per instance and per epoch *during* training. As mentioned earlier, if the dataset is small enough to fit in RAM, you can use its `cache()` method. But if it is too large, then tools like Apache Beam or Spark will help. They let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so you can use them to preprocess all the training data before training.

This works great and indeed can speed up training, but there is one problem: once your model is trained, suppose you want to deploy it to a mobile app. In that case you will need to write some code in your app to take care of preprocessing the data before

it is fed to the model. And suppose you also want to deploy the model to TensorFlow.js so that it runs in a web browser? Once again, you will need to write some pre-processing code. This can become a maintenance nightmare: whenever you want to change the preprocessing logic, you will need to update your Apache Beam code, your mobile app code, and your JavaScript code. This is not only time-consuming, but also error-prone: you may end up with subtle differences between the preprocessing operations performed before training and the ones performed in your app or in the browser. This *training/serving skew* will lead to bugs or degraded performance.

One improvement would be to take the trained model (trained on data that was pre-processed by your Apache Beam or Spark code) and, before deploying it to your app or the browser, add extra preprocessing layers to take care of preprocessing on the fly. That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam or Spark code, and the preprocessing layers' code.

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of [TensorFlow Extended \(TFX\)](#), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component such as TF Transform, you must install it; it does not come bundled with TensorFlow. You then define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

Next, TF Transform lets you apply this `preprocess()` function to the whole training set using Apache Beam (it provides an `AnalyzeAndTransformDataset` class that you can use for this purpose in your Apache Beam pipeline). In the process, it will also compute all the necessary statistics over the whole training set: in this example, the mean and standard deviation of the `housing_median_age` feature, and the vocabulary for the `ocean_proximity` feature. The components that compute these statistics are called *analyzers*.

Importantly, TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy. This TF Function includes some constants

that correspond to all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary).

With the Data API, TFRecords, the Keras preprocessing layers, and TF Transform, you can build highly scalable input pipelines for training and benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

The TensorFlow Datasets (TFDS) Project

The [TensorFlow Datasets](#) project makes it very easy to download 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), and audio and video datasets. You can visit <https://hml.info/tfds> to view the full list, along with a description of each dataset.

TFDS is not bundled with TensorFlow, so you need to install the `tensorflow-datasets` library (e.g., using pip). Then call the `tfds.load()` function, and it 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

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["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:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```



The `load()` function shuffles each data shard it downloads (only for the training set). This may not be sufficient, 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(10000).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). You can also specify the batch size if you want. Then you can pass the dataset directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

This was quite a technical chapter, and 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 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, or using TF Transform. Can you list a few pros and cons of each option?
8. Name a few common techniques you can use to encode categorical 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.¹¹ 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 movies reviews from the [Internet Movie Database](#). 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 pre-processed bag-of-words), 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. If the `TextVectorization` layer is not yet available (or if you like a challenge), try to create your own custom preprocessing layer: you can use the functions in the `tf.strings` package, for example `lower()` to make everything lowercase, `regex_replace()` to replace punctuation with spaces, and `split()` to split words on spaces. You should use a lookup table to output word indices, which must be prepared in the `adapt()` method.
 - 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 in [Appendix A](#).

¹¹ 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.

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 the sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few 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 TensorFlow and 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 1958¹ and 1959² (and a few years later on monkeys³), 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.

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 a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

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.

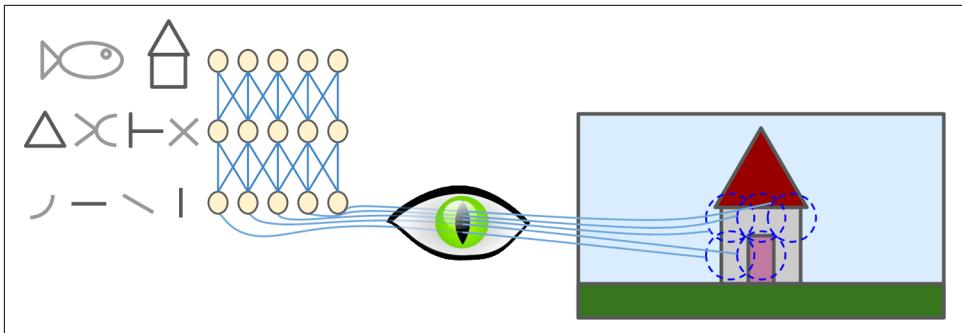


Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

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, widely used by banks to recognize handwritten check numbers. 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.



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.

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.

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.

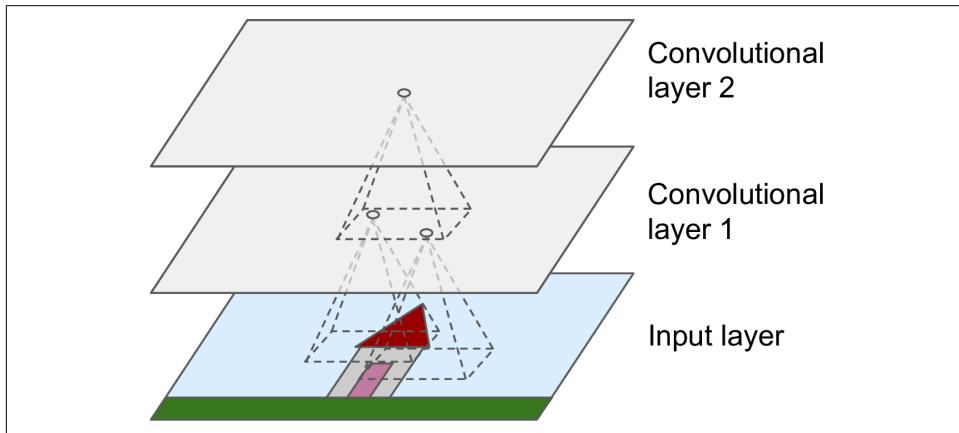


Figure 14-2. CNN layers with rectangular local receptive fields



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.

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).

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*.

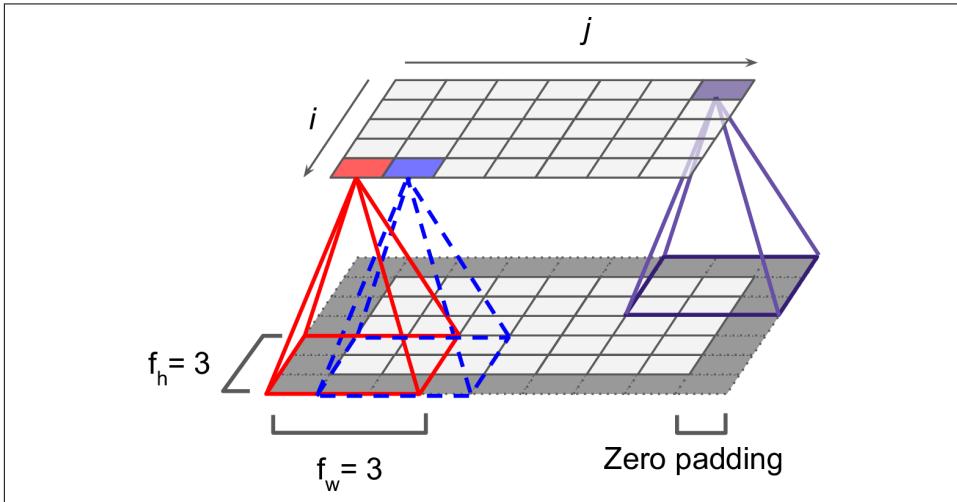


Figure 14-3. Connections between layers and zero padding

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 shift 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.

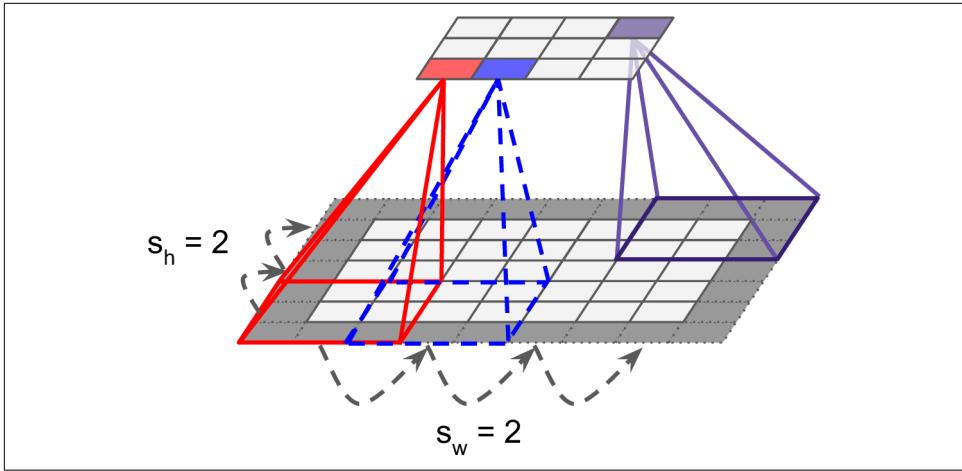


Figure 14-4. Reducing dimensionality using a stride of 2

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*). The first one is represented as a black square with a vertical white line in the middle (it is 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 get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

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. Of course, you do not 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.

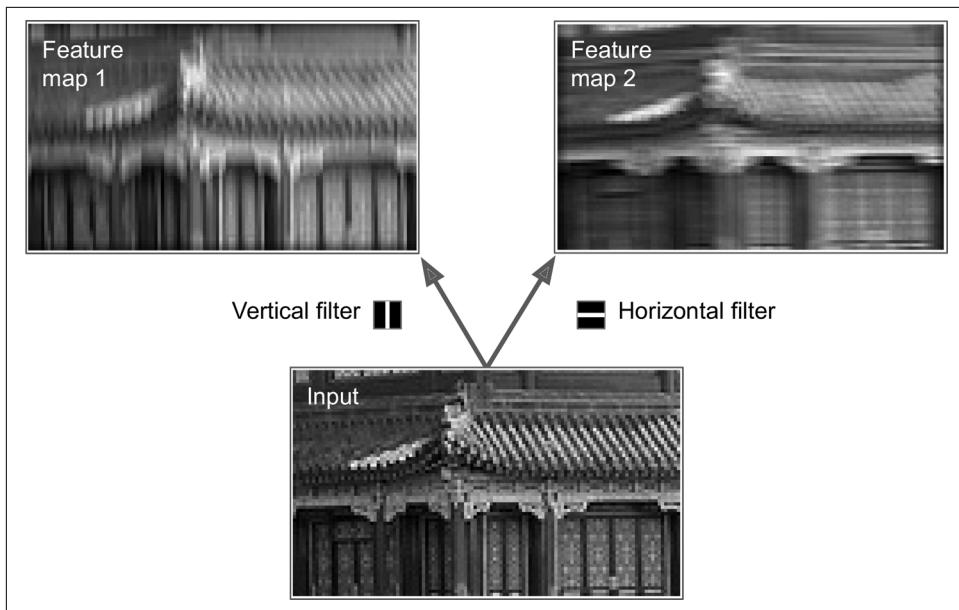


Figure 14-5. Applying two different filters to get two feature maps

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 weights 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 previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.



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 regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one

channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

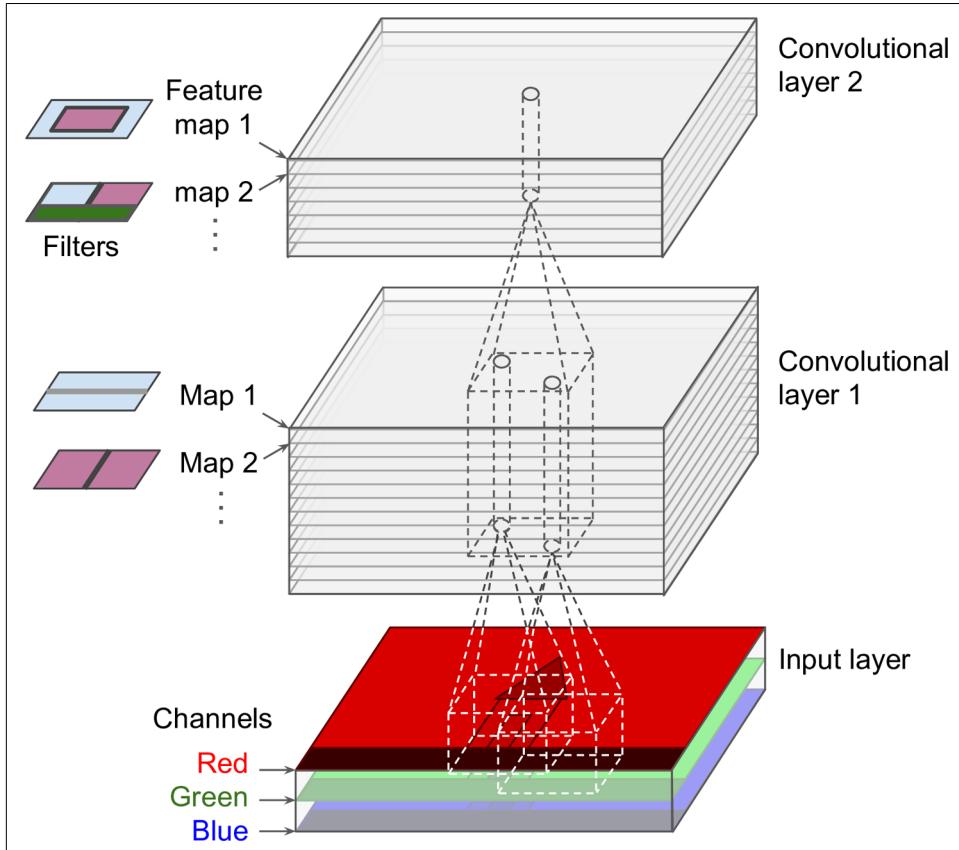


Figure 14-6. Convolutional layers with multiple feature maps, and images with three color channels

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 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

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

In this equation:

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

TensorFlow Implementation

In TensorFlow, each input image is typically represented as a 3D tensor of shape `[height, width, channels]`. A mini-batch is represented as a 4D tensor of shape `[mini-batch size, height, width, channels]`. The weights of a convolutional layer are represented as a 4D tensor of shape `[fh, fw, fn', fn]`. The bias terms of a convolutional layer are simply represented as a 1D tensor of shape `[fn]`.

Let's look at a simple example. The following code loads two sample images, using Scikit-Learn's `load_sample_image()` (which loads two color images, one of a Chinese temple, and the other of a flower), then it creates two filters and applies them to both images, and finally it displays one of the resulting feature maps. Note that you must pip install the `Pillow` package to use `load_sample_image()`.

```
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
```

```

images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()

```

Let's go through this code:

- The pixel intensity for each color channel is represented as a byte from 0 to 255, so we scale these features simply by dividing by 255, to get floats ranging from 0 to 1.
- Then we create two 7×7 filters (one with a vertical white line in the middle, and the other with a horizontal white line in the middle).
- We apply them to both images using the `tf.nn.conv2d()` function, which is part of TensorFlow's low-level Deep Learning API. In this example, we use zero padding (`padding="SAME"`) and a stride of 1.
- Finally, we plot one of the resulting feature maps (similar to the top-right image in [Figure 14-5](#)).

The `tf.nn.conv2d()` line deserves a bit more explanation:

- `images` is the input mini-batch (a 4D tensor, as explained earlier).
- `filters` is the set of filters to apply (also a 4D tensor, as explained earlier).
- `strides` is equal to 1, but it could also be a 1D array with four elements, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to 1. They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).
- `padding` must be either "SAME" or "VALID":
 - If set to "SAME", the convolutional layer uses zero padding if necessary. The output size is set to the number of input neurons divided by the stride, rounded up. For example, if the input size is 13 and the stride is 5 (see [Figure 14-7](#)), then the output size is 3 (i.e., $13 / 5 = 2.6$, rounded up to 3). Then zeros are added as evenly as possible around the inputs, as needed. When `strides=1`, the layer's outputs will have the same spatial dimensions (width and height) as its inputs, hence the name *same*.

- If set to "VALID", the convolutional layer does *not* use zero padding and may ignore some rows and columns at the bottom and right of the input image, depending on the stride, as shown in [Figure 14-7](#) (for simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension). This means that every neuron's receptive field lies strictly within valid positions inside the input (it does not go out of bounds), hence the name *valid*.

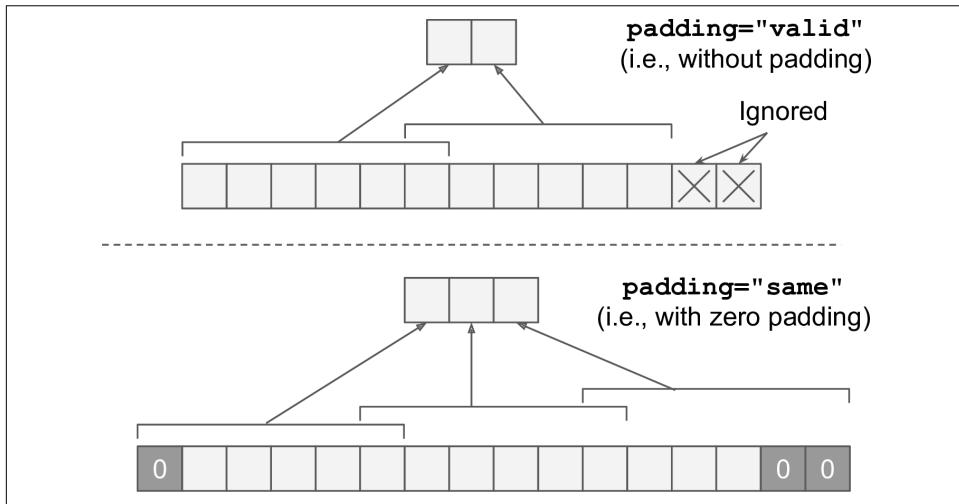


Figure 14-7. Padding="SAME" or "VALID" (with input width 13, filter width 6, stride 5)

In this example we manually defined the filters, but in a real CNN you would normally define filters as trainable variables so the neural net can learn which filters work best, as explained earlier. Instead of manually creating the variables, use the `keras.layers.Conv2D` layer:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

This code creates a `Conv2D` layer with 32 filters, each 3×3 , using a stride of 1 (both horizontally and vertically) and "same" padding, and applying the ReLU activation function to its outputs. As you can see, convolutional layers have quite a few hyperparameters: you must choose the number of filters, their height and width, the strides, and the padding type. 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, to give you some idea of which hyperparameter values work best in practice.

Memory Requirements

Another problem 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 5×5 filters, outputting 200 feature maps of size 150×100 , with stride 1 and "same" padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15,200$ (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 $5 \times 5 \times 3 = 75$ 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 $200 \times 150 \times 100 \times 32 = 96$ 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.



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, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

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

⁷ A fully connected layer with 150×100 neurons, each connected to all $150 \times 100 \times 3$ inputs, would have $150^2 \times 100^2 \times 3 = 675$ million parameters!

⁸ In the international system of units (SI), 1 MB = 1,000 KB = $1,000 \times 1,000$ bytes = $1,000 \times 1,000 \times 8$ bits.

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 14-8](#) 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 14-8](#), 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).

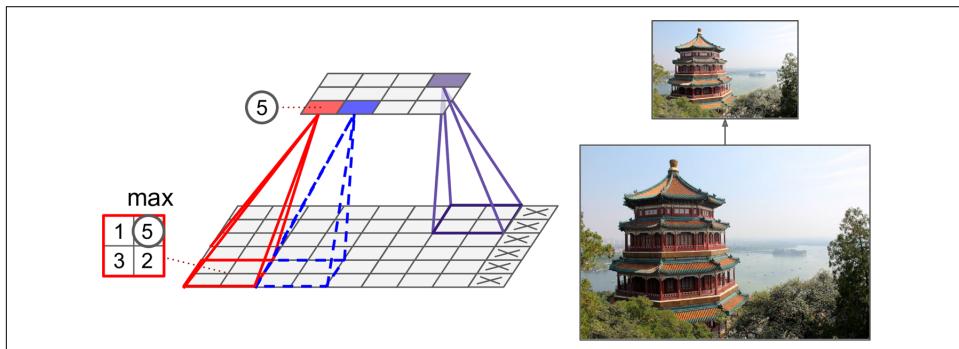


Figure 14-8. Max pooling layer (2×2 pooling kernel, stride 2, no padding)



A pooling layer typically works on every input channel independently, so the output depth 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 14-9](#). 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

⁹ Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

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 75% 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.

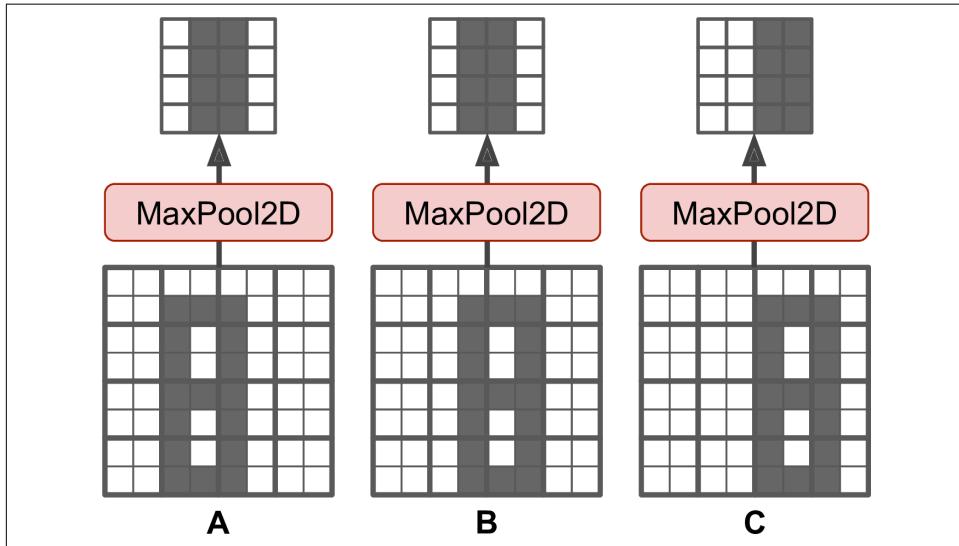


Figure 14-9. Invariance to small translations

However, max pooling has some downsides too. Firstly, it is 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.

