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 by default it 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 the `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 actually 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 custom model classes in Chapter 10 when we discussed the subclassing API.[10] It is actually quite straightforward, just subclass the `keras.models.Model` class, create layers and variables in the constructor, and implement the `call()` method to do whatever you want the model to do. For example, suppose you want to build the model represented in Figure 12-3:

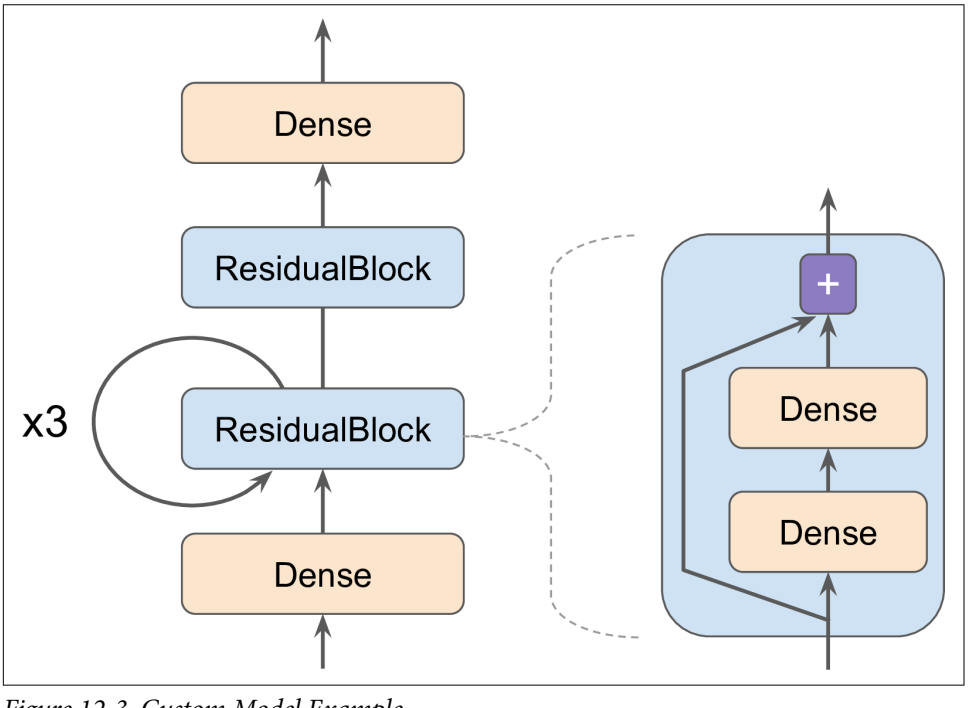

Figure 12-3. Custom Model Example

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 3 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 contain-

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.

ing 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 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.models.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 just save and load the weights using the `save_weights()` and `load_weights()` methods.

The `Model` class is actually a subclass of the `Layer` class, so models can be defined and used exactly like layers. But a model also has some extra functionalities, including of course its `compile()`, `fit()`, `evaluate()` and `predict()` methods (and a few var-

iants, such as `train_on_batch()` or `fit_generator()`), 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()`). So if models provide more functionalities than layers, why not just define every layer as a model? Well, technically you could, but it is probably cleaner to distinguish the internal components of your model (layers or reusable blocks of layers) from the model itself. The former should subclass the `Layer` class, while the latter should subclass the `Model` class.

With that, you can quite naturally and concisely build almost any model that you find in a paper, either using the sequential API, the functional API, the subclassing API, or even a mix of these. "Almost" any model? Yes, there are still a couple 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). However, you will occasionally 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, just compute it based on any part of the model you want, then pass the result to the `add_loss()` method. For example, the following custom model represents a standard MLP regressor with 5 hidden layers, except it also implements a *reconstruction loss* (see ???): we add an extra `Dense` layer on top of the last hidden layer, and its role is to try to reconstruct the inputs of the model. Since the reconstruction must have the same shape as the model's inputs, we need to create this `Dense` layer in the `build()` method to have access to the shape of the inputs. In the `call()` method, we compute both the regular output of the MLP, plus the output of the reconstruction layer. We then compute the mean squared difference between the reconstructions and the inputs, and we add this value (times 0.05) to the model's list of losses by calling `add_loss()`. During training, Keras will add this loss to the main loss (which is why we scaled down the reconstruction loss, to ensure the main loss dominates). As a result, the model will be forced 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:

```
class ReconstructingRegressor(keras.models.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)
```