TensorFlow Implementation

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a 2×2 kernel. The strides default to the kernel size, so this layer will use a stride of 2 (both horizontally and vertically). By default, it uses "valid" padding (i.e., no padding at all):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use `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 rather than the spatial dimensions, although this is 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 14-10](#)), 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 else: thickness, brightness, skew, color, and so on.

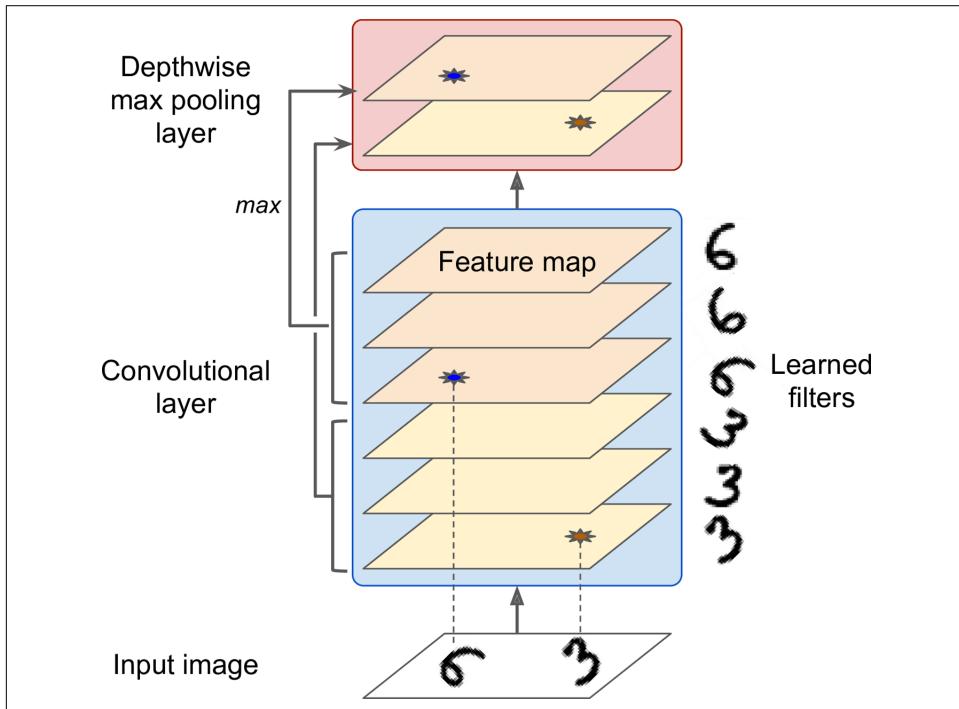


Figure 14-10. Depthwise max pooling can help the CNN learn any invariance

Keras does not include a depthwise max pooling layer, but TensorFlow's low-level Deep Learning API does: just use the `tf.nn.max_pool()` function, and specify the kernel size and strides as 4-tuples (i.e., tuples of size 4). The first three values of each should be 1: this indicates that the kernel size and stride along the batch, height, and width dimensions should be 1. The last value should be whatever kernel size and stride you want along the depth dimension—for example, 3 (this must be a divisor of the input depth; it will not work if the previous layer outputs 20 feature maps, since 20 is not a multiple of 3):

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

If you want to include this as a layer in your Keras models, wrap it in a `Lambda` layer (or create a custom Keras layer):

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                            padding="valid"))
```

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 as the output layer, as we will see later in this chapter. To create such a layer, simply use the `keras.layers.GlobalAvgPool2D` class:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

It's equivalent to this simple `Lambda` layer, which computes the mean over the spatial dimensions (height and width):

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

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 14-11](#)). 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).

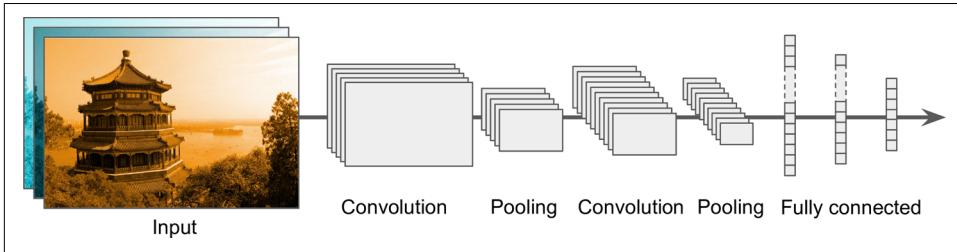


Figure 14-11. Typical CNN architecture



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 simple CNN to tackle the Fashion MNIST dataset (introduced in [Chapter 10](#)):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

Let's go through this model:

- The first layer uses 64 fairly large filters (7×7) but no stride 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).
- Next we have a max pooling layer which uses a 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 grows 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. Note that we must flatten its inputs, since a dense network 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.

This CNN reaches 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 fell from over 26% to less than 2.3% in just six years. The top-five error rate is the number of test images for which the system's top five predictions did not include the correct answer. The images are large (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.

We will first look at the classical LeNet-5 architecture (1998), then three of the winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), and ResNet (2015).

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

There are a few extra details to be noted:

- MNIST images are 28×28 pixels, but they are zero-padded to 32×32 pixels and normalized before being fed to the network. The rest of the network does not use any padding, which is why the size keeps shrinking as the image progresses through the network.
- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.
- Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps). See table 1 (page 8) in the original paper¹⁰ for details.
- The output layer is a bit special: instead of computing the matrix multiplication of the inputs and the weight vector, each neuron outputs the square of the Euclidean distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross-entropy

¹⁰ Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and converging faster.

Yann LeCun's [website](#) features great demos of LeNet-5 classifying digits.

AlexNet

The [AlexNet CNN architecture](#)¹¹ won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%! It 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	Stride	Padding	Activation
Out	Fully connected	—	1,000	—	—	—	Softmax
F10	Fully connected	—	4,096	—	—	—	ReLU
F9	Fully connected	—	4,096	—	—	—	ReLU
S8	Max pooling	256	6×6	3×3	2	valid	—
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Max pooling	256	13×13	3×3	2	valid	—
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Max pooling	96	27×27	3×3	2	valid	—
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	—	—	—	—

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

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.

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 14-12](#)). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. For 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 the size of your training set.

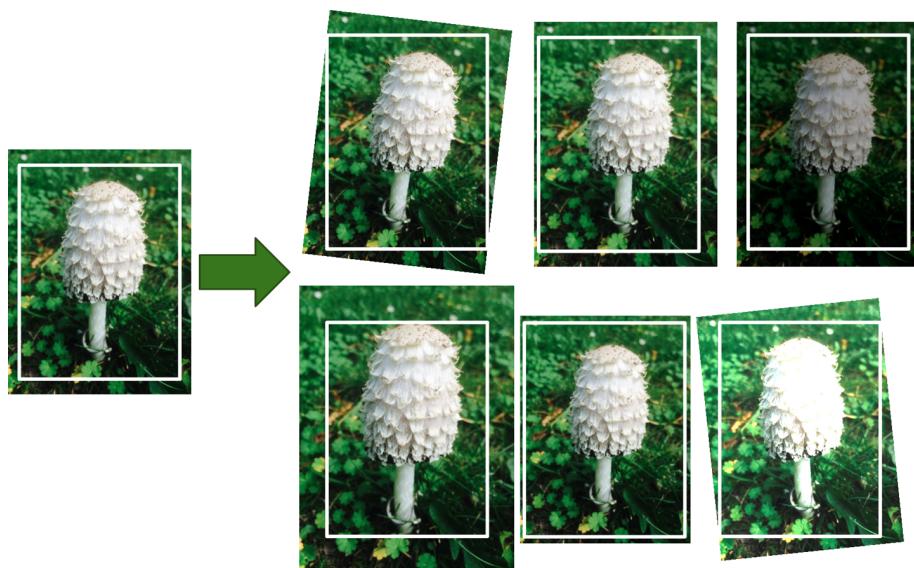


Figure 14-12. Generating new training instances from existing ones

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)

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{\text{high}} = \min \left(i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left(0, i - \frac{r}{2} \right) \end{cases}$$

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 follows: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$, and $k = 1$. This step can be implemented 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 14-14](#)). This was made

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.

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 14-13 shows the architecture of an inception module. The notation “ $3 \times 3 + 1(S)$ ” means that the layer uses a 3×3 kernel, stride 1, and "same" padding. The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function. Note that the second set of convolutional layers uses 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., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the `tf.concat()` operation, with `axis=3` (the axis is the depth).

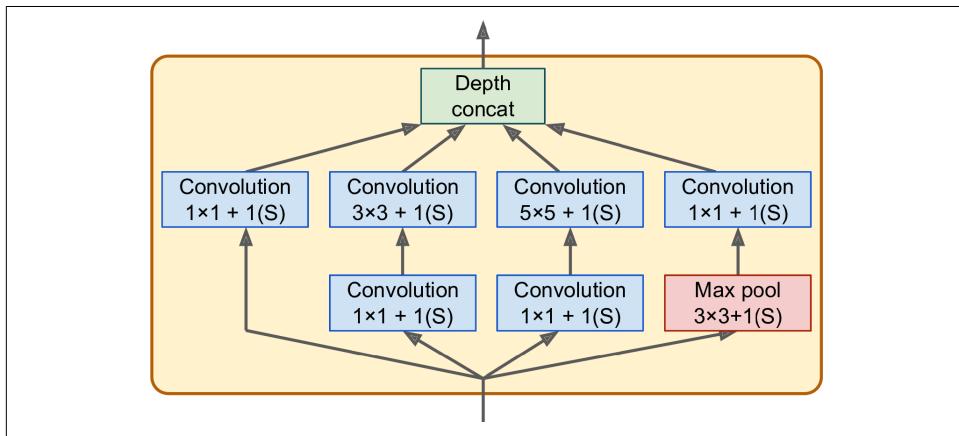


Figure 14-13. Inception module

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? In fact, the layers serve three purposes:

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the computa-

¹⁴ In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

tional cost and the number of parameters, speeding up training and improving generalization.

- Each pair of convolutional layers ($[1 \times 1, 3 \times 3]$ and $[1 \times 1, 5 \times 5]$) acts like a single powerful convolutional layer, capable of capturing more complex patterns. Indeed, instead of sweeping a simple linear classifier across the image (as a single convolutional layer does), this pair of convolutional layers sweeps a two-layer neural network 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.



The number of convolutional kernels for each convolutional layer is a hyperparameter. Unfortunately, this means that you have six more hyperparameters to tweak for every inception layer you add.

Now let's look at the architecture of the GoogLeNet CNN (see [Figure 14-14](#)). 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 14-13](#)). Note that all the convolutional layers use the ReLU activation function.

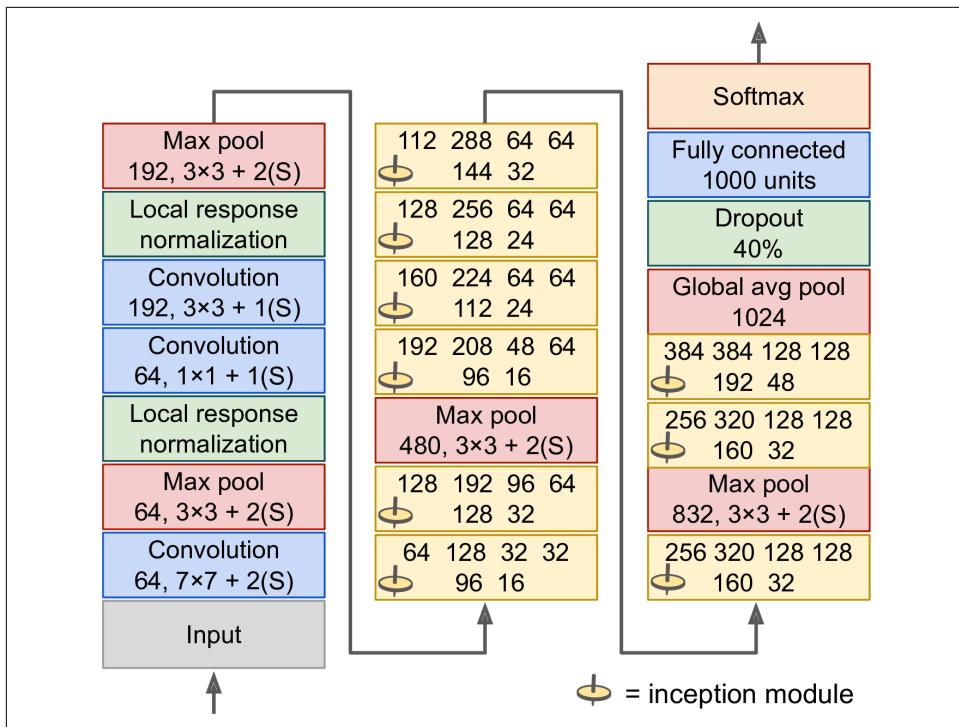


Figure 14-14. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size 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 explained earlier, 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 tall stack of nine inception modules, interleaved with a couple 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 was 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, it is a classification task, not localization, so it does not 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.

This diagram is slightly simplified: the original GoogLeNet architecture also 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. However, 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 and reaching even better performance.

VGGNet

The runner-up in the ILSVRC 2014 challenge was **VGGNet**,¹⁵ developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University. It had a very simple and classical architecture, with 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 just 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 only 3×3 filters, but many filters.

¹⁵ Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” arXiv preprint arXiv:1409.1556 (2014).

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a *Residual Network* (or *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: models are 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 a bit 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 14-15).

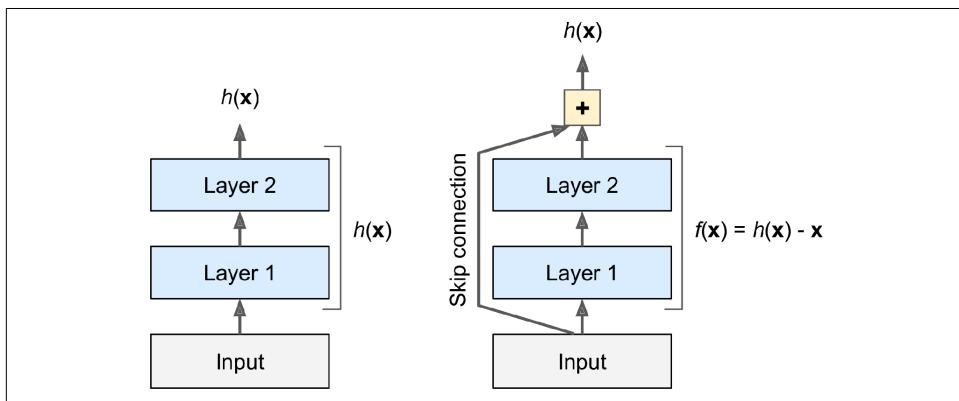


Figure 14-15. Residual learning

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 14-16). 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.

¹⁶ Kaiming He et al., “Deep Residual Learning for Image Recognition,” arXiv preprint arXiv:1512:03385 (2015).

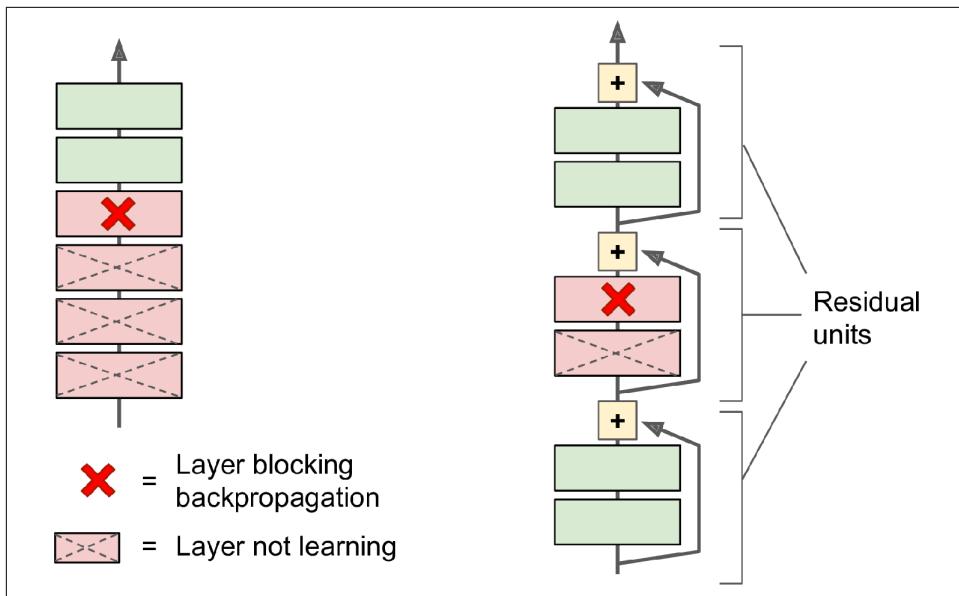


Figure 14-16. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see [Figure 14-17](#)). 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 simple 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).

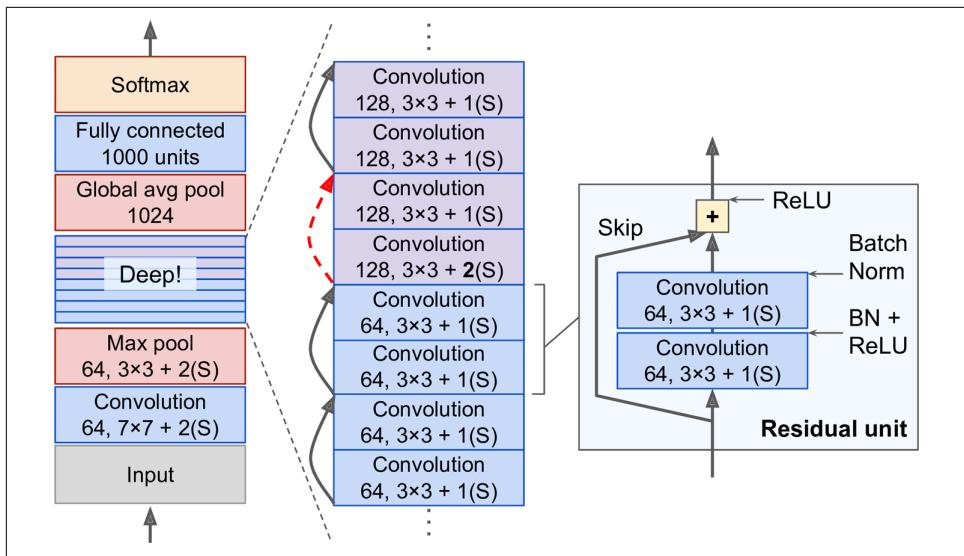


Figure 14-17. ResNet architecture

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 14-17). 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 14-18).

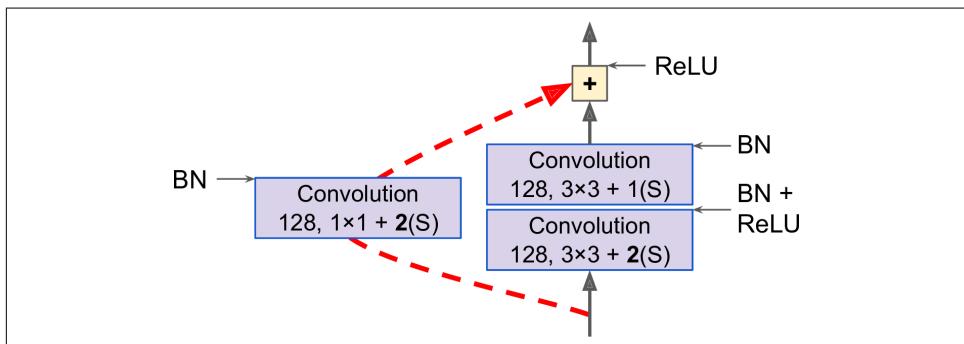


Figure 14-18. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)¹⁷ containing 3 residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

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



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.

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 short²⁰). 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 14-19](#)). Thus, it is composed of two parts: the first part applies a single spatial filter for each input feature map, then the second part looks

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.

exclusively for cross-channel patterns—it is just a regular convolutional layer with 1×1 filters.

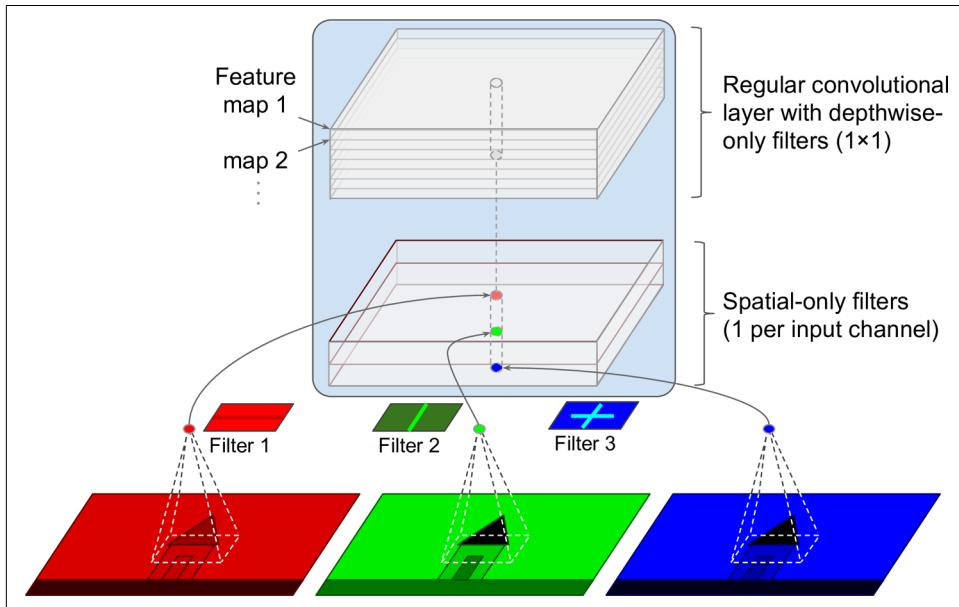


Figure 14-19. Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 14-19 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 module at all. Well, as we 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 generally perform better.



Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default (except after layers with few channels).

The ILSVRC 2016 challenge was won by the CUImage team from the Chinese University of Hong Kong. They used an ensemble of many different techniques, including a sophisticated object-detection system called **GBD-Net**,²¹ to achieve a top-five error rate below 3%. Although this result is unquestionably impressive, the complexity of the solution contrasted with the simplicity of ResNets. Moreover, one year later another fairly simple architecture performed even better, as we will see now.

SENet

The winning architecture in the ILSVRC 2017 challenge was the **Squeeze-and-Excitation Network (SENet)**.²² 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 unit in the original architecture (i.e., every inception module or every residual unit), as shown in Figure 14-20.

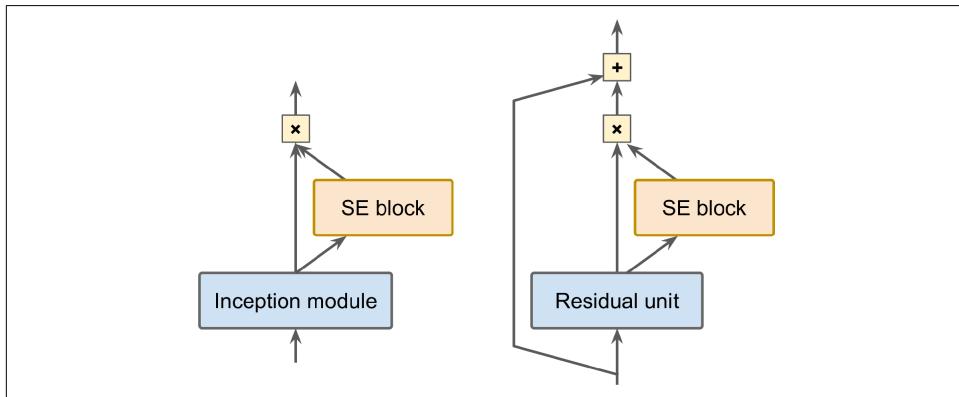


Figure 14-20. SE-Inception module (left) and SE-ResNet unit (right)

²¹ Xingyu Zeng et al., “Crafting GBD-Net for Object Detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, no. 9 (2018): 2109–2123.

²² Jie Hu et al., “Squeeze-and-Excitation Networks,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

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 14-21](#). 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.

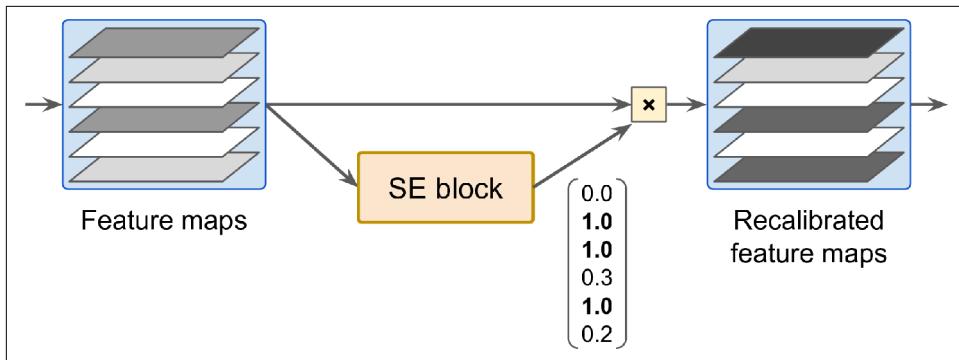


Figure 14-21. An SE block performs feature map recalibration

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 14-22](#)).

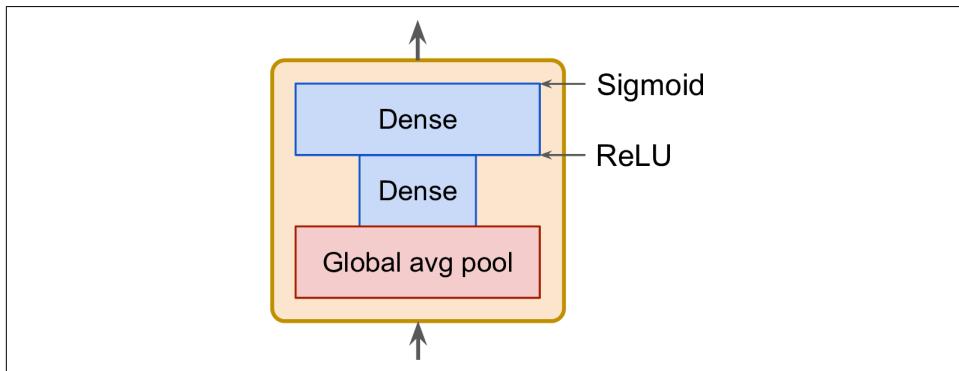


Figure 14-22. SE block architecture

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.

Implementing a ResNet-34 CNN Using Keras

Most CNN architectures described so far are fairly straightforward to implement (although generally you would load a pretrained network instead, as we will see). To illustrate the process, let’s implement a ResNet-34 from scratch using Keras. First, let’s create a `ResidualUnit` layer:

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                keras.layers.Conv2D(filters, 1, strides=strides,
                                   padding="same", use_bias=False),
                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 14-18](#) 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), then we add both outputs and apply the activation function.

Next, we can build the 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):