Similarly, you can add a custom metric based on model internals by computing it in any way you want, as long at 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 the cases, everything we have discussed so far will be sufficient to implement whatever model you want to build, even with complex architectures, losses, metrics, and so on. However, in some rare cases you may need to customize the training loop itself. However, 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 ???) 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 regards to `w1` is `6 * w1 + 2 * w2`. You can also find that its partial derivative with regards to `w2` is `2 * w1`. For example, at the point `(w1, w2) = (5, 3)`, these par-

tial 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
10.000000003174137
```

Looks about right! This works rather well and it is trivial to implement, but it is just an approximation, and importantly you need to call `f()` at least once per parameter (not twice, since we could compute `f(w1, w2)` just once). This makes this approach intractable for large neural networks. So instead we should use autodiff (see Chapter 10 and ???). TensorFlow makes this pretty simple:

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

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

We first define two variables `w1` and `w2`, then we create a `tf.GradientTape` context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result `z` with regards to both variables `[w1, w2]`. 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!

Only put the strict minimum inside the `tf.GradientTape()` block, to save memory. Alternatively, you can 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 when you are done with it to free resources:

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

dz_dw1 = tape.gradient(z, w1) # => 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 regards to anything else 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 regards to these tensors, as if they were variables:

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

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

This can be useful in some cases, for example 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 regards to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.

If you compute the gradient of a list of tensors (e.g., `[z1, z2, z3]`) with regards to some variables (e.g., `[w1, w2]`), TensorFlow actually efficiently computes the sum of the gradients of these tensors (i.e., `gradient(z1, [w1, w2])`, plus `gradient(z2, [w1, w2])`, plus `gradient(z3, [w1, w2])`). Due to the way reverse-mode autodiff works, it is not possible to compute the individual gradients (`z1`, `z2` and `z3`) without actually calling `gradient()` multiple times (once for `z1`, once for `z2` and once for `z3`), which requires making the tape persistent (and deleting it afterwards).

Moreover, it is actually possible to compute second order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives)! To do this, we need to record the operations that are performed when computing the first-order partial derivatives (the Jacobians): this requires a second tape. Here is how it works:

```
with tf.GradientTape(persistent=True) as hessian_tape:
    with tf.GradientTape() as jacobian_tape:
        z = f(w1, w2)
    jacobians = jacobian_tape.gradient(z, [w1, w2])
hessians = [hessian_tape.gradient(jacobian, [w1, w2])
            for jacobian in jacobians]
del hessian_tape
```

The inner tape is used to compute the Jacobians, as we did earlier. The outer tape is used to compute the partial derivatives of each Jacobian. Since we need to call `gradient()` once for each Jacobian (or else we would get the sum of the partial derivatives over all the Jabobians, as explained earlier), we need the outer tape to be persistent, so we delete it at the end. The Jacobians are obviously the same as earlier (36 and 5), but now we also have the Hessians:

```
>>> hessians # dz_dw1_dw1, dz_dw1_dw2, dz_dw2_dw1, dz_dw2_dw2
[[<tf.Tensor: id=830578, shape=(), dtype=float32, numpy=6.0>,
  <tf.Tensor: id=830595, shape=(), dtype=float32, numpy=2.0>],
 [<tf.Tensor: id=830600, shape=(), dtype=float32, numpy=2.0>, None]]
```

Let's verify these Hessians. The first two are the partial derivatives of `6 * w1 + 2 * w2` (which is, as we saw earlier, the partial derivative of `f` with regards to `w1`), with regards to `w1` and `w2`. The result is correct: 6 for `w1` and 2 for `w2`. The next two are the partial derivatives of `2 * w1` (the partial derivative of `f` with regards to `w2`), with regards to `w1` and `w2`, which are 2 for `w1` and 0 for `w2`. Note that TensorFlow returns `None` instead of 0 since `w2` does not appear at all in `2 * w1`. TensorFlow also returns `None` when you use an operation whose gradients are not defined (e.g., `tf.argmax()`).

In some rare 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: it just returns its inputs during the forward pass (like `tf.identity()`), but it does not let gradients through during backpropagation (it acts like a constant). For example:

```
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 just 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), you can even compute Hessians, block backpropagation when needed and even 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 and Deep paper we discussed in Chapter 10 actually 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 your own custom training loops simply to feel more confident that it does precisely what you intent it 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 4 digits after the decimal point. Moreover, 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 also includes a progress bar, but you could use the handy tqdm library instead.