```
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                            padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(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(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))
```

The only slightly 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. We then 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 fewer than 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 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 `keras.applications` package. For example, you can load the ResNet-50 model, pretrained on ImageNet, with the following line of code:

```
model = keras.applications.resnet50.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 TensorFlow's `tf.image.resize()` function to resize the images we loaded earlier:

```
images_resized = tf.image.resize(images, [224, 224])
```



The `tf.image.resize()` will not preserve the aspect ratio. If this is a problem, try cropping the images to the appropriate aspect ratio before resizing. Both operations can be done in one shot with `tf.image.crop_and_resize()`.

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 -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 pixel values range from 0 to 255, so we must multiply them by 255 (since earlier we scaled them to the 0–1 range):

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Now we can use the pretrained model to make predictions:

```
Y_proba = model.predict(inputs)
```

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,²³ its name, and the corresponding confidence score:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print(" {} - {:.12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

The output looks like this:

```
Image #0
n03877845 - palace      42.87%
n02825657 - bell_cote   40.57%
n03781244 - monastery   14.56%
```

²³ In the ImageNet dataset, each image is associated to a word in the [WordNet dataset](#): the class ID is just a WordNet ID.

```
Image #1
n04522168 - vase           46.83%
n07930864 - cup             7.78%
n11939491 - daisy           4.87%
```

The correct classes (monastery and daisy) appear in the top three results for both images. That's pretty good, considering that the model had to choose from among 1,000 classes.

As you can see, it is very easy to create a pretty good image classifier using a pre-trained model. Other vision models are available in `keras.applications`, including several ResNet variants, GoogLeNet variants like Inception-v3 and Xception, VGGNet variants, and MobileNet and MobileNetV2 (lightweight models for use in mobile applications).

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 to perform transfer learning.

Pretrained Models for Transfer Learning

If you want to build an image classifier but you do not have enough training data, 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, let's load the dataset using TensorFlow Datasets (see [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. The TF Datasets project provides an API for this. For example, let's take the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Next we must preprocess the images. The CNN expects 224×224 images, so we need to resize them. We also need to run the images through Xception's `preprocess_input()` function:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Let's apply this preprocessing function to all three datasets, shuffle the training set, and add batching and prefetching to all the datasets:

```
batch_size = 32
train_set = train_set.shuffle(1000)
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

If you want to perform some data augmentation, change the preprocessing function for the training set, adding some random transformations to the training images. For example, use `tf.image.random_crop()` to randomly crop the images, use `tf.image.random_flip_left_right()` to randomly flip the images horizontally, and so on (see the “Pretrained Models for Transfer Learning” section of the notebook for an example).



The `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, building a `tf.data` pipeline has many advantages: it can read the images efficiently (e.g., in parallel) from any source, not just the local disk; you can manipulate the `Dataset` as you wish; and if you write a preprocessing function based on `tf.image` operations, this function can be used both in the `tf.data` pipeline and in the model you will deploy to production (see [Chapter 19](#)).

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, based on the output of the base model, followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we create the Keras Model:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                    include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = 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 pre-trained layers, at least at the beginning of training:

```
for layer in base_model.layers:  
    layer.trainable = False
```



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 = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)  
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,  
              metrics=["accuracy"])  
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```



This will be very slow, unless you have a GPU. If you do not, then you should run this chapter's notebook in Colab, using a GPU runtime (it's free!). See the instructions at <https://github.com/ageron/handson-ml2>.

After training the model for a few epochs, its validation accuracy should reach about 75–80% and stop making much progress. This means that the top layers are now pretty well trained, so we are ready to unfreeze all the layers (or you could try unfreezing just the top ones) and continue training (don't forget to compile the model when you freeze or unfreeze layers). This time we use a much lower learning rate to avoid damaging the pretrained weights:

```
for layer in base_model.layers:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)  
model.compile(...)  
history = model.fit(...)
```

It will take a while, but this model should reach around 95% accuracy on the test set. With that, you can start training amazing image classifiers! But there's more to computer vision than just classification. For example, what if you also want to know *where* the flower is in the 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 = keras.applications.Xception(weights="imagenet",
                                         include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = 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. If there are just a few thousand images to label, and you don’t plan to do this frequently, it may be preferable to do it yourself. Adriana Kovashka et al. wrote a very practical [paper](#)²⁴ about crowdsourcing in computer vision. I recommend you check it out, even if you do not plan to use crowdsourcing.

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!

²⁴ Adriana Kovashka et al., “Crowdsourcing in Computer Vision,” *Foundations and Trends in Computer Graphics and Vision* 10, no. 3 (2014): 177–243.



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 14-23](#)). In tf.keras, it is implemented by the `tf.keras.metrics.MeanIoU` class.



Figure 14-23. Intersection over Union (IoU) metric for bounding boxes

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

Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object, then slide it across the image, as shown in [Figure 14-24](#). In this example, the image was chopped into a 6×8 grid, and we show a CNN (the thick black rectangle) sliding across all 3×3 regions. When the CNN was looking at the top left of the image, it detected part of the leftmost rose, and then it detected that same rose again when it was first shifted one step to the right. At

the next step, it started detecting part of the topmost rose, and then it detected it again once it was shifted one more step to the right. You would then continue to slide the CNN through the whole image, looking at all 3×3 regions. Moreover, since objects can have varying sizes, you would also slide the CNN across regions of different sizes. For example, once you are done with the 3×3 regions, you might want to slide the CNN across all 4×4 regions as well.

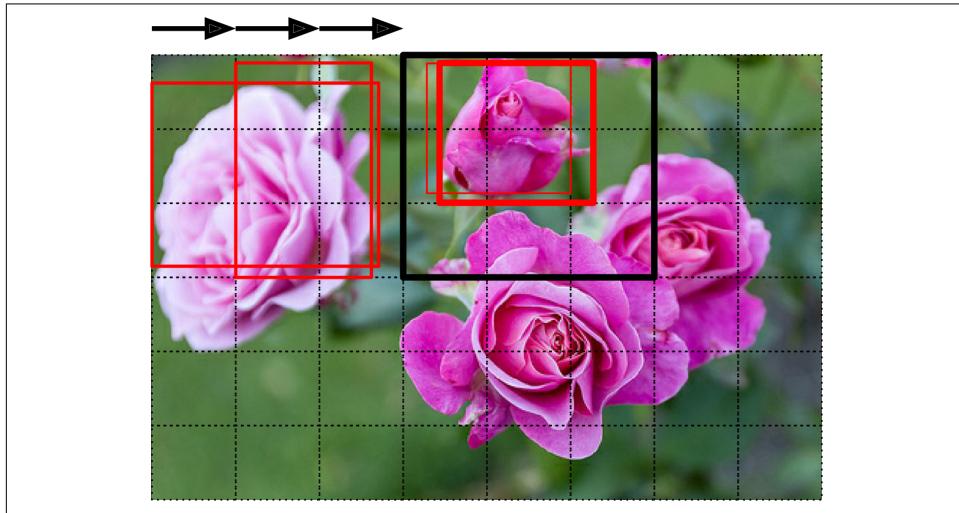


Figure 14-24. Detecting multiple objects by sliding a CNN across the image

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

1. First, you need to add an extra *objectness* output to your CNN, to estimate the probability that a flower is indeed present in the image (alternatively, you could add a “no-flower” class, but this usually does not work as well). It must use the sigmoid activation function, and you can train it using binary cross-entropy loss. Then get rid of all the bounding boxes for which the objectness score is below some threshold: this will drop all the bounding boxes that don't actually contain a flower.
2. Find the bounding box with the highest objectness score, and get rid of all the other bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in Figure 14-24, the bounding box with the max objectness score is the thick bounding box over the topmost rose (the objectness score is represented by the thickness of the bounding boxes). The other bounding box

over that same rose overlaps a lot with the max bounding box, so we will get rid of it.

3. Repeat step two 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, 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 paper²⁵](#) 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 by 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 $100 \times 7 \times 7$ 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 $[batch\ size, 1, 1, 200]$.



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

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size²⁶ (however, it does expect its inputs to have a specific number of channels,

²⁵ Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.

²⁶ There is one small exception: a convolutional layer using "valid" padding will complain if the input size is smaller than the kernel size.

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 logistic activation function, and this gives the objectness score; outputs 6 to 9 do not use any activation function, and they represent the bounding box's center coordinates, as well as its height and width. We can now convert its 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 14-25](#)). If we feed the FCN a 448×448 image (see the right side of [Figure 14-25](#)), the bottleneck layer will now output 14×14 feature maps.²⁷ 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 $8 \times 8 = 64$ 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.

²⁷ This assumes we used only "same" padding in the network; indeed, "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.

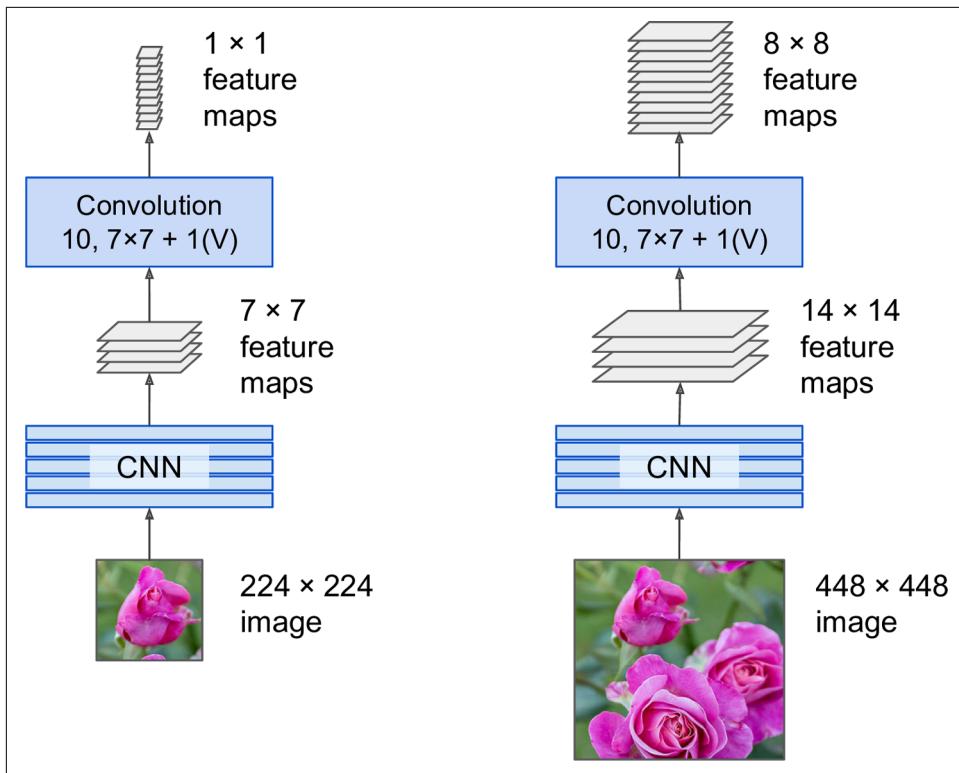


Figure 14-25. The same fully convolutional network processing a small image (left) and a large one (right)

You Only Look Once (YOLO)

YOLO is an extremely fast and accurate object detection architecture proposed by Joseph Redmon et al. in a [2015 paper](#),²⁸ and subsequently improved [in 2016](#)²⁹ (YOLOv2) and [in 2018](#)³⁰ (YOLOv3). It is so fast that it can run in real time on a video, as seen in Redmon's [demo](#).

YOLOv3's architecture is quite similar to the one we just discussed, but with a few important differences:

²⁸ 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.

²⁹ Joseph Redmon and Ali Farhadi, "YOLO9000: Better, Faster, Stronger," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017): 6517–6525.

³⁰ Joseph Redmon and Ali Farhadi, "YOLOv3: An Incremental Improvement," arXiv preprint arXiv:1804.02767 (2018).

- It outputs five bounding boxes for each grid cell (instead of just one), and each bounding box comes with an objectness score. It also outputs 20 class probabilities per grid cell, as it was trained on the PASCAL VOC dataset, which contains 20 classes. That's a total of 45 numbers per grid cell: 5 bounding boxes, each with 4 coordinates, plus 5 objectness scores, plus 20 class probabilities.
- Instead of predicting the absolute coordinates of the bounding box centers, YOLOv3 predicts an offset relative to the coordinates of the grid cell, where $(0, 0)$ means the top left of that cell and $(1, 1)$ means the bottom right. For each grid cell, YOLOv3 is trained to predict only bounding boxes whose center lies in that cell (but the bounding box itself generally extends well beyond the grid cell). YOLOv3 applies the logistic activation function to the bounding box coordinates to ensure they remain in the 0 to 1 range.
- Before training the neural net, YOLOv3 finds five representative bounding box dimensions, called *anchor boxes* (or *bounding box priors*). It does this by applying the K-Means algorithm (see [Chapter 9](#)) to the height and width of the training set bounding boxes. For example, if the training images contain many pedestrians, then one of the anchor boxes will likely have the dimensions of a typical pedestrian. Then when the neural net predicts five bounding boxes per grid cell, it actually predicts how much to rescale each of the anchor boxes. For example, suppose one anchor box is 100 pixels tall and 50 pixels wide, and the network predicts, say, a vertical rescaling factor of 1.5 and a horizontal rescaling of 0.9 (for one of the grid cells). This will result in a predicted bounding box of size 150×45 pixels. To be more precise, for each grid cell and each anchor box, the network predicts the log of the vertical and horizontal rescaling factors. Having these priors makes the network more likely to predict bounding boxes of the appropriate dimensions, and it also speeds up training because it will more quickly learn what reasonable bounding boxes look like.
- The network is trained using images of different scales: every few batches during training, the network randomly chooses a new image dimension (from 330×330 to 608×608 pixels). This allows the network to learn to detect objects at different scales. Moreover, it makes it possible to use YOLOv3 at different scales: the smaller scale will be less accurate but faster than the larger scale, so you can choose the right trade-off for your use case.

There are a few more innovations you might be interested in, such as the use of 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). In the 2016 paper, the authors introduce the YOLO9000 model that uses hierarchical classification: the model predicts a probability for each node in a visual hierarchy called *WordTree*. This makes it possible for the network to predict with high confidence that an image represents, say, a dog, even though it is unsure what specific type of dog. I encourage you

to go ahead and read all three papers: they are quite pleasant to read, and they provide excellent examples of how Deep Learning systems can be incrementally improved.

Mean Average Precision (mAP)

A very common metric used in object detection tasks is the *mean Average Precision* (mAP). “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 3-5](#)). 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 3-5](#)). 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 AP₅₀). 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 AP@[.50:.95] or AP@[.50:0.05:.95]). Yes, that’s a mean mean average.

Several YOLO implementations built using TensorFlow are available on GitHub. In particular, check out [Zihao Zang’s TensorFlow 2 implementation](#). Other object detection models are available in the TensorFlow Models project, many with pretrained

weights; and some have even been ported to TF Hub, such as **SSD**³¹ and **Faster-RCNN**,³² which are both quite popular. SSD is also a “single shot” detection model, similar to YOLO. 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, and a classifier is run for each bounding box, based on the cropped output of the CNN.

The choice of detection system depends on many factors: speed, accuracy, available pretrained models, training time, complexity, etc. The papers contain tables of metrics, but there is quite a lot of variability in the testing environments, and the technologies evolve so fast that it is difficult to make a fair comparison that will be useful for most people and remain valid for more than a few months.

So, we can locate objects by drawing bounding boxes around them. Great! But perhaps you want to be a bit more precise. 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 14-26](#). 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.

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. we discussed earlier. 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 add a single *upsampling layer* that multiplies the resolution by 32.

³¹ Wei Liu et al., “SSD: Single Shot Multibox Detector,” *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.

³² 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.



Figure 14-26. Semantic segmentation

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 $\times 4$ or $\times 8$. Instead, they use a *transposed convolutional layer*:³³ it is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 14-27). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., $1/2$ in Figure 14-27). 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 tf.keras, you can use the Conv2DTranspose layer.

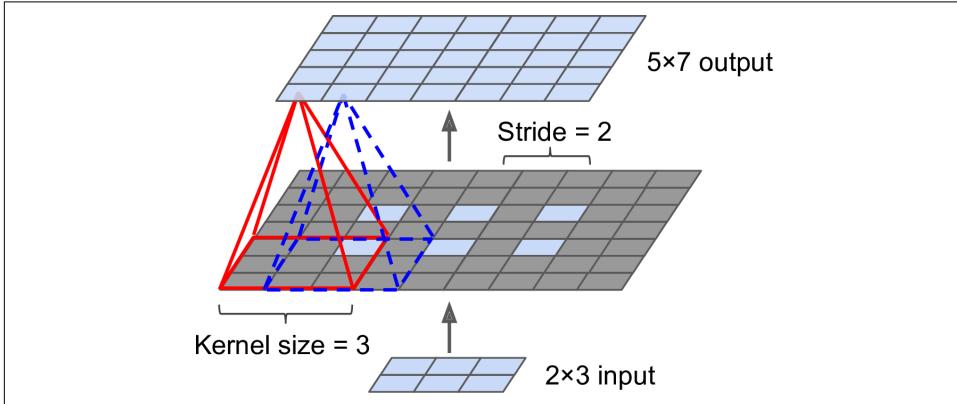


Figure 14-27. Upsampling using a transposed convolutional layer

³³ 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.



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).

TensorFlow Convolution Operations

TensorFlow also offers a few other kinds of convolutional layers:

`keras.layers.Conv1D`

Creates a convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as we will see in [Chapter 15](#).

`keras.layers.Conv3D`

Creates 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.

`tf.nn.depthwise_conv2d()`

Can be used to create a *depthwise convolutional layer* (but you need to create the variables yourself). It applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.

This solution is OK, but still too imprecise. To do better, the authors 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 14-28](#)). 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: upscale $\times 2$, add the output of a lower layer (of the appropriate scale), upscale $\times 2$, add the output of an even lower layer, and finally upscale $\times 8$. 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*.

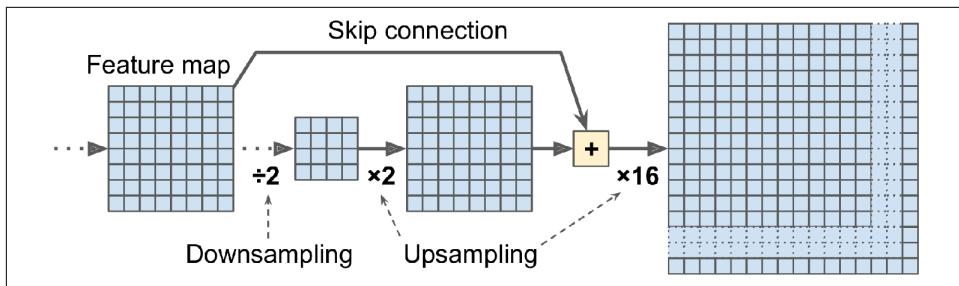


Figure 14-28. Skip layers recover some spatial resolution from lower layers

Once again, many GitHub repositories provide TensorFlow implementations of semantic segmentation (TensorFlow 1 for now), and you will even find pretrained *instance segmentation* models in the TensorFlow Models project. 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). At present, the instance segmentation models available in the TensorFlow Models project are based on the *Mask R-CNN* architecture, which was proposed in a [2017 paper](#):³⁴ it 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.

As you can see, the field of Deep Computer Vision is vast and moving fast, with all sorts of architectures popping out every year, all based on convolutional neural networks. The progress made in just a few years has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts to make the network more resistant to images designed to fool it), explainability (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in [Chapter 17](#)), and *single-shot learning* (a system that can recognize an object after it has seen it just once). Some even explore completely novel architectures, such as Geoffrey Hinton's *capsule networks*³⁵ (I presented them in a couple of [videos](#), with the corresponding code in a notebook). 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.

³⁴ Kaiming He et al., “Mask R-CNN,” arXiv preprint arXiv:1703.06870 (2017).

³⁵ Geoffrey Hinton et al., “Matrix Capsules with EM Routing,” *Proceedings of the International Conference on Learning Representations* (2018).

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.
What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? 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, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, and Xception?
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, including 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](#). It is a fun way to generate art using Deep Learning.

Solutions to these exercises are available in [Appendix A](#).

Processing Sequences Using RNNs and CNNs

The batter hits the ball. The outfielder immediately starts running, anticipating the ball's trajectory. He tracks it, adapts his movements, and finally catches it (under a thunder of applause). Predicting the future is something you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs), a class of nets that can predict the future (well, up to a point, of course). They can analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have considered so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter we will first look at the fundamental concepts underlying RNNs and how to train them using backpropagation through time, then we will use them to forecast a time series. After that we'll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in [Chapter 11](#)), which can be alleviated using various techniques, including recurrent dropout and recurrent layer normalization
- A (very) limited short-term memory, which can be extended using LSTM and GRU cells

RNNs are not the only types of neural networks capable of handling sequential data: for small sequences, a regular dense network can do the trick; and for very long sequences, such as audio samples or text, convolutional neural networks can actually

work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a *WaveNet*: this is a CNN architecture capable of handling sequences of tens of thousands of time steps. In [Chapter 16](#), we will continue to explore RNNs and see how to use them for natural language processing, along with more recent architectures based on attention mechanisms. Let's get started!

Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer (a few exceptions are discussed in [Appendix E](#)). A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward. Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in [Figure 15-1](#) (left). At each *time step* t (also called a *frame*), this *recurrent neuron* receives the inputs $\mathbf{x}_{(t)}$ as well as its own output from the previous time step, $y_{(t-1)}$. Since there is no previous output at the first time step, it is generally set to 0. We can represent this tiny network against the time axis, as shown in [Figure 15-1](#) (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

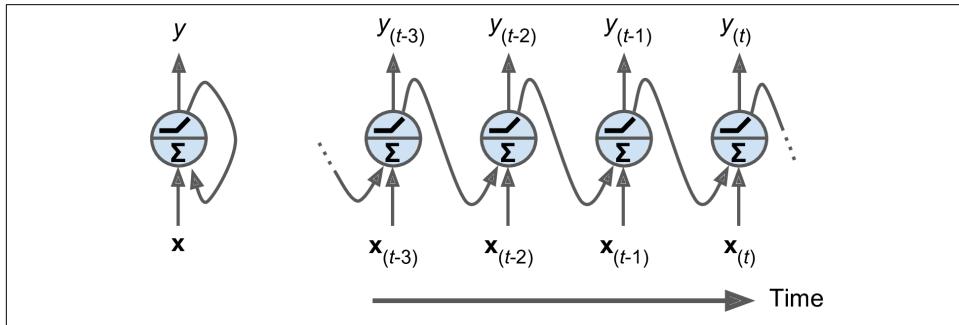


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

You can easily create a layer of recurrent neurons. At each time step t , every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step $\mathbf{y}_{(t-1)}$, as shown in [Figure 15-2](#). Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).

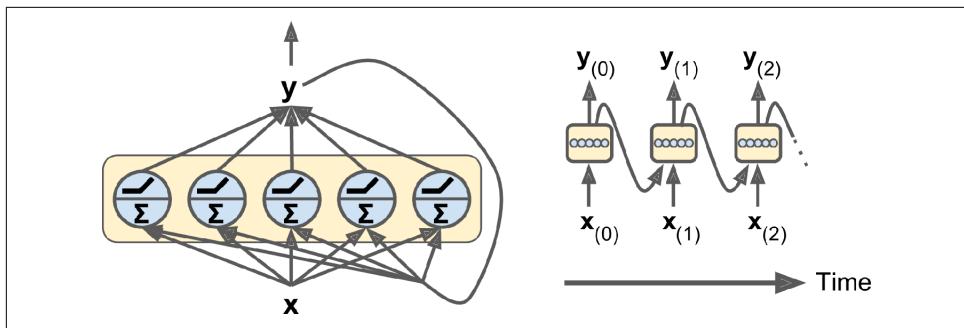


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\mathbf{y}_{(t-1)}$. Let's call these weight vectors \mathbf{w}_x and \mathbf{w}_y . If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, \mathbf{W}_x and \mathbf{W}_y . The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in [Equation 15-1](#) (\mathbf{b} is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU¹).

Equation 15-1. Output of a recurrent layer for a single instance

$$\mathbf{y}_{(t)} = \phi(\mathbf{W}_x^\top \mathbf{x}_{(t)} + \mathbf{W}_y^\top \mathbf{y}_{(t-1)} + \mathbf{b})$$

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for a whole mini-batch by placing all the inputs at time step t in an input matrix $\mathbf{X}_{(t)}$ (see [Equation 15-2](#)).

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\begin{aligned} \mathbf{Y}_{(t)} &= \phi(\mathbf{X}_{(t)} \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}) \\ &= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix} \end{aligned}$$

¹ Note that many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than the ReLU activation function. For example, take a look at Vu Pham et al.'s 2013 paper "[Dropout Improves Recurrent Neural Networks for Handwriting Recognition](#)". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s 2015 paper "[A Simple Way to Initialize Recurrent Networks of Rectified Linear Units](#)".

In this equation:

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- \mathbf{W}_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- \mathbf{W}_y is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- \mathbf{b} is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices \mathbf{W}_x and \mathbf{W}_y are often concatenated vertically into a single weight matrix \mathbf{W} of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of [Equation 15-2](#)).
- The notation $[\mathbf{X}_{(t)} \mathbf{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$.

Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on. This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

In general a cell's state at time step t , denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Its output at time step t , denoted $\mathbf{y}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is simply equal to the state, but in more complex cells this is not always the case, as shown in [Figure 15-3](#).

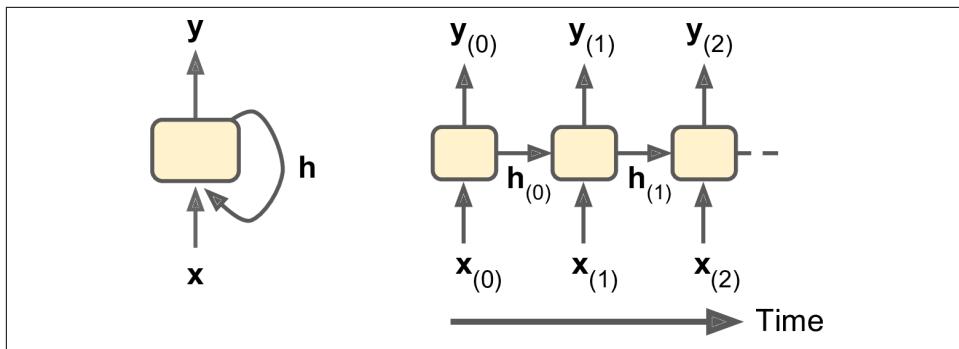


Figure 15-3. A cell's hidden state and its output may be different

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in [Figure 15-4](#)). This type of *sequence-to-sequence network* is useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from $N - 1$ days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in [Figure 15-4](#)). In other words, this is a *sequence-to-vector network*. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from -1 [hate] to $+1$ [love]).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of [Figure 15-4](#)). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of [Figure 15-4](#)). For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an *Encoder–Decoder*, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it. We will see how to implement an Encoder–Decoder in [Chapter 16](#) (as we will see, it is a bit more complex than in [Figure 15-4](#) suggests).

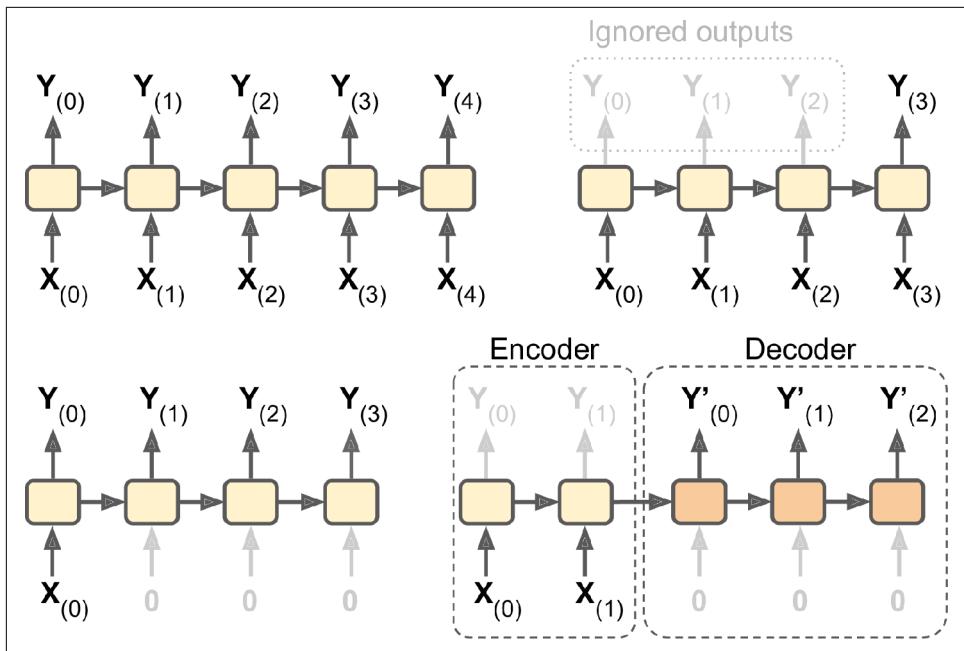


Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

Sounds promising, but how do you train a recurrent neural network?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then simply use regular backpropagation (see [Figure 15-5](#)). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a cost function $C(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \dots, \mathbf{Y}_{(T)})$ (where T is the max time step). Note that this cost function may ignore some outputs, as shown in [Figure 15-5](#) (for example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one). The gradients of that cost function are then propagated backward through the unrolled network (represented by the solid arrows). Finally the model parameters are updated using the gradients computed during BPTT. Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output (for example, in [Figure 15-5](#) the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs,

but not through $\mathbf{Y}_{(0)}$ and $\mathbf{Y}_{(1)}$). Moreover, since the same parameters \mathbf{W} and \mathbf{b} are used at each time step, backpropagation will do the right thing and sum over all time steps.

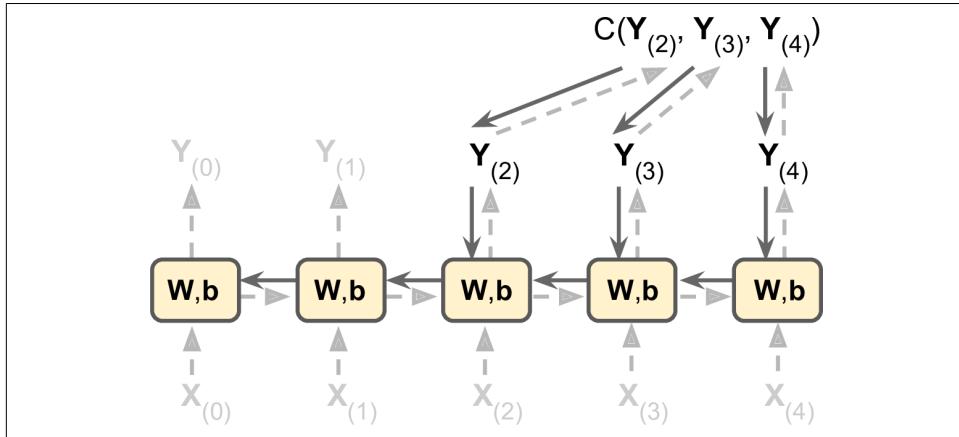


Figure 15-5. Backpropagation through time

Fortunately, tf.keras takes care of all of this complexity for you—so let’s start coding!

Forecasting a Time Series

Suppose you are studying the number of active users per hour on your website, or the daily temperature in your city, or your company’s financial health, measured quarterly using multiple metrics. In all these cases, the data will be a sequence of one or more values per time step. This is called a *time series*. In the first two examples there is a single value per time step, so these are *univariate time series*, while in the financial example there are multiple values per time step (e.g., the company’s revenue, debt, and so on), so it is a *multivariate time series*. A typical task is to predict future values, which is called *forecasting*. Another common task is to fill in the blanks: to predict (or rather “postdict”) missing values from the past. This is called *imputation*. For example, Figure 15-6 shows 3 univariate time series, each of them 50 time steps long, and the goal here is to forecast the value at the next time step (represented by the X) for each of them.

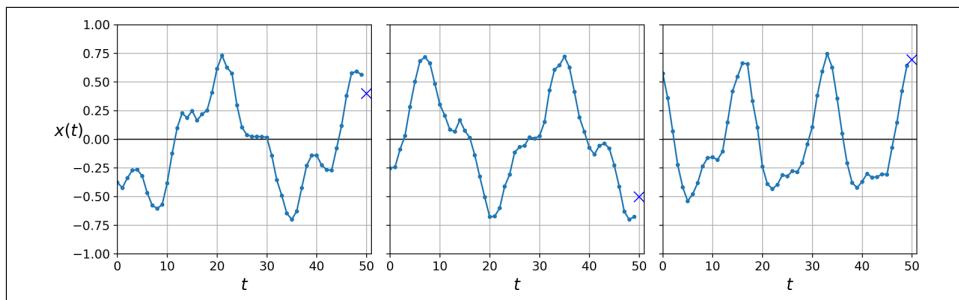


Figure 15-6. Time series forecasting

For simplicity, we are using a time series generated by the `generate_time_series()` function, shown here:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) # wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) # + noise
    return series[..., np.newaxis].astype(np.float32)
```

This function creates as many time series as requested (via the `batch_size` argument), each of length `n_steps`, and there is just one value per time step in each series (i.e., all series are univariate). The function returns a NumPy array of shape `[batch size, time steps, 1]`, where each series is the sum of two sine waves of fixed amplitudes but random frequencies and phases, plus a bit of noise.



When dealing with time series (and other types of sequences such as sentences), the input features are generally represented as 3D arrays of shape `[batch size, time steps, dimensionality]`, where `dimensionality` is 1 for univariate time series and more for multivariate time series.

Now let's create a training set, a validation set, and a test set using this function:

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

`X_train` contains 7,000 time series (i.e., its shape is `[7000, 50, 1]`), while `X_valid` contains 2,000 (from the 7,000th time series to the 8,999th) and `X_test` contains 1,000 (from the 9,000th to the 9,999th). Since we want to forecast a single value for each series, the targets are column vectors (e.g., `y_train` has a shape of `[7000, 1]`).

Baseline Metrics

Before we start using RNNs, it is often a good idea to have a few baseline metrics, or else we may end up thinking our model works great when in fact it is doing worse than basic models. For example, the simplest approach is to predict the last value in each series. This is called *naive forecasting*, and it is sometimes surprisingly difficult to outperform. In this case, it gives us a mean squared error of about 0.020:

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Another simple approach is to use a fully connected network. Since it expects a flat list of features for each input, we need to add a `Flatten` layer. Let's just use a simple Linear Regression model so that each prediction will be a linear combination of the values in the time series:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

If we compile this model using the MSE loss and the default Adam optimizer, then fit it on the training set for 20 epochs and evaluate it on the validation set, we get an MSE of about 0.004. That's much better than the naive approach!

Implementing a Simple RNN

Let's see if we can beat that with a simple RNN:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

That's really the simplest RNN you can build. It just contains a single layer, with a single neuron, as we saw in [Figure 15-1](#). We do not need to specify the length of the input sequences (unlike in the previous model), since a recurrent neural network can process any number of time steps (this is why we set the first input dimension to `None`). By default, the `SimpleRNN` layer uses the hyperbolic tangent activation function. It works exactly as we saw earlier: the initial state $h_{(init)}$ is set to 0, and it is passed to a single recurrent neuron, along with the value of the first time step, $x_{(0)}$. The neuron computes a weighted sum of these values and applies the hyperbolic tangent activation function to the result, and this gives the first output, y_0 . In a simple RNN, this output is also the new state h_0 . This new state is passed to the same recurrent neuron along with the next input value, $x_{(1)}$, and the process is repeated until the last time step. Then the layer just outputs the last value, y_{49} . All of this is performed simultaneously for every time series.



By default, recurrent layers in Keras only return the final output. To make them return one output per time step, you must set `return_sequences=True`, as we will see.

If you compile, fit, and evaluate this model (just like earlier, we train for 20 epochs using Adam), you will find that its MSE reaches only 0.014, so it is better than the naive approach but it does not beat a simple linear model. Note that for each neuron, a linear model has one parameter per input and per time step, plus a bias term (in the simple linear model we used, that's a total of 51 parameters). In contrast, for each recurrent neuron in a simple RNN, there is just one parameter per input and per hidden state dimension (in a simple RNN, that's just the number of recurrent neurons in the layer), plus a bias term. In this simple RNN, that's a total of just three parameters.

Trend and Seasonality

There are many other models to forecast time series, such as *weighted moving average* models or *autoregressive integrated moving average* (ARIMA) models. Some of them require you to first remove the trend and seasonality. For example, if you are studying the number of active users on your website, and it is growing by 10% every month, you would have to remove this trend from the time series. Once the model is trained and starts making predictions, you would have to add the trend back to get the final predictions. Similarly, if you are trying to predict the amount of sunscreen lotion sold every month, you will probably observe strong seasonality: since it sells well every summer, a similar pattern will be repeated every year. You would have to remove this seasonality from the time series, for example by computing the difference between the value at each time step and the value one year earlier (this technique is called *differencing*). Again, after the model is trained and makes predictions, you would have to add the seasonal pattern back to get the final predictions.

When using RNNs, it is generally not necessary to do all this, but it may improve performance in some cases, since the model will not have to learn the trend or the seasonality.

Apparently our simple RNN was too simple to get good performance. So let's try to add more recurrent layers!

Deep RNNs

It is quite common to stack multiple layers of cells, as shown in [Figure 15-7](#). This gives you a *deep RNN*.

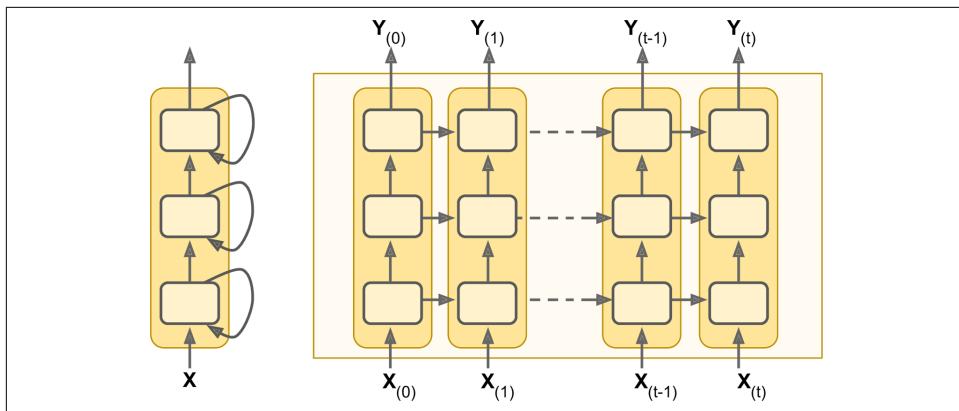


Figure 15-7. Deep RNN (left) unrolled through time (right)

Implementing a deep RNN with `tf.keras` is quite simple: just stack recurrent layers. In this example, we use three `SimpleRNN` layers (but we could add any other type of recurrent layer, such as an `LSTM` layer or a `GRU` layer, which we will discuss shortly):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```



Make sure to set `return_sequences=True` for all recurrent layers (except the last one, if you only care about the last output). If you don't, they will output a 2D array (containing only the output of the last time step) instead of a 3D array (containing outputs for all time steps), and the next recurrent layer will complain that you are not feeding it sequences in the expected 3D format.

If you compile, fit, and evaluate this model, you will find that it reaches an MSE of 0.003. We finally managed to beat the linear model!

Note that the last layer is not ideal: it must have a single unit because we want to forecast a univariate time series, and this means we must have a single output value per time step. However, having a single unit means that the hidden state is just a single number. That's really not much, and it's probably not that useful; presumably, the RNN will mostly use the hidden states of the other recurrent layers to carry over all the information it needs from time step to time step, and it will not use the final layer's hidden state very much. Moreover, since a `SimpleRNN` layer uses the `tanh` activation function by default, the predicted values must lie within the range -1 to 1 . But what if you want to use another activation function? For both these reasons, it might be preferable to replace the output layer with a `Dense` layer: it would run slightly

faster, the accuracy would be roughly the same, and it would allow us to choose any output activation function we want. If you make this change, also make sure to remove `return_sequences=True` from the second (now last) recurrent layer:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

If you train this model, you will see that it converges faster and performs just as well. Plus, you could change the output activation function if you wanted.

Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict 10 steps ahead, just change the targets to be the value 10 steps ahead instead of 1 step ahead). But what if we want to predict the next 10 values?

The first option is to use the model we already trained, make it predict the next value, then add that value to the inputs (acting as if this predicted value had actually occurred), and use the model again to predict the following value, and so on, as in the following code:

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[..., np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

As you might expect, the prediction for the next step will usually be more accurate than the predictions for later time steps, since the errors might accumulate (as you can see in [Figure 15-8](#)). If you evaluate this approach on the validation set, you will find an MSE of about 0.029. This is much higher than the previous models, but it's also a much harder task, so the comparison doesn't mean much. It's much more meaningful to compare this performance with naive predictions (just forecasting that the time series will remain constant for 10 time steps) or with a simple linear model. The naive approach is terrible (it gives an MSE of about 0.223), but the linear model gives an MSE of about 0.0188: it's much better than using our RNN to forecast the future one step at a time, and also much faster to train and run. Still, if you only want to forecast a few time steps ahead, on more complex tasks, this approach may work well.

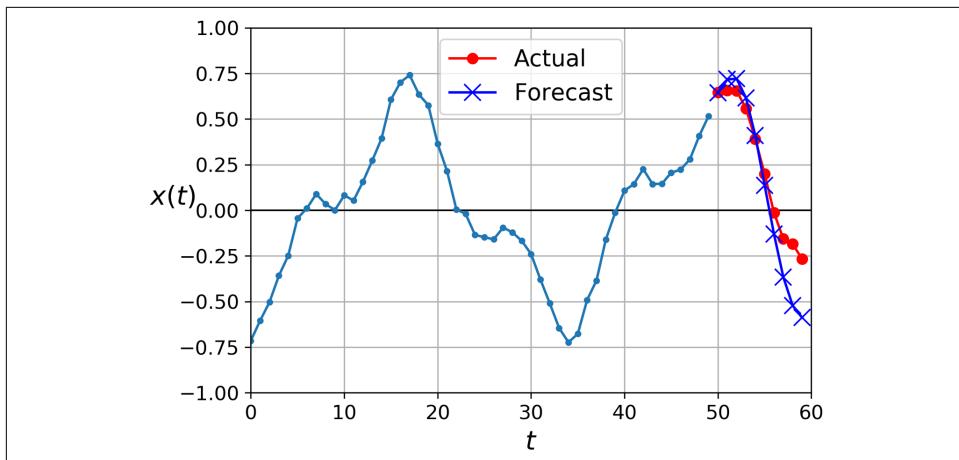


Figure 15-8. Forecasting 10 steps ahead, 1 step at a time

The second option is to train an RNN to predict all 10 next values at once. We can still use a sequence-to-vector model, but it will output 10 values instead of 1. However, we first need to change the targets to be vectors containing the next 10 values:

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Now we just need the output layer to have 10 units instead of 1:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

After training this model, you can predict the next 10 values at once very easily:

```
Y_pred = model.predict(X_new)
```

This model works nicely: the MSE for the next 10 time steps is about 0.008. That's much better than the linear model. But we can still do better: indeed, instead of training the model to forecast the next 10 values only at the very last time step, we can train it to forecast the next 10 values at each and every time step. In other words, we can turn this sequence-to-vector RNN into a sequence-to-sequence RNN. The advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just the output at the last time step. This means there will be many more error gradients flowing through the model, and they won't have to flow only through time; they will also flow from the output of each time step. This will both stabilize and speed up training.

To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to 10, then at time step 1 the model will forecast time steps 2 to 11, and so on. So each target must be a sequence of the same length as the input sequence, containing a 10-dimensional vector at each step. Let's prepare these target sequences:

```
Y = np.empty((10000, n_steps, 10)) # each target is a sequence of 10D vectors
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```



It may be surprising that the targets will contain values that appear in the inputs (there is a lot of overlap between `X_train` and `Y_train`). Isn't that cheating? Fortunately, not at all: at each time step, the model only knows about past time steps, so it cannot look ahead. It is said to be a *causal* model.

To turn the model into a sequence-to-sequence model, we must set `return_sequences=True` in all recurrent layers (even the last one), and we must apply the output `Dense` layer at every time step. Keras offers a `TimeDistributed` layer for this very purpose: it wraps any layer (e.g., a `Dense` layer) and applies it at every time step of its input sequence. It does this efficiently, by reshaping the inputs so that each time step is treated as a separate instance (i.e., it reshapes the inputs from `[batch size, time steps, input dimensions]` to `[batch size × time steps, input dimensions]`; in this example, the number of input dimensions is 20 because the previous `SimpleRNN` layer has 20 units), then it runs the `Dense` layer, and finally it reshapes the outputs back to sequences (i.e., it reshapes the outputs from `[batch size × time steps, output dimensions]` to `[batch size, time steps, output dimensions]`; in this example the number of output dimensions is 10, since the `Dense` layer has 10 units).² Here is the updated model:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

The `Dense` layer actually supports sequences as inputs (and even higher-dimensional inputs): it handles them just like `TimeDistributed(Dense(...))`, meaning it is applied to the last input dimension only (independently across all time steps). Thus, we could replace the last layer with just `Dense(10)`. For the sake of clarity, however, we will keep using `TimeDistributed(Dense(10))` because it makes it clear that the `Dense`

² Note that a `TimeDistributed(Dense(n))` layer is equivalent to a `Conv1D(n, filter_size=1)` layer.

layer is applied independently at each time step and that the model will output a sequence, not just a single vector.

All outputs are needed during training, but only the output at the last time step is useful for predictions and for evaluation. So although we will rely on the MSE over all the outputs for training, we will use a custom metric for evaluation, to only compute the MSE over the output at the last time step:

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

We get a validation MSE of about 0.006, which is 25% better than the previous model. You can combine this approach with the first one: just predict the next 10 values using this RNN, then concatenate these values to the input time series and use the model again to predict the next 10 values, and repeat the process as many times as needed. With this approach, you can generate arbitrarily long sequences. It may not be very accurate for long-term predictions, but it may be just fine if your goal is to generate original music or text, as we will see in [Chapter 16](#).



When forecasting time series, it is often useful to have some error bars along with your predictions. For this, an efficient technique is MC Dropout, introduced in [Chapter 11](#): add an MC Dropout layer within each memory cell, dropping part of the inputs and hidden states. After training, to forecast a new time series, use the model many times and compute the mean and standard deviation of the predictions at each time step.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

Handling Long Sequences

To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem, discussed in [Chapter 11](#): it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.

Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here; in fact, they may actually lead the RNN to be even more unstable during training. Why? Well, suppose Gradient Descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that. You can reduce this risk by using a smaller learning rate, but you can also simply use a saturating activation function like the hyperbolic tangent (this explains why it is the default). In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients (e.g., using TensorBoard) and perhaps use Gradient Clipping.

Moreover, Batch Normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a BN layer to a memory cell (as we will see shortly) so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results, as was demonstrated by César Laurent et al. in a [2015 paper](#):³ the authors found that BN was slightly beneficial only when it was applied to the inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in [Figure 15-7](#)), but not within recurrent layers (i.e., horizontally). In Keras this can be done simply by adding a `Batch Normalization` layer before each recurrent layer, but don't expect too much from it.

Another form of normalization often works better with RNNs: *Layer Normalization*. This idea was introduced by Jimmy Lei Ba et al. in a [2016 paper](#):⁴ it is very similar to Batch Normalization, but instead of normalizing across the batch dimension, it normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set. Like BN, Layer Normalization learns a

³ César Laurent et al., “Batch Normalized Recurrent Neural Networks,” *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.

⁴ Jimmy Lei Ba et al., “Layer Normalization,” arXiv preprint arXiv:1607.06450 (2016).

scale and an offset parameter for each input. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

Let's use tf.keras to implement Layer Normalization within a simple memory cell. For this, we need to define a custom memory cell. It is just like a regular layer, except its `call()` method takes two arguments: the `inputs` at the current time step and the `hidden states` from the previous time step. Note that the `states` argument is a list containing one or more tensors. In the case of a simple RNN cell it contains a single tensor equal to the outputs of the previous time step, but other cells may have multiple state tensors (e.g., an `LSTMCell` has a long-term state and a short-term state, as we will see shortly). A cell must also have a `state_size` attribute and an `output_size` attribute. In a simple RNN, both are simply equal to the number of units. The following code implements a custom memory cell which will behave like a `SimpleRNNCell`, except it will also apply Layer Normalization at each time step:

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

The code is quite straightforward.⁵ Our `LNSimpleRNNCell` class inherits from the `keras.layers.Layer` class, just like any custom layer. The constructor takes the number of units and the desired activation function, and it sets the `state_size` and `output_size` attributes, then creates a `SimpleRNNCell` with no activation function (because we want to perform Layer Normalization after the linear operation but before the activation function). Then the constructor creates the `LayerNormalization` layer, and finally it fetches the desired activation function. The `call()` method starts by applying the simple RNN cell, which computes a linear combination of the current inputs and the previous hidden states, and it returns the result twice (indeed, in a `SimpleRNNCell`, the outputs are just equal to the hidden states: in other words, `new_states[0]` is equal to `outputs`, so we can safely ignore `new_states` in the rest of the `call()` method). Next, the `call()` method applies Layer Normalization, followed

⁵ It would have been simpler to inherit from `SimpleRNNCell` instead so that we wouldn't have to create an internal `SimpleRNNCell` or handle the `state_size` and `output_size` attributes, but the goal here was to show how to create a custom cell from scratch.

by the activation function. Finally, it returns the outputs twice (once as the outputs, and once as the new hidden states). To use this custom cell, all we need to do is create a `keras.layers.RNN` layer, passing it a cell instance:

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Similarly, you could create a custom cell to apply dropout between each time step. But there's a simpler way: all recurrent layers (except for `keras.layers.RNN`) and all cells provided by Keras have a `dropout` hyperparameter and a `recurrent_dropout` hyperparameter: the former defines the dropout rate to apply to the inputs (at each time step), and the latter defines the dropout rate for the hidden states (also at each time step). No need to create a custom cell to apply dropout at each time step in an RNN.

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.

Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish⁶ trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

LSTM cells

The *Long Short-Term Memory* (LSTM) cell was proposed in 1997⁷ by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,⁸ and Wojciech Zaremba.⁹ If you consider the

⁶ A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.

⁷ Sepp Hochreiter and Jürgen Schmidhuber, “Long Short-Term Memory,” *Neural Computation* 9, no. 8 (1997): 1735–1780.

⁸ Haşim Sak et al., “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition,” arXiv preprint arXiv:1402.1128 (2014).

⁹ Wojciech Zaremba et al., “Recurrent Neural Network Regularization,” arXiv preprint arXiv:1409.2329 (2014).

LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect long-term dependencies in the data. In Keras, you can simply use the `LSTM` layer instead of the `SimpleRNN` layer:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Alternatively, you could use the general-purpose `keras.layers.RNN` layer, giving it an `LSTMCell` as an argument:

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                    input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

However, the `LSTM` layer uses an optimized implementation when running on a GPU (see [Chapter 19](#)), so in general it is preferable to use it (the `RNN` layer is mostly useful when you define custom cells, as we did earlier).

So how does an LSTM cell work? Its architecture is shown in [Figure 15-9](#).

If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell"). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

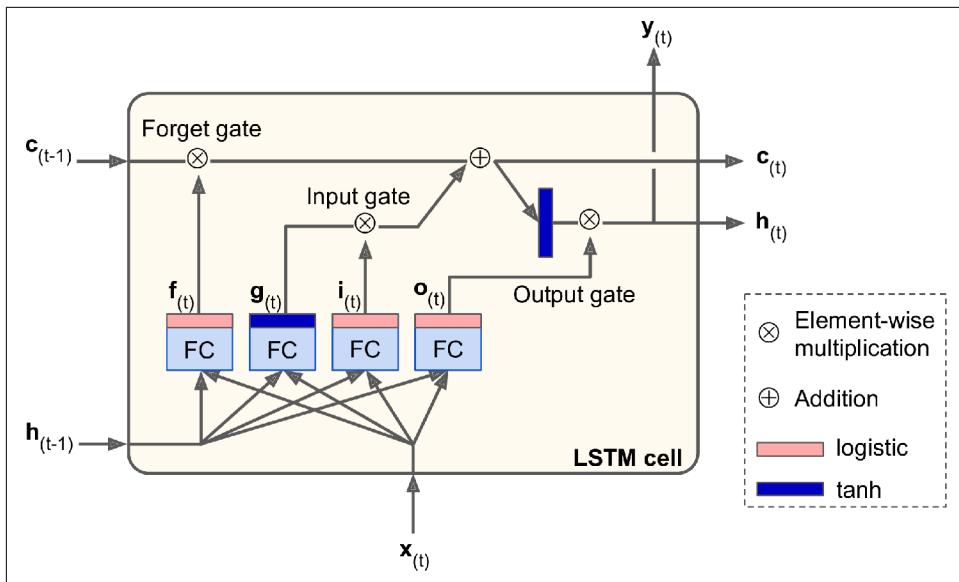


Figure 15-9. LSTM cell

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $c_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $c_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $h_{(t)}$ (which is equal to the cell's output for this time step, $y_{(t)}$). Now let's look at where new memories come from and how the gates work.

First, the current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $g_{(t)}$. It has the usual role of analyzing the current inputs $x_{(t)}$ and the previous (short-term) state $h_{(t-1)}$. In a basic cell, there is nothing other than this layer, and its output goes straight out to $y_{(t)}$ and $h_{(t)}$. In contrast, in an LSTM cell this layer's output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, their outputs range from 0 to 1. As you can see, their outputs are fed to

element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it. Specifically:

- The *forget gate* (controlled by $f_{(t)}$) controls which parts of the long-term state should be erased.
- The *input gate* (controlled by $i_{(t)}$) controls which parts of $g_{(t)}$ should be added to the long-term state.
- Finally, the *output gate* (controlled by $o_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $h_{(t)}$ and to $y_{(t)}$.

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

[Equation 15-3](#) summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

Equation 15-3. LSTM computations

$$\begin{aligned} i_{(t)} &= \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i) \\ f_{(t)} &= \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f) \\ o_{(t)} &= \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o) \\ g_{(t)} &= \tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g) \\ c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\ y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \end{aligned}$$

In this equation:

- W_{xi} , W_{xf} , W_{xo} , W_{xg} are the weight matrices of each of the four layers for their connection to the input vector $x_{(t)}$.
- W_{hi} , W_{hf} , W_{ho} , and W_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state $h_{(t-1)}$.
- b_i , b_f , b_o , and b_g are the bias terms for each of the four layers. Note that TensorFlow initializes b_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Peephole connections

In a regular LSTM cell, the gate controllers can look only at the input $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$. It may be a good idea to give them a bit more context by letting them peek at the long-term state as well. This idea was proposed by [Felix Gers and Jürgen Schmidhuber in 2000](#).¹⁰ They proposed an LSTM variant with extra connections called *peephole connections*: the previous long-term state $\mathbf{c}_{(t-1)}$ is added as an input to the controllers of the forget gate and the input gate, and the current long-term state $\mathbf{c}_{(t)}$ is added as input to the controller of the output gate. This often improves performance, but not always, and there is no clear pattern for which tasks are better off with or without them: you will have to try it on your task and see if it helps.

In Keras, the LSTM layer is based on the `keras.layers.LSTMCell` cell, which does not support peepholes. The experimental `tf.keras.experimental.PeepholeLSTMCell` does, however, so you can create a `keras.layers.RNN` layer and pass a `PeepholeLSTMCell` to its constructor.

There are many other variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

GRU cells

The *Gated Recurrent Unit* (GRU) cell (see [Figure 15-10](#)) was proposed by Kyunghyun Cho et al. in a [2014 paper](#)¹¹ that also introduced the Encoder–Decoder network we discussed earlier.

¹⁰ F. A. Gers and J. Schmidhuber, “Recurrent Nets That Time and Count,” *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000): 189–194.

¹¹ Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation,” *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.

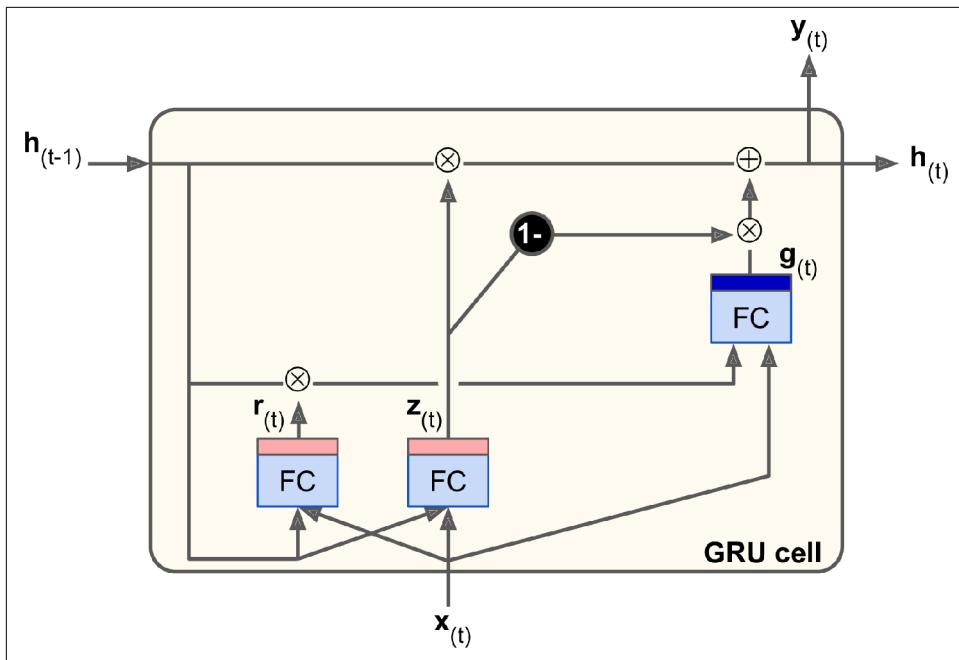


Figure 15-10. GRU cell

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well¹² (which explains its growing popularity). These are the main simplifications:

- Both state vectors are merged into a single vector $h_{(t)}$.
- A single gate controller $z_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open ($= 1$) and the input gate is closed ($1 - 1 = 0$). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $r_{(t)}$ that controls which part of the previous state will be shown to the main layer ($g_{(t)}$).

¹² A 2015 paper by Klaus Greff et al., “LSTM: A Search Space Odyssey”, seems to show that all LSTM variants perform roughly the same.

Equation 15-4 summarizes how to compute the cell's state at each time step for a single instance.

Equation 15-4. GRU computations

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\ \mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}\end{aligned}$$

Keras provides a `keras.layers.GRU` layer (based on the `keras.layers.GRUCell` memory cell); using it is just a matter of replacing `SimpleRNN` or `LSTM` with `GRU`.

LSTM and GRU cells are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences. One way to solve this is to shorten the input sequences, for example using 1D convolutional layers.

Using 1D convolutional layers to process sequences

In [Chapter 14](#), we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel). Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size). If you use 10 kernels, then the layer's output will be composed of 10 1-dimensional sequences (all of the same length), or equivalently you can view this output as a single 10-dimensional sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). If you use a 1D convolutional layer with a stride of 1 and "same" padding, then the output sequence will have the same length as the input sequence. But if you use "valid" padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly. For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that downsamples the input sequence by a factor of 2, using a stride of 2. The kernel size is larger than the stride, so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details. By shortening the sequences, the convolutional layer may help the GRU layers detect longer patterns. Note that we must also crop off the first three

time steps in the targets (since the kernel's size is 4, the first output of the convolutional layer will be based on the input time steps 0 to 3), and downsample the targets by a factor of 2:

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, ::2], epochs=20,
                      validation_data=(X_valid, Y_valid[:, ::2]))
```

If you train and evaluate this model, you will find that it is the best model so far. The convolutional layer really helps. In fact, it is actually possible to use only 1D convolutional layers and drop the recurrent layers entirely!

WaveNet

In a [2016 paper](#),¹³ Aaron van den Oord and other DeepMind researchers introduced an architecture called *WaveNet*. They stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time steps, and so on (see [Figure 15-11](#)). This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently.

¹³ Aaron van den Oord et al., “WaveNet: A Generative Model for Raw Audio,” arXiv preprint arXiv:1609.03499 (2016).

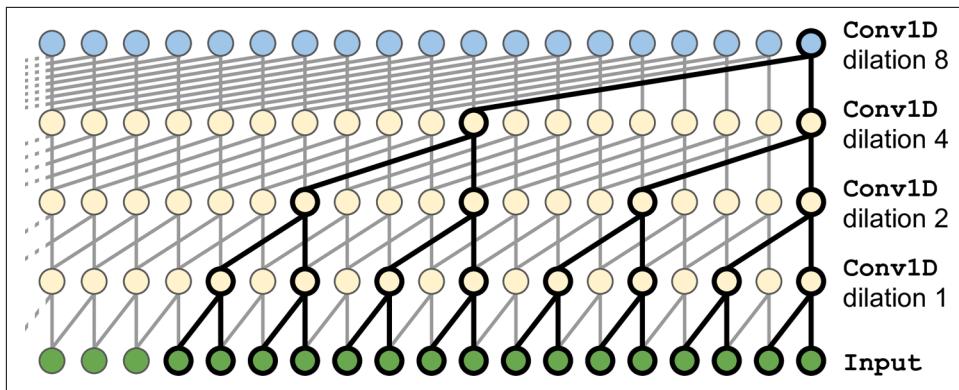


Figure 15-11. WaveNet architecture

In the WaveNet paper, the authors actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, ..., 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, ..., 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters), which is why they stacked 3 such blocks. They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network. Here is how to implement a simplified WaveNet to tackle the same sequences as earlier:¹⁴

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                 activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                     validation_data=(X_valid, Y_valid))
```

This Sequential model starts with an explicit input layer (this is simpler than trying to set `input_shape` only on the first layer), then continues with a 1D convolutional layer using "causal" padding: this ensures that the convolutional layer does not peek into the future when making predictions (it is equivalent to padding the inputs with the right amount of zeros on the left and using "valid" padding). We then add

¹⁴ The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *Gated Activation Units* similar to those found in a GRU cell. Please see the notebook for more details.

similar pairs of layers using growing dilation rates: 1, 2, 4, 8, and again 1, 2, 4, 8. Finally, we add the output layer: a convolutional layer with 10 filters of size 1 and without any activation function. Thanks to the padding layers, every convolutional layer outputs a sequence of the same length as the input sequences, so the targets we use during training can be the full sequences: no need to crop them or downsample them.

The last two models offer the best performance so far in forecasting our time series! In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing incredibly realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.

In [Chapter 16](#), we will continue to explore RNNs, and we will see how they can tackle various NLP tasks.

Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`? What about a sequence-to-vector RNN?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell’s architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Train a classification model for the SketchRNN dataset, available in TensorFlow Datasets.
10. Download the [Bach chorales](#) dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note’s index on a piano (except for the value 0, which means that no note is played). Train a model—recurrent, convolutional, or both—that can predict the next time step (four notes), given a sequence of time steps from a chorale. Then use this

model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on. Also make sure to check out [Google's Coconet model](#), which was used for a nice Google doodle about Bach.

Solutions to these exercises are available in [Appendix A](#).

Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous [Turing test](#)¹ in 1950, his objective was to evaluate a machine's ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.² This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords; it could pretend that it is joking or drunk, to get a pass on its weirdest answers; or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability. Can we build a machine that can read and write natural language?

A common approach for natural language tasks is to use recurrent neural networks. We will therefore continue to explore RNNs (introduced in [Chapter 15](#)), starting with a *character RNN*, trained to predict the next character in a sentence. This will allow us to generate some original text, and in the process we will see how to build a TensorFlow Dataset on a very long sequence. We will first use a *stateless RNN* (which learns

¹ Alan Turing, "Computing Machinery and Intelligence," *Mind* 49 (1950): 433–460.

² Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

on random portions of text at each iteration, without any information on the rest of the text), then we will build a *stateful RNN* (which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns). Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater’s feeling about the movie), this time treating sentences as sequences of words, rather than characters. Then we will show how RNNs can be used to build an Encoder–Decoder architecture capable of performing neural machine translation (NMT). For this, we will use the seq2seq API provided by the TensorFlow Addons project.

In the second part of this chapter, we will look at *attention mechanisms*. As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step. First we will see how to boost the performance of an RNN-based Encoder–Decoder architecture using attention, then we will drop RNNs altogether and look at a very successful attention-only architecture called the *Transformer*. Finally, we will take a look at some of the most important advances in NLP in 2018 and 2019, including incredibly powerful language models such as GPT-2 and BERT, both based on Transformers.

Let’s start with a simple and fun model that can write like Shakespeare (well, sort of).

Generating Shakespearean Text Using a Character RNN

In a famous [2015 blog post](#) titled “The Unreasonable Effectiveness of Recurrent Neural Networks,” Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *Char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a Char-RNN model after it was trained on all of Shakespeare’s work:

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain’d into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. Let’s look at how to build a Char-RNN, step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, let's download all of Shakespeare's work, using Keras's handy `get_file()` function and downloading the data from Andrej Karpathy's [Char-RNN project](#):

```
shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Next, we must encode every character as an integer. One option is to create a custom preprocessing layer, as we did in [Chapter 13](#). But in this case, it will be simpler to use Keras's `Tokenizer` class. First we need to fit a tokenizer to the text: it will find all the characters used in the text and map each of them to a different character ID, from 1 to the number of distinct characters (it does not start at 0, so we can use that value for masking, as we will see later in this chapter):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

We set `char_level=True` to get character-level encoding rather than the default word-level encoding. Note that this tokenizer converts the text to lowercase by default (but you can set `lower=False` if you do not want that). Now the tokenizer can encode a sentence (or a list of sentences) to a list of character IDs and back, and it tells us how many distinct characters there are and the total number of characters in the text:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # number of distinct characters
>>> dataset_size = tokenizer.document_count # total number of characters
```

Let's encode the full text so each character is represented by its ID (we subtract 1 to get IDs from 0 to 38, rather than from 1 to 39):

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Before we continue, we need to split the dataset into a training set, a validation set, and a test set. We can't just shuffle all the characters in the text, so how do you split a sequential dataset?

How to Split a Sequential Dataset

It is very important to avoid any overlap between the training set, the validation set, and the test set. For example, we can take the first 90% of the text for the training set, then the next 5% for the validation set, and the final 5% for the test set. It would also

be a good idea to leave a gap between these sets to avoid the risk of a paragraph overlapping over two sets.

When dealing with time series, you would in general split across time,: for example, you might take the years 2000 to 2012 for the training set, the years 2013 to 2015 for the validation set, and the years 2016 to 2018 for the test set. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2000 to 2018, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

So, it is often safer to split across time—but this implicitly assumes that the patterns the RNN can learn in the past (in the training set) will still exist in the future. In other words, we assume that the time series is *stationary* (at least in a wide sense).³ For many time series this assumption is reasonable (e.g., chemical reactions should be fine, since the laws of chemistry don't change every day), but for many others it is not (e.g., financial markets are notoriously not stationary since patterns disappear as soon as traders spot them and start exploiting them). To make sure the time series is indeed sufficiently stationary, you can plot the model's errors on the validation set across time: if the model performs much better on the first part of the validation set than on the last part, then the time series may not be stationary enough, and you might be better off training the model on a shorter time span.

In short, splitting a time series into a training set, a validation set, and a test set is not a trivial task, and how it's done will depend strongly on the task at hand.

Now back to Shakespeare! Let's take the first 90% of the text for the training set (keeping the rest for the validation set and the test set), and create a `tf.data.Dataset` that will return each character one by one from this set:

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Chopping the Sequential Dataset into Multiple Windows

The training set now consists of a single sequence of over a million characters, so we can't just train the neural network directly on it: the RNN would be equivalent to a

³ By definition, a stationary time series's mean, variance, and *autocorrelations* (i.e., correlations between values in the time series separated by a given interval) do not change over time. This is quite restrictive; for example, it excludes time series with trends or cyclical patterns. RNNs are more tolerant in that they can learn trends and cyclical patterns.

deep net with over a million layers, and we would have a single (very long) instance to train it. Instead, we will use the dataset's `window()` method to convert this long sequence of characters into many smaller windows of text. Every instance in the dataset will be a fairly short substring of the whole text, and the RNN will be unrolled only over the length of these substrings. This is called *truncated backpropagation through time*. Let's call the `window()` method to create a dataset of short text windows:

```
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



You can try tuning `n_steps`: it is easier to train RNNs on shorter input sequences, but of course the RNN will not be able to learn any pattern longer than `n_steps`, so don't make it too small.

By default, the `window()` method creates nonoverlapping windows, but to get the largest possible training set we use `shift=1` so that the first window contains characters 0 to 100, the second contains characters 1 to 101, and so on. To ensure that all windows are exactly 101 characters long (which will allow us to create batches without having to do any padding), we set `drop_remainder=True` (otherwise the last 100 windows will contain 100 characters, 99 characters, and so on down to 1 character).

The `window()` method creates a dataset that contains windows, each of which is also represented as a dataset. It's a *nested dataset*, analogous to a list of lists. This is useful when you want to transform each window by calling its dataset methods (e.g., to shuffle them or batch them). However, we cannot use a nested dataset directly for training, as our model will expect tensors as input, not datasets. So, we must call the `flat_map()` method: it converts a nested dataset into a *flat dataset* (one that does not contain datasets). For example, suppose `{1, 2, 3}` represents a dataset containing the sequence of tensors 1, 2, and 3. If you flatten the nested dataset `[[1, 2], [3, 4, 5, 6]]`, you get back the flat dataset `[1, 2, 3, 4, 5, 6]`. Moreover, the `flat_map()` method takes a function as an argument, which allows you to transform each dataset in the nested dataset before flattening. For example, if you pass the function `lambda ds: ds.batch(2)` to `flat_map()`, then it will transform the nested dataset `[[1, 2], [3, 4, 5, 6]]` into the flat dataset `[[1, 2], [3, 4], [5, 6]]`: it's a dataset of tensors of size 2. With that in mind, we are ready to flatten our dataset:

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Notice that we call `batch(window_length)` on each window: since all windows have exactly that length, we will get a single tensor for each of them. Now the dataset contains consecutive windows of 101 characters each. Since Gradient Descent works best

when the instances in the training set are independent and identically distributed (see [Chapter 4](#)), we need to shuffle these windows. Then we can batch the windows and separate the inputs (the first 100 characters) from the target (the last character):

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, -1]))
```

[Figure 16-1](#) summarizes the dataset preparation steps discussed so far (showing windows of length 11 rather than 101, and a batch size of 3 instead of 32).

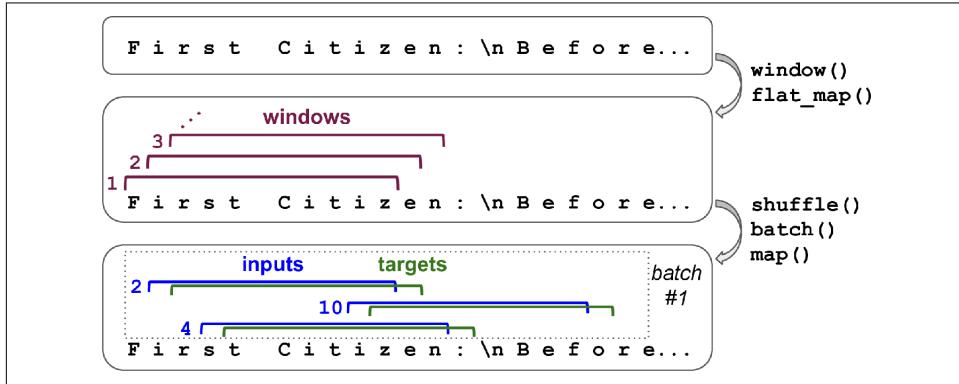


Figure 16-1. Preparing a dataset of shuffled windows

As discussed in [Chapter 13](#), categorical input features should generally be encoded, usually as one-hot vectors or as embeddings. Here, we will encode each character using a one-hot vector because there are fairly few distinct characters (only 39):

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Finally, we just need to add prefetching:

```
dataset = dataset.prefetch(1)
```

That's it! Preparing the dataset was the hardest part. Now let's create the model.

Building and Training the Char-RNN Model

To predict the next character based on the previous 100 characters, we can use an RNN with 2 GRU layers of 128 units each and 20% dropout on both the inputs (`dropout`) and the hidden states (`recurrent_dropout`). We can tweak these hyperparameters later, if needed. The output layer is a time-distributed Dense layer like we saw in [Chapter 15](#). This time this layer must have 39 units (`max_id`) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The output probabilities should sum up to 1 at each time step, so we apply the softmax activation function to the outputs of the Dense

layer. We can then compile this model, using the "sparse_categorical_crossentropy" loss and an Adam optimizer. Finally, we are ready to train the model for several epochs (this may take many hours, depending on your hardware):

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax")))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

Using the Char-RNN Model

Now we have a model that can predict the next character in text written by Shakespeare. To feed it some text, we first need to preprocess it like we did earlier, so let's create a little function for this:

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Now let's use the model to predict the next letter in some text:

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
'u'
```

Success! The model guessed right. Now let's use this model to generate new text.

Generating Fake Shakespearean Text

To generate new text using the Char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it at the end of the text, then give the extended text to the model to guess the next letter, and so on. But in practice this often leads to the same words being repeated over and over again. Instead, we can pick the next character randomly, with a probability equal to the estimated probability, using TensorFlow's `tf.random.categorical()` function. This will generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish: a temperature close to 0 will favor the high-probability characters, while a very high temperature will give all characters an equal probability. The following `next_char()` function uses this approach to pick the next character to add to the input text:

```

def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]

```

Next, we can write a small function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```

def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text

```

We are now ready to generate some text! Let's try with different temperatures:

```

>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb

```

Apparently our Shakespeare model works best at a temperature close to 1. To generate more convincing text, you could try using more GRU layers and more neurons per layer, train for longer, and add some regularization (for example, you could set `recurrent_dropout=0.3` in the GRU layers). Moreover, the model is currently incapable of learning patterns longer than `n_steps`, which is just 100 characters. You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences. Alternatively, you could use a stateful RNN.

Stateful RNN

Until now, we have used only *stateless RNNs*: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away, as it is not needed anymore. What if we told the RNN to preserve this final state after processing one training batch and use it as the initial state for the next training batch? This way the model can learn long-term patterns despite only backpropagating through short sequences. This is called a *stateful RNN*. Let's see how to build one.

First, note that a stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing we need to do to build a stateful RNN is to use sequential and nonoverlap-

ping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs). When creating the `Dataset`, we must therefore use `shift=n_steps` (instead of `shift=1`) when calling the `window()` method. Moreover, we must obviously *not* call the `shuffle()` method. Unfortunately, batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if we were to call `batch(32)`, then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these window where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use “batches” containing a single window:

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

Figure 16-2 summarizes the first steps.

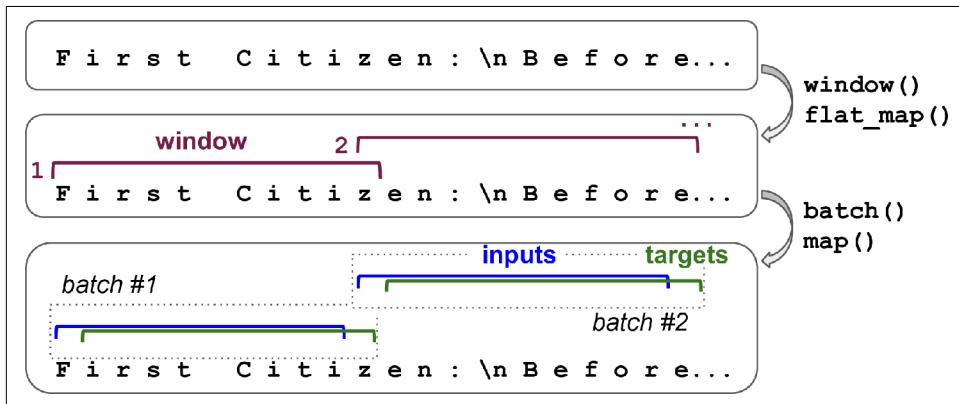


Figure 16-2. Preparing a dataset of consecutive sequence fragments for a stateful RNN

Batching is harder, but it is not impossible. For example, we could chop Shakespeare’s text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` to create proper consecutive batches, where the n^{th} input sequence in a batch starts off exactly where the n^{th} input sequence ended in the previous batch (see the notebook for the full code).

Now let's create the stateful RNN. First, we need to set `stateful=True` when creating every recurrent layer. Second, the stateful RNN needs to know the batch size (since it will preserve a state for each input sequence in the batch), so we must set the `batch_input_shape` argument in the first layer. Note that we can leave the second dimension unspecified, since the inputs could have any length:

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                   activation="softmax"))
])
```

At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small callback:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And now we can compile and fit the model (for more epochs, because each epoch is much shorter than earlier, and there is only one instance per batch):

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```



After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical *stateless* model, and copy the stateful model's weights to this model.

Now that we have built a character-level model, it's time to look at word-level models and tackle a common natural language processing task: *sentiment analysis*. In the process we will learn how to handle sequences of variable lengths using masking.

Sentiment Analysis

If MNIST is the “hello world” of computer vision, then the IMDb reviews dataset is the “hello world” of natural language processing: it consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous [Internet Movie Database](#), along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount

of time, but challenging enough to be fun and rewarding. Keras provides a simple function to load it:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Where are the movie reviews? Well, as you can see, the dataset is already preprocessed for you: `X_train` consists of a list of reviews, each of which is represented as a NumPy array of integers, where each integer represents a word. All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words). The integers 0, 1, and 2 are special: they represent the padding token, the *start-of-sequence* (SSS) token, and unknown words, respectively. If you want to visualize a review, you can decode it like this:

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate("<pad>", "<sos>", "<unk>"):
...     id_to_word[id_] = token
...
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

In a real project, you will have to preprocess the text yourself. You can do that using the same `Tokenizer` class we used earlier, but this time setting `char_level=False` (which is the default). When encoding words, it filters out a lot of characters, including most punctuation, line breaks, and tabs (but you can change this by setting the `filters` argument). Most importantly, it uses spaces to identify word boundaries. This is OK for English and many other scripts (written languages) that use spaces between words, but not all scripts use spaces this way. Chinese does not use spaces between words, Vietnamese uses spaces even within words, and languages such as German often attach multiple words together, without spaces. Even in English, spaces are not always the best way to tokenize text: think of “San Francisco” or “#ILoveDeepLearning.”

Fortunately, there are better options! The [2018 paper⁴](#) by Taku Kudo introduced an unsupervised learning technique to tokenize and detokenize text at the subword level in a language-independent way, treating spaces like other characters. With this approach, even if your model encounters a word it has never seen before, it can still reasonably guess what it means. For example, it may never have seen the word “smartest” during training, but perhaps it learned the word “smart” and it also learned that the suffix “est” means “the most,” so it can infer the meaning of

⁴ Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” arXiv preprint arXiv:1804.10959 (2018).

“smartest.” Google’s *SentencePiece* project provides an open source implementation, described in a paper⁵ by Taku Kudo and John Richardson.

Another option was proposed in an earlier paper⁶ by Rico Sennrich et al. that explored other ways of creating subword encodings (e.g., using *byte pair encoding*). Last but not least, the TensorFlow team released the `TF.Text` library in June 2019, which implements various tokenization strategies, including *WordPiece*⁷ (a variant of byte pair encoding).

If you want to deploy your model to a mobile device or a web browser, and you don’t want to have to write a different preprocessing function every time, then you will want to handle preprocessing using only TensorFlow operations, so it can be included in the model itself. Let’s see how. First, let’s load the original IMDb reviews, as text (byte strings), using TensorFlow Datasets (introduced in [Chapter 13](#)):

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Next, let’s write the preprocessing function:

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z]", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

It starts by truncating the reviews, keeping only the first 300 characters of each: this will speed up training, and it won’t impact performance too much because you can generally tell whether a review is positive or not in the first sentence or two. Then it uses *regular expressions* to replace `
` tags with spaces, and to replace any characters other than letters and quotes with spaces. For example, the text “Well, I can’t`
`” will become “Well I can’t”. Finally, the `preprocess()` function splits the reviews by the spaces, which returns a ragged tensor, and it converts this ragged tensor to a dense tensor, padding all reviews with the padding token “`<pad>`” so that they all have the same length.

⁵ Taku Kudo and John Richardson, “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing,” arXiv preprint arXiv:1808.06226 (2018).

⁶ Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units,” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.

⁷ Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation,” arXiv preprint arXiv:1609.08144 (2016).

Next, we need to construct the vocabulary. This requires going through the whole training set once, applying our `preprocess()` function, and using a `Counter` to count the number of occurrences of each word:

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Let's look at the three most common words:

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Great! We probably don't need our model to know all the words in the dictionary to get good performance, though, so let's truncate the vocabulary, keeping only the 10,000 most common words:

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Now we need to add a preprocessing step to replace each word with its ID (i.e., its index in the vocabulary). Just like we did in [Chapter 13](#), we will create a lookup table for this, using 1,000 out-of-vocabulary (oov) buckets:

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

We can then use this table to look up the IDs of a few words:

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,   12,   11, 10054]])>
```

Note that the words "this," "movie," and "was" were found in the table, so their IDs are lower than 10,000, while the word "faaaaaantastic" was not found, so it was mapped to one of the oov buckets, with an ID greater than or equal to 10,000.



TF Transform (introduced in [Chapter 13](#)) provides some useful functions to handle such vocabularies. For example, check out the `tft.compute_and_apply_vocabulary()` function: it will go through the dataset to find all distinct words and build the vocabulary, and it will generate the TF operations required to encode each word using this vocabulary.

Now we are ready to create the final training set. We batch the reviews, then convert them to short sequences of words using the `preprocess()` function, then encode

these words using a simple `encode_words()` function that uses the table we just built, and finally prefetch the next batch:

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

At last we can create the model and train it:

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

The first layer is an `Embedding` layer, which will convert word IDs into embeddings (introduced in [Chapter 13](#)). The embedding matrix needs to have one row per word ID (`vocab_size + num_oov_buckets`) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune). Whereas the inputs of the model will be 2D tensors of shape `[batch size, time steps]`, the output of the `Embedding` layer will be a 3D tensor of shape `[batch size, time steps, embedding size]`.

The rest of the model is fairly straightforward: it is composed of two `GRU` layers, with the second one returning only the output of the last time step. The output layer is just a single neuron using the `sigmoid` activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie. We then compile the model quite simply, and we fit it on the dataset we prepared earlier, for a few epochs.

Masking

As it stands, the model will need to learn that the padding tokens should be ignored. But we already know that! Why don't we tell the model to ignore the padding tokens, so that it can focus on the data that actually matters? It's actually trivial: simply add

`mask_zero=True` when creating the `Embedding` layer. This means that padding tokens (whose ID is 0)⁸ will be ignored by all downstream layers. That's all!

The way this works is that the `Embedding` layer creates a *mask tensor* equal to `K.not_equal(inputs, 0)` (where `K = keras.backend`): it is a Boolean tensor with the same shape as the inputs, and it is equal to `False` anywhere the word IDs are 0, or `True` otherwise. This mask tensor is then automatically propagated by the model to all subsequent layers, as long as the time dimension is preserved. So in this example, both `GRU` layers will receive this mask automatically, but since the second `GRU` layer does not return sequences (it only returns the output of the last time step), the mask will not be transmitted to the `Dense` layer. Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is `False`). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step. If the mask propagates all the way to the output (in models that output sequences, which is not the case in this example), then it will be applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0).



The `LSTM` and `GRU` layers have an optimized implementation for GPUs, based on Nvidia's cuDNN library. However, this implementation does not support masking. If your model uses a mask, then these layers will fall back to the (much slower) default implementation. Note that the optimized implementation also requires you to use the default values for several hyperparameters: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias`, and `reset_after`.

All layers that receive the mask must support masking (or else an exception will be raised). This includes all recurrent layers, as well as the `TimeDistributed` layer and a few other layers. Any layer that supports masking must have a `supports_masking` attribute equal to `True`. If you want to implement your own custom layer with masking support, you should add a `mask` argument to the `call()` method (and obviously make the method use the mask somehow). Additionally, you should set `self.supports_masking = True` in the constructor. If your layer does not start with an `Embedding` layer, you may use the `keras.layers.Masking` layer instead: it sets the mask to `K.any(K.not_equal(inputs, 0), axis=-1)`, meaning that time steps where the last dimension is full of zeros will be masked out in subsequent layers (again, as long as the time dimension exists).

⁸ Their ID is 0 only because they are the most frequent “words” in the dataset. It would probably be a good idea to ensure that the padding tokens are always encoded as 0, even if they are not the most frequent.

Using masking layers and automatic mask propagation works best for simple Sequential models. It will not always work for more complex models, such as when you need to mix Conv1D layers with recurrent layers. In such cases, you will need to explicitly compute the mask and pass it to the appropriate layers, using either the Functional API or the Subclassing API. For example, the following model is identical to the previous model, except it is built using the Functional API and handles masking manually:

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

After training for a few epochs, this model will become quite good at judging whether a review is positive or not. If you use the `TensorBoard()` callback, you can visualize the embeddings in TensorBoard as they are being learned: it is fascinating to see words like “awesome” and “amazing” gradually cluster on one side of the embedding space, while words like “awful” and “terrible” cluster on the other side. Some words are not as positive as you might expect (at least with this model), such as the word “good,” presumably because many negative reviews contain the phrase “not good.” It’s impressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews. Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately we don’t, but perhaps we can reuse word embeddings trained on some other large text corpus (e.g., Wikipedia articles), even if it is not composed of movie reviews? After all, the word “amazing” generally has the same meaning whether you use it to talk about movies or anything else. Moreover, perhaps embeddings would be useful for sentiment analysis even if they were trained on another task: since words like “awesome” and “amazing” have a similar meaning, they will likely cluster in the embedding space even for other tasks (e.g., predicting the next word in a sentence). If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis. So instead of using so many parameters to learn word embeddings, let’s see if we can’t just reuse pretrained embeddings.

Reusing Pretrained Embeddings

The TensorFlow Hub project makes it easy to reuse pretrained model components in your own models. 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, along with its pretrained weights, and included in your model. Easy!

For example, let's use the `nnlm-en-dim50` sentence embedding module, version 1, in our sentiment analysis model:

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                  dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])
```

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 We can then add two simple `Dense` layers to create a good sentiment analysis model. By default, a `hub.KerasLayer` is not trainable, but you can set `trainable=True` when creating it to change that so that you can fine-tune it for your task.



Not all TF Hub modules support TensorFlow 2, so make sure you choose a module that does.

Next, we can just load the IMDb reviews dataset—no need to preprocess it (except for batching and prefetching)—and directly train the model:

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

Note that the last part of the TF Hub module URL specified that we wanted version 1 of the model. This versioning ensures that if a new module version is released, it will not break our model. Conveniently, if you just enter this URL in a web browser, you

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 vectors gets shorter as n grows.

will get the documentation for this module. By default, TF Hub will cache the downloaded files into the local system’s temporary directory. You may prefer to download them into a more permanent directory to avoid having to download them again after every system cleanup. To do that, set the `TFHUB_CACHE_DIR` environment variable to the directory of your choice (e.g., `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

So far, we have looked at time series, text generation using Char-RNN, and sentiment analysis using word-level RNN models, training our own word embeddings or reusing pretrained embeddings. Let’s now look at another important NLP task: *neural machine translation* (NMT), first using a pure Encoder–Decoder model, then improving it with attention mechanisms, and finally looking the extraordinary Transformer architecture.

An Encoder–Decoder Network for Neural Machine Translation

Let’s take a look at a simple [neural machine translation model](#)¹⁰ that will translate English sentences to French (see Figure 16-3).

In short, the English sentences are fed to the encoder, and the decoder outputs the French translations. Note that the French translations are also used as inputs to the decoder, but shifted back by one step. In other words, the decoder is given as input the word that it *should* have output at the previous step (regardless of what it actually output). For the very first word, it is given the start-of-sequence (SOS) token. The decoder is expected to end the sentence with an end-of-sequence (EOS) token.

Note that the English sentences are reversed before they are fed to the encoder. For example, “I drink milk” is reversed to “milk drink I.” This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that’s generally the first thing that the decoder needs to translate.

Each word is initially represented by its ID (e.g., 288 for the word “milk”). Next, an embedding layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.

¹⁰ Ilya Sutskever et al., “Sequence to Sequence Learning with Neural Networks,” arXiv preprint arXiv:1409.3215 (2014).

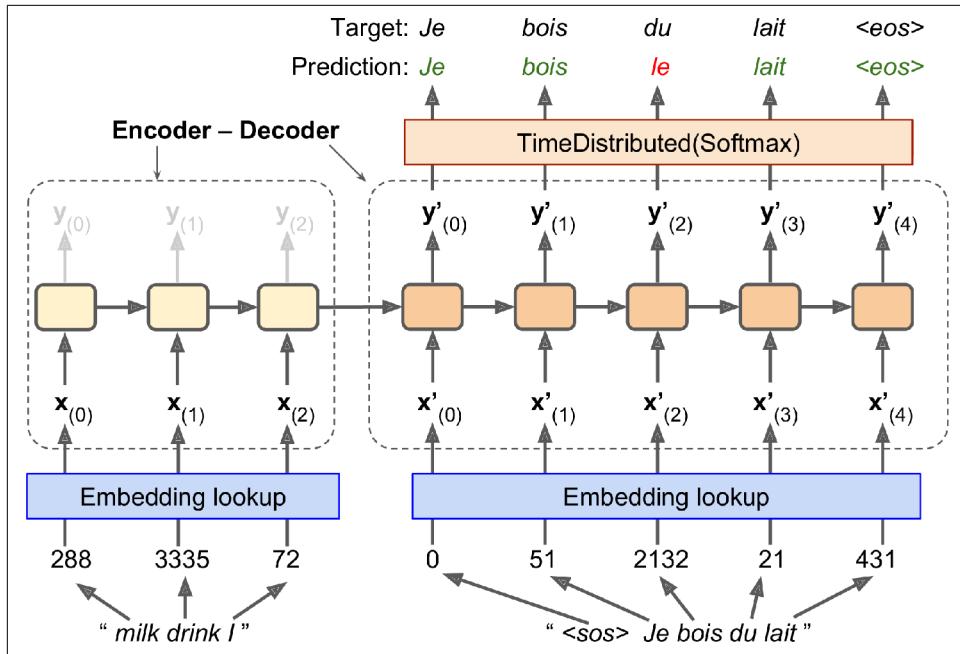


Figure 16-3. A simple machine translation model

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities. For example, at the first step the word “*Je*” may have a probability of 20%, “*Tu*” may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, so you can train the model using the “`sparse_categorical_crossentropy`” loss, much like we did in the Char-RNN model.

Note that at inference time (after training), you will not have the target sentence to feed to the decoder. Instead, simply feed the decoder the word that it output at the previous step, as shown in [Figure 16-4](#) (this will require an embedding lookup that is not shown in the diagram).

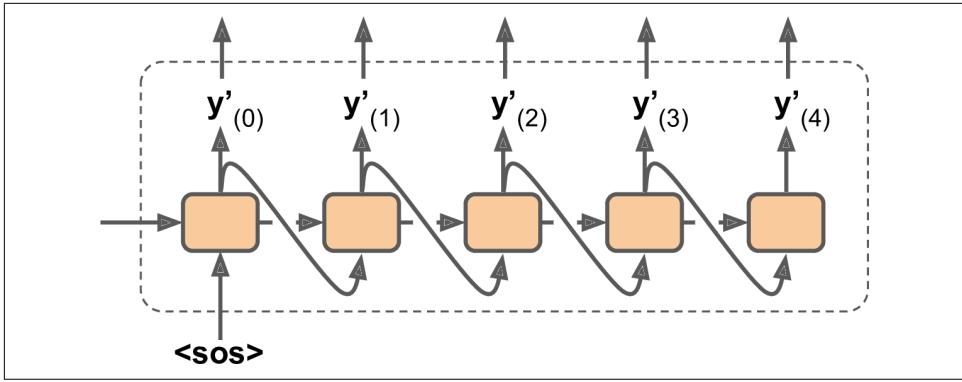


Figure 16-4. Feeding the previous output word as input at inference time

OK, now you have the big picture. Still, there are a few more details to handle if you implement this model:

- So far we have assumed that all input sequences (to the encoder and to the decoder) have a constant length. But obviously sentence lengths vary. Since regular tensors have fixed shapes, they can only contain sentences of the same length. You can use masking to handle this, as discussed earlier. However, if the sentences have very different lengths, you can't just crop them like we did for sentiment analysis (because we want full translations, not cropped translations). Instead, group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on), using padding for the shorter sequences to ensure all sentences in a bucket have the same length (check out the `tf.data.experimental.bucket_by_sequence_length()` function for this). For example, “I drink milk” becomes “<pad> <pad> <pad> milk drink I.”
- We want to ignore any output past the EOS token, so these tokens should not contribute to the loss (they must be masked out). For example, if the model outputs “Je bois du lait <eos> oui,” the loss for the last word should be ignored.
- When the output vocabulary is large (which is the case here), outputting a probability for each and every possible word would be terribly slow. If the target vocabulary contains, say, 50,000 French words, then the decoder would output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This *sampled softmax* technique was [introduced](#) in

2015 by Sébastien Jean et al.¹¹ In TensorFlow you can use the `tf.nn.sampled_softmax_loss()` function for this during training and use the normal softmax function at inference time (sampled softmax cannot be used at inference time because it requires knowing the target).

The TensorFlow Addons project includes many sequence-to-sequence tools to let you easily build production-ready Encoder–Decoders. For example, the following code creates a basic Encoder–Decoder model, similar to the one represented in Figure 16-3:

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                     outputs=[Y_proba])
```

The code is mostly self-explanatory, but there are a few points to note. First, we set `return_state=True` when creating the LSTM layer so that we can get its final hidden state and pass it to the decoder. Since we are using an LSTM cell, it actually returns two hidden states (short term and long term). The `TrainingSampler` is one of several samplers available in TensorFlow Addons: their role is to tell the decoder at each step what it should pretend the previous output was. During inference, this should be the

¹¹ Sébastien Jean et al., “On Using Very Large Target Vocabulary for Neural Machine Translation,” *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing 1* (2015): 1–10.

embedding of the token that was actually output. During training, it should be the embedding of the previous target token: this is why we used the `TrainingSampler`. In practice, it is often a good idea to start training with the embedding of the target of the previous time step and gradually transition to using the embedding of the actual token that was output at the previous step. This idea was introduced in a 2015 paper¹² by Samy Bengio et al. The `ScheduledEmbeddingTrainingSampler` will randomly choose between the target or the actual output, with a probability that you can gradually change during training.

Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is “causal,” meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, but for many NLP tasks, such as Neural Machine Translation, it is often preferable to look ahead at the next words before encoding a given word. For example, consider the phrases “the Queen of the United Kingdom,” “the queen of hearts,” and “the queen bee”: to properly encode the word “queen,” you need to look ahead. To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then simply combine their outputs at each time step, typically by concatenating them. This is called a *bidirectional recurrent layer* (see Figure 16-5).

To implement a bidirectional recurrent layer in Keras, wrap a recurrent layer in a `keras.layers.Bidirectional` layer. For example, the following code creates a bidirectional GRU layer:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```



The `Bidirectional` layer will create a clone of the `GRU` layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the `GRU` layer has 10 units, the `Bidirectional` layer will output 20 values per time step.

¹² Samy Bengio et al., “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks,” arXiv preprint arXiv:1506.03099 (2015).

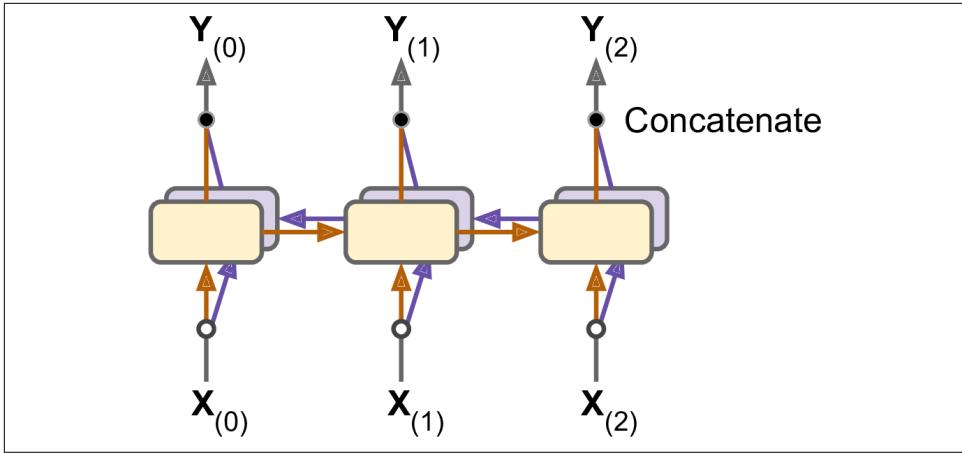


Figure 16-5. A bidirectional recurrent layer

Beam Search

Suppose you train an Encoder–Decoder model, and use it to translate the French sentence “Comment vas-tu?” to English. You are hoping that it will output the proper translation (“How are you?”), but unfortunately it outputs “How will you?” Looking at the training set, you notice many sentences such as “Comment vas-tu jouer?” which translates to “How will you play?” So it wasn’t absurd for the model to output “How will” after seeing “Comment vas.” Unfortunately, in this case it was a mistake, and the model could not go back and fix it, so it tried to complete the sentence as best it could. By greedily outputting the most likely word at every step, it ended up with a suboptimal translation. How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the k most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the k most likely sentences. The parameter k is called the *beam width*.

For example, suppose you use the model to translate the sentence “Comment vas-tu?” using beam search with a beam width of 3. At the first decoder step, the model will output an estimated probability for each possible word. Suppose the top three words are “How” (75% estimated probability), “What” (3%), and “You” (1%). That’s our short list so far. Next, we create three copies of our model and use them to find the next word for each sentence. Each model will output one estimated probability per word in the vocabulary. The first model will try to find the next word in the sentence “How,” and perhaps it will output a probability of 36% for the word “will,” 32% for the word “are,” 16% for the word “do,” and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “How.” The second model will try to complete the sentence “What”; it might output a conditional probability of 50% for

the word “are,” and so on. Assuming the vocabulary has 10,000 words, each model will output 10,000 probabilities.

Next, we compute the probabilities of each of the 30,000 two-word sentences that these models considered ($3 \times 10,000$). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “How” was 75%, while the estimated conditional probability of the word “will” (given that the first word is “How”) was 36%, so the estimated probability of the sentence “How will” is $75\% \times 36\% = 27\%$. After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. Perhaps they all start with the word “How”: “How will” (27%), “How are” (24%), and “How do” (12%). Right now, the sentence “How will” is winning, but “How are” has not been eliminated.

Then we repeat the same process: we use three models to predict the next word in each of these three sentences, and we compute the probabilities of all 30,000 three-word sentences we considered. Perhaps the top three are now “How are you” (10%), “How do you” (8%), and “How will you” (2%). At the next step we may get “How do you do” (7%), “How are you <eos>” (6%), and “How are you doing” (3%). Notice that “How will” was eliminated, and we now have three perfectly reasonable translations. We boosted our Encoder–Decoder model’s performance without any extra training, simply by using it more wisely.

You can implement beam search fairly easily using TensorFlow Addons:

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

We first create a `BeamSearchDecoder`, which wraps all the decoder clones (in this case 10 clones). Then we create one copy of the encoder’s final state for each decoder clone, and we pass these states to the decoder, along with the start and end tokens.

With all this, you can get good translations for fairly short sentences (especially if you use pretrained word embeddings). Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word “milk” to its translation “lait” in [Figure 16-3](#): it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

This was the core idea in a groundbreaking [2014 paper](#)¹³ by Dzmitry Bahdanau et al. They introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “lait,” it will focus its attention on the word “milk.” This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words).¹⁴

[Figure 16-6](#) shows this model’s architecture (slightly simplified, as we will see). On the left, you have the encoder and the decoder. Instead of just sending the encoder’s final hidden state to the decoder (which is still done, although it is not shown in the figure), we now send all of its outputs to the decoder. At each time step, the decoder’s memory cell computes a weighted sum of all these encoder outputs: this determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to word number 2 (“milk”) than to the other two words, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

¹³ Dzmitry Bahdanau et al., “Neural Machine Translation by Jointly Learning to Align and Translate,” arXiv preprint arXiv:1409.0473 (2014).

¹⁴ The most common metric used in NMT is the BiLingual Evaluation Understudy (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of n -grams (sequences of n words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced n -grams in the target translations.

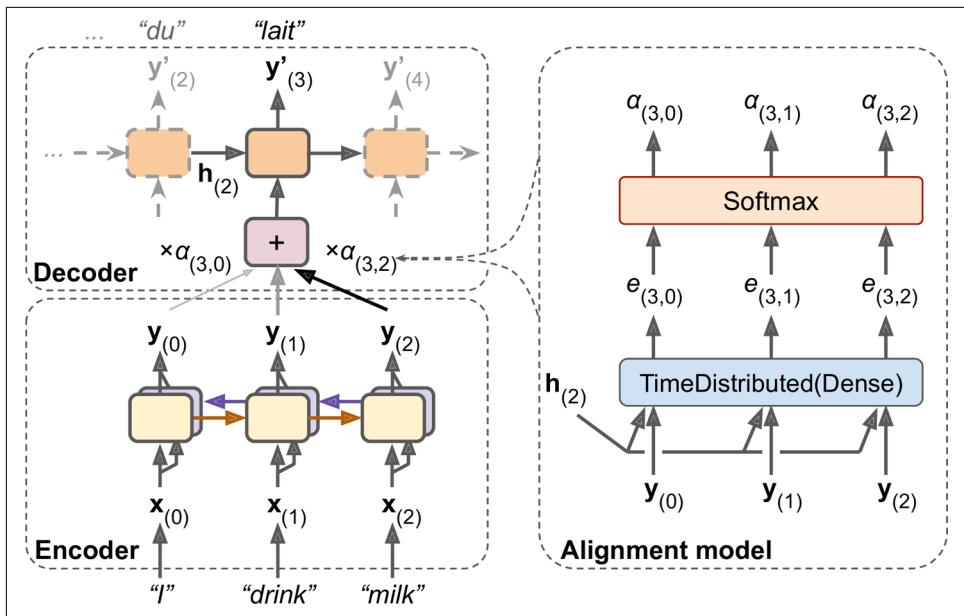


Figure 16-6. Neural machine translation using an Encoder–Decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? It's actually pretty simple: they are generated by a type of small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the Encoder–Decoder model. This alignment model is illustrated on the righthand side of Figure 16-6. It starts with a time-distributed Dense layer¹⁵ with a single neuron, which receives as input all the encoder outputs, concatenated with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$): this score measures how well each output is aligned with the decoder's previous hidden state. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1 (since the softmax layer is not time-distributed). This particular attention mechanism is called *Bahdanau attention* (named after the paper's first author). Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

¹⁵ Recall that a time-distributed Dense layer is equivalent to a regular Dense layer that you apply independently at each time step (only much faster).



If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism was proposed shortly after, in a 2015 paper¹⁶ by Minh-Thang Luong et al. Because the goal of the attention mechanism is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the *dot product* (see Chapter 4) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it much faster. For this to be possible, both vectors must have the same dimensionality. This is called *Luong attention* (again, after the paper's first author), or sometimes *multiplicative attention*. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification they proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$) rather than $\mathbf{h}_{(t-1)}$, then to use the output of the attention mechanism (noted $\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions (rather than using it to compute the decoder's current hidden state). They also proposed a variant of the dot product mechanism where the encoder outputs first go through a linear transformation (i.e., a time-distributed Dense layer without a bias term) before the dot products are computed. This is called the "general" dot product approach. They compared both dot product approaches to the concatenative attention mechanism (adding a rescaling parameter vector \mathbf{v}), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in Equation 16-1.

¹⁶ Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

Equation 16-1. Attention mechanisms

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t, i)} \mathbf{y}_{(i)}$$

with $\alpha_{(t, i)} = \frac{\exp(e_{(t, i)})}{\sum_{i'} \exp(e_{(t, i')})}$

and $e_{(t, i)} = \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & dot \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & general \\ \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & concat \end{cases}$

Here is how you can add Luong attention to an Encoder–Decoder model using TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(  
    units, encoder_state, memory_sequence_length=encoder_sequence_length)  
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(  
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

We simply wrap the decoder cell in an `AttentionWrapper`, and we provide the desired attention mechanism (Luong attention in this example).

Visual Attention

Attention mechanisms are now used for a variety of purposes. One of their first applications beyond NMT was in generating image captions using **visual attention**.¹⁷ a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image. For example, in [Figure 16-7](#), the model generated the caption “A woman is throwing a frisbee in a park,” and you can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”: clearly, most of its attention was focused on the frisbee.

¹⁷ Kelvin Xu et al., “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,” *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.



Figure 16-7. Visual attention: an input image (left) and the model's focus before producing the word "frisbee" (right)¹⁸

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake: for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow," then you can go back and check what the model focused on when it output the word "wolf." You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great [2016 paper¹⁹](#) by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement (think of a system deciding whether or not it should grant you a loan).

¹⁸ This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.

¹⁹ Marco Tulio Ribeiro et al., "Why Should I Trust You?: Explaining the Predictions of Any Classifier," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

Attention mechanisms are so powerful that you can actually build state-of-the-art models using only attention mechanisms.

Attention Is All You Need: The Transformer Architecture

In a groundbreaking 2017 paper,²⁰ a team of Google researchers suggested that “Attention Is All You Need.” They managed to create an architecture called the *Transformer*, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers,²¹ just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). As an extra bonus, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

The Transformer architecture is represented in Figure 16-8.

²⁰ Ashish Vaswani et al., “Attention Is All You Need,” *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

²¹ Since the Transformer uses time-distributed Dense layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

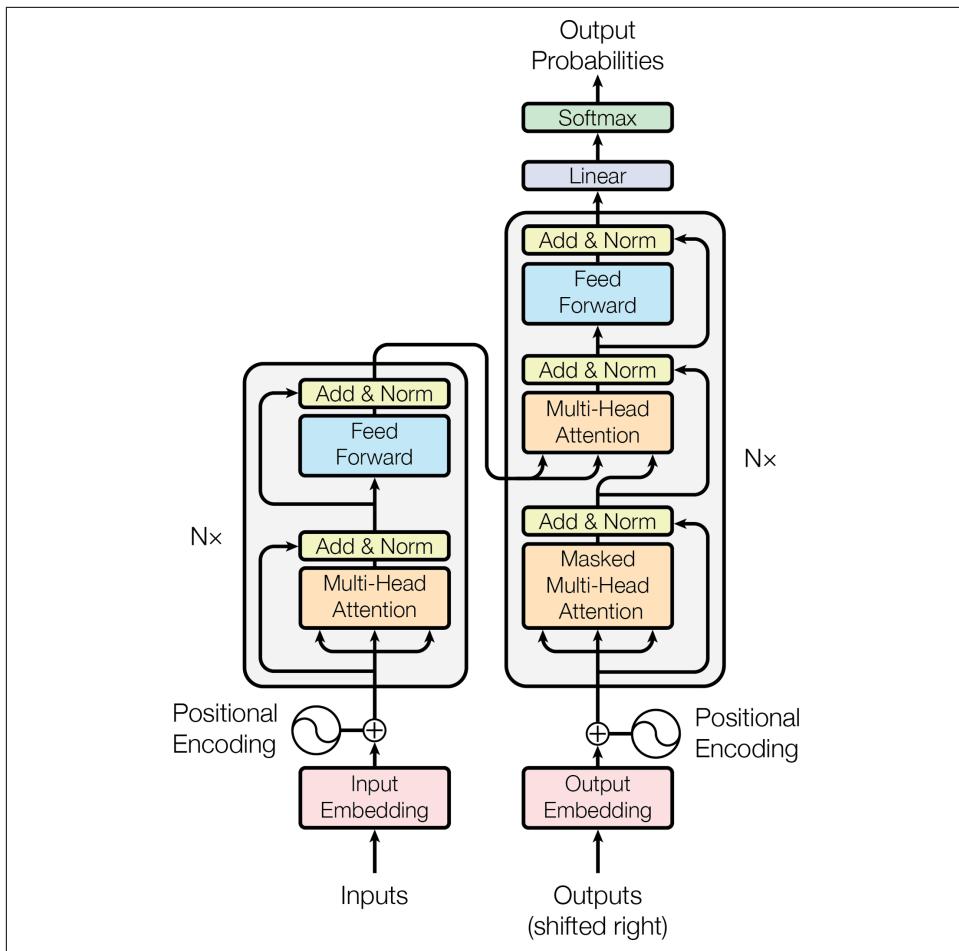


Figure 16-8. The Transformer architecture²²

Let's walk through this figure:

- The lefthand part is the encoder. Just like earlier, it takes as input a batch of sentences represented as sequences of word IDs (the input shape is $[batch\ size, max\ input\ sentence\ length]$), and it encodes each word into a 512-dimensional representation (so the encoder's output shape is $[batch\ size, max\ input\ sentence\ length, 512]$). Note that the top part of the encoder is stacked N times (in the paper, $N = 6$).

²² This is figure 1 from the paper, reproduced with the kind authorization of the authors.

- The righthand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning). It also receives the outputs of the encoder (i.e., the arrows coming from the left side). Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels. Just like earlier, the decoder outputs a probability for each possible next word, at each time step (its output shape is $[batch\ size, max\ output\ sentence\ length, vocabulary\ length]$).
- During inference, the decoder cannot be fed targets, so we feed it the previously output words (starting with a start-of-sequence token). So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).
- Looking more closely, you can see that you are already familiar with most components: there are two embedding layers, $5 \times N$ skip connections, each of them followed by a layer normalization layer, $2 \times N$ “Feed Forward” modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. All of these layers are time-distributed, so each word is treated independently of all the others. But how can we translate a sentence by only looking at one word at a time? Well, that's where the new components come in:
 - The encoder's *Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word “Queen” in the sentence “They welcomed the Queen of the United Kingdom” will depend on all the words in the sentence, but it will probably pay more attention to the words “United” and “Kingdom” than to the words “They” or “welcomed.” This attention mechanism is called *self-attention* (the sentence is paying attention to itself). We will discuss exactly how it works shortly. The decoder's *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it. Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word “Queen” in the input sentence when it is about to output this word's translation.
 - The *positional embeddings* are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence. The n^{th} positional embedding is added to the word embedding of the n^{th} word in each sentence. This gives the model access to each word's position, which is needed because the Multi-Head Attention layers do not consider the order or the position of the words; they only look at their relationships. Since all the other layers are

time-distributed, they have no way of knowing the position of each word (either relative or absolute). Obviously, the relative and absolute word positions are important, so we need to give this information to the Transformer somehow, and positional embeddings are a good way to do this.

Let's look a bit closer at both these novel components of the Transformer architecture, starting with the positional embeddings.

Positional embeddings

A positional embedding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional embedding is simply added to the word embedding of the i^{th} word in the sentence. These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the sine and cosine functions of different frequencies. The positional embedding matrix \mathbf{P} is defined in [Equation 16-2](#) and represented at the bottom of [Figure 16-9](#) (transposed), where $P_{p,i}$ is the i^{th} component of the embedding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional embeddings

$$P_{p,2i} = \sin(p/10000^{2i/d})$$

$$P_{p,2i+1} = \cos(p/10000^{2i/d})$$

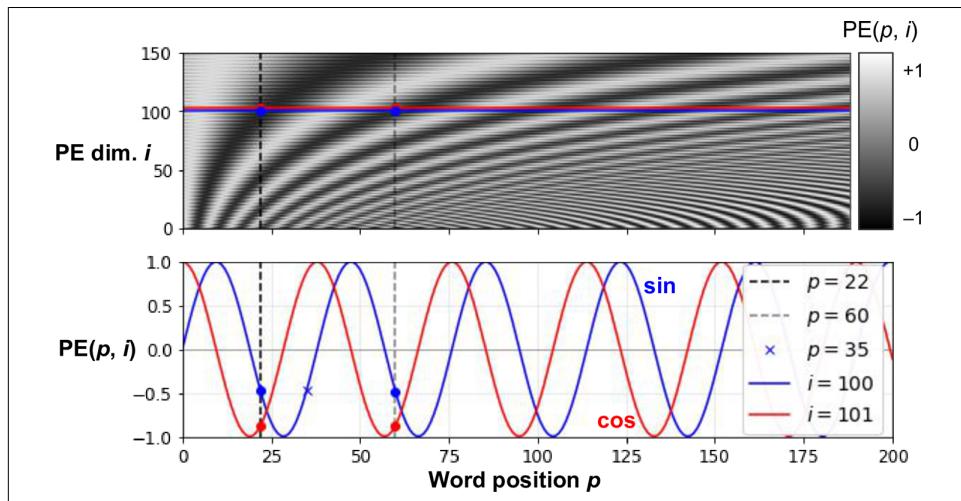


Figure 16-9. Sine/cosine positional embedding matrix (transposed, top) with a focus on two values of i (bottom)

This solution gives the same performance as learned positional embeddings do, but it can extend to arbitrarily long sentences, which is why it's favored. After the positional embeddings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional embedding for each position (e.g., the positional embedding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the bottom left of [Figure 16-9](#), and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional embedding values in the embedding dimensions $i = 100$ and $i = 101$, as you can see in [Figure 16-9](#). This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 25$ and $p = 35$ (marked by a cross).

There is no `PositionalEmbedding` layer in TensorFlow, but it is easy to create one. For efficiency reasons, we precompute the positional embedding matrix in the constructor (so we need to know the maximum sentence length, `max_steps`, and the number of dimensions for each word representation, `max_dims`). Then the `call()` method crops this embedding matrix to the size of the inputs, and it adds it to the inputs. Since we added an extra first dimension of size 1 when creating the positional embedding matrix, the rules of broadcasting will ensure that the matrix gets added to every sentence in the inputs:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**((2 * i) / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**((2 * i) / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Then we can create the first layers of the Transformer:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Now let's look deeper into the heart of the Transformer model: the Multi-Head Attention layer.

Multi-Head Attention

To understand how a Multi-Head Attention layer works, we must first understand the *Scaled Dot-Product Attention* layer, which it is based on. Let's suppose the encoder analyzed the input sentence "They played chess," and it managed to understand that the word "They" is the subject and the word "played" is the verb, so it encoded this information in the representations of these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it's as if the encoder created a dictionary {"subject": "They", "verb": "played", ...} and the decoder wanted to look up the value that corresponds to the key "verb." However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the *query*) will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1. Then the model can compute a weighted sum of the corresponding values, so if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played." In short, you can think of this whole process as a differentiable dictionary lookup. The similarity measure used by the Transformer is just the dot product, like in Luong attention. In fact, the equation is the same as for Luong attention, except for a scaling factor. The equation is shown in [Equation 16-3](#), in a vectorized form.

Equation 16-3. Scaled Dot-Product Attention

$$\text{Attention } (\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{keys}}}} \right) \mathbf{V}$$

In this equation:

- \mathbf{Q} is a matrix containing one row per query. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- \mathbf{K} is a matrix containing one row per key. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.

- \mathbf{V} is a matrix containing one row per value. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of each value.
- The shape of $\mathbf{Q} \mathbf{K}^T$ is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).
- The scaling factor scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the Masked Multi-Head Attention layer.

In the encoder, this equation is applied to every input sentence in the batch, with \mathbf{Q} , \mathbf{K} , and \mathbf{V} all equal to the list of words in the input sentence (so each word in the sentence will be compared to every word in the same sentence, including itself). Similarly, in the decoder’s masked attention layer, the equation will be applied to every target sentence in the batch, with \mathbf{Q} , \mathbf{K} , and \mathbf{V} all equal to the list of words in the target sentence, but this time using a mask to prevent any word from comparing itself to words located after it (at inference time the decoder will only have access to the words it already output, not to future words, so during training we must mask out future output tokens). In the upper attention layer of the decoder, the keys \mathbf{K} and values \mathbf{V} are simply the list of word encodings produced by the encoder, and the queries \mathbf{Q} are the list of word encodings produced by the decoder.

The `keras.layers.Attention` layer implements Scaled Dot-Product Attention, efficiently applying [Equation 16-3](#) to multiple sentences in a batch. Its inputs are just like \mathbf{Q} , \mathbf{K} , and \mathbf{V} , except with an extra batch dimension (the first dimension).



In TensorFlow, if \mathbf{A} and \mathbf{B} are tensors with more than two dimensions—say, of shape $[2, 3, 4, 5]$ and $[2, 3, 5, 6]$ respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in \mathbf{A} will be multiplied by the matrix at the i^{th} row and j^{th} column in \mathbf{B} . Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape $[2, 3, 4, 6]$.

If we ignore the skip connections, the layer normalization layers, the Feed Forward blocks, and the fact that this is Scaled Dot-Product Attention, not exactly Multi-Head Attention, then the rest of the Transformer model can be implemented like this:

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

The `use_scale=True` argument creates an additional parameter that lets the layer learn how to properly downscale the similarity scores. This is a bit different from the Transformer model, which always downscales the similarity scores by the same factor ($\sqrt{d_{\text{keys}}}$). The `causal=True` argument when creating the second attention layer ensures that each output token only attends to previous output tokens, not future ones.

Now it's time to look at the final piece of the puzzle: what is a Multi-Head Attention layer? Its architecture is shown in [Figure 16-10](#).

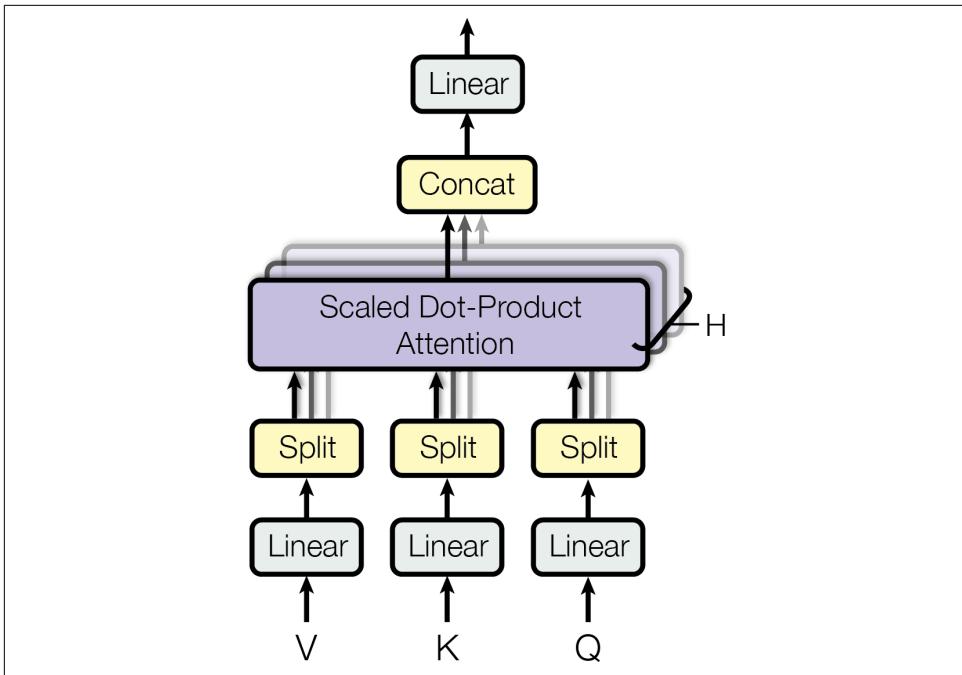


Figure 16-10. Multi-Head Attention layer architecture²³

As you can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed Dense layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed). But why? What is the intuition behind this architecture? Well, consider the word “played” we discussed earlier (in the sentence “They played chess”). The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the past tense. In short, the word representation encodes many different characteristics of the word. If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot. This is why the Multi-Head Attention layer applies multiple different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word’s characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a

²³ This is the right part of figure 2 from the paper, reproduced with the kind authorization of the authors.

verb, another linear layer will extract just the fact that it is past tense, and so on. Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

At the time of this writing, there is no `Transformer` class or `MultiHeadAttention` class available for TensorFlow 2. However, you can check out TensorFlow's great [tutorial for building a Transformer model for language understanding](#). Moreover, the TF Hub team is currently porting several Transformer-based modules to TensorFlow 2, and they should be available soon. In the meantime, I hope I have demonstrated that it is not that hard to implement a Transformer yourself, and it is certainly a great exercise!

Recent Innovations in Language Models

The year 2018 has been called the “ImageNet moment for NLP”: progress was astounding, with larger and larger LSTM and Transformer-based architectures trained on immense datasets. I highly recommend you check out the following papers, all published in 2018:

- The [ELMo paper²⁴](#) by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model. For example, the word “queen” will not have the same embedding in “Queen of the United Kingdom” and in “queen bee.”
- The [ULMFiT paper²⁵](#) by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model using self-supervised learning (i.e., generating the labels automatically from the data) on a huge text corpus, then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, they showed that by fine-tuning the pretrained model on just 100 labeled examples, they could achieve the same performance as a model trained from scratch on 10,000 examples.
- The [GPT paper²⁶](#) by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining, but this time using a

²⁴ Matthew Peters et al., “Deep Contextualized Word Representations,” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

²⁵ Jeremy Howard and Sebastian Ruder, “Universal Language Model Fine-Tuning for Text Classification,” *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

²⁶ Alec Radford et al., “Improving Language Understanding by Generative Pre-Training” (2018).

Transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A entails sentence B),²⁷ similarity (e.g., “Nice weather today” is very similar to “It is sunny”), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions). Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the [GPT-2 paper](#),²⁸ which proposed a very similar architecture, but larger still (with over 1.5 billion parameters!) and they showed that it could achieve good performance on many tasks without any fine-tuning. This is called *zero-shot learning* (ZSL). A smaller version of the GPT-2 model (with “just” 117 million parameters) is available at <https://github.com/openai/gpt-2>, along with its pretrained weights.

- The [BERT paper](#)²⁹ by Jacob Devlin and other Google researchers also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer’s encoder). This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model’s strength:

Masked language model (MLM)

Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is “She had fun at the birthday party,” then the model may be given the sentence “She <mask> fun at the <mask> party” and it must predict the words “had” and “birthday” (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

²⁷ For example, the sentence “Jane had a lot of fun at her friend’s birthday party” entails “Jane enjoyed the party,” but it is contradicted by “Everyone hated the party” and it is unrelated to “The Earth is flat.”

²⁸ Alec Radford et al., “Language Models Are Unsupervised Multitask Learners” (2019).

²⁹ Jacob Devlin et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that “The dog sleeps” and “It snores loudly” are consecutive sentences, while “The dog sleeps” and “The Earth orbits the Sun” are not consecutive. This is a challenging task, and it significantly improves the performance of the model when it is fine-tuned on tasks such as question answering or entailment.

As you can see, the main innovations in 2018 and 2019 have been better subword tokenization, shifting from LSTMs to Transformers, and pretraining universal language models using self-supervised learning, then fine-tuning them with very few architectural changes (or none at all). Things are moving fast; no one can say what architectures will prevail next year. Today, it’s clearly Transformers, but tomorrow it might be CNNs (e.g., check out the [2018 paper³⁰](#) by Maha Elbayad et al., where the researchers use masked 2D convolutional layers for sequence-to-sequence tasks). Or it might even be RNNs, if they make a surprise comeback (e.g., check out the [2018 paper³¹](#) by Shuai Li et al. that shows that by making neurons independent of each other in a given RNN layer, it is possible to train much deeper RNNs capable of learning much longer sequences).

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks (GANs) to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use Encoder–Decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?

³⁰ Maha Elbayad et al., “Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction,” arXiv preprint arXiv:1808.03867 (2018).

³¹ Shuai Li et al., “Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 5457–5466.

6. What is the most important layer in the Transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in [their paper](#) about LSTMs. They are artificial grammars that produce strings such as “BPBTSXXVPSEPE.” Check out Jenny Orr’s [nice introduction](#) to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr’s page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don’t.
9. Train an Encoder–Decoder model that can convert a date string from one format to another (e.g., from “April 22, 2019” to “2019-04-22”).
10. Go through TensorFlow’s [Neural Machine Translation with Attention](#) tutorial.
11. Use one of the recent language models (e.g., BERT) to generate more convincing Shakespearean text.

Solutions to these exercises are available in [Appendix A](#).

Representation Learning and Generative Learning Using Autoencoders and GANs

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see [Chapter 8](#)), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in [Chapter 11](#)). Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. However, the generated images are usually fuzzy and not entirely realistic.

In contrast, faces generated by generative adversarial networks (GANs) are now so convincing that it is hard to believe that the people they represent do not exist. You can judge so for yourself by visiting <https://thispersondoesnotexist.com/>, a website that shows faces generated by a recent GAN architecture called *StyleGAN* (you can also check out <https://thisrentaldoesnotexist.com/> to see some generated Airbnb bedrooms). GANs are now widely used for super resolution (increasing the resolution of an image), [colorization](#), powerful image editing (e.g., replacing photo bombers with realistic background), turning a simple sketch into a photorealistic image, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models and strengthening them, and more.

Autoencoders and GANs are both unsupervised, they both learn dense representations, they can both be used as generative models, and they have many similar applications. However, they work very differently:

- Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder learning the identity function under some constraints.
- GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in Deep Learning in that the generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake. *Adversarial training* (training competing neural networks) is widely considered as one of the most important ideas in recent years. In 2016, Yann LeCun even said that it was “the most interesting idea in the last 10 years in Machine Learning.”

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction, unsupervised pre-training, or as generative models. This will naturally lead us to GANs. We will start by building a simple GAN to generate fake images, but we will see that training is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. Let’s start with autoencoders!

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was [famously studied by William Chase and Herbert Simon in the early 1970s](#).¹ They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient latent representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or) that converts the inputs to a latent representation, followed by a *decoder* (or) that converts the internal representation to the outputs (see [Figure 17-1](#)).

¹ William G. Chase and Herbert A. Simon, “Perception in Chess,” *Cognitive Psychology* 4, no. 1 (1973): 55–81.

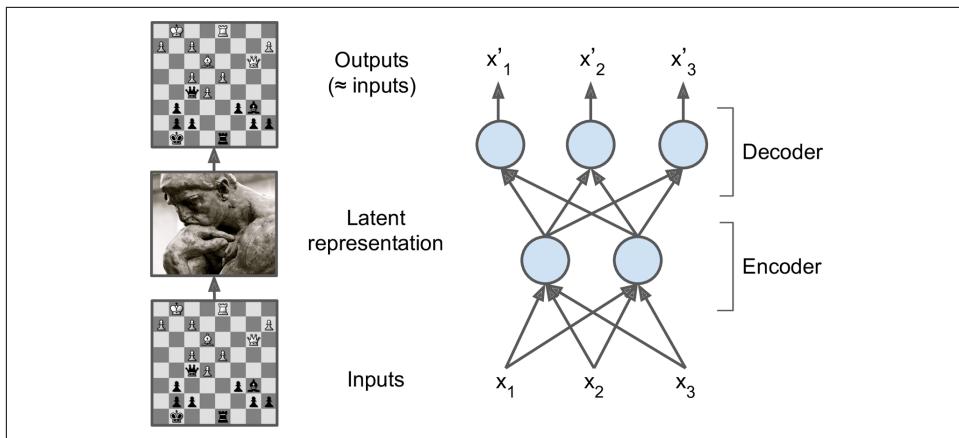


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

As you can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see [Chapter 10](#)), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing Principal Component Analysis (PCA; see [Chapter 8](#)).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D: