CHAPTER 12

# Custom Models and Training with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 12 in the final release of the book.

So far we have 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, learning rate schedules, and more. In fact, 95% of the use cases you will encounter will not require anything else 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, then 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 was released in March 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 (NLP), recommender systems, time series forecasting, and much more.

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

- Its core is very similar to NumPy, but with GPU support.
- It also 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 ???), and provides some excellent optimizers, such as RMSProp, Nadam and FTRL (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 & 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 also includes another Deep Learning API called the *Estimators API*, but it is now recommended to use tf.keras instead.

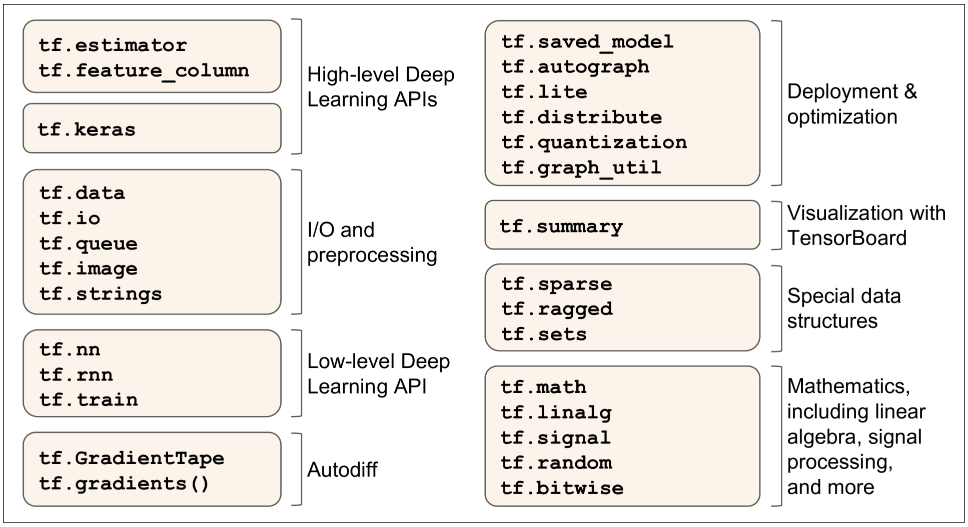


*Figure 12-1. TensorFlow's Python API*

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

At the lowest level, each TensorFlow operation is implemented using highly efficient C++ code[2]. Many operations (or *ops* for short) 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 computations into many smaller chunks and running them in parallel across many GPU threads. TPUs are even faster. You can purchase your own GPU devices (for now, TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+), but TPUs are only available on *Google Cloud Machine Learning Engine* (see ???).[3]

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 APIs for other languages are also available. In any case, TensorFlow's execution

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

3 If you are a researcher, you may be eligible to use these TPUs for free, see *https://tensorflow.org/tfrc/* for more details.

engine will take care of running the operations efficiently, even across multiple devices and machines if you tell it to.

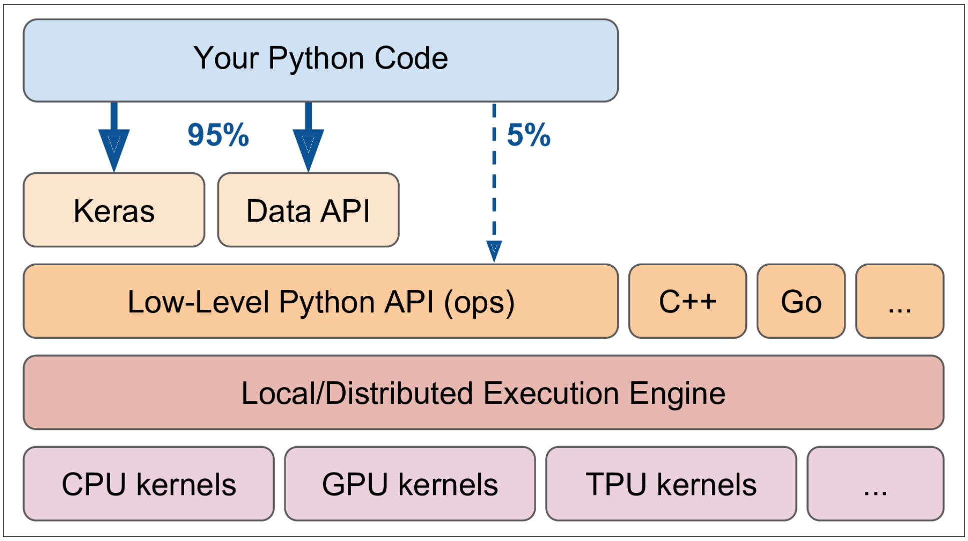


*Figure 12-2. TensorFlow's architecture*

TensorFlow runs not only on Windows, Linux, and MacOS, but also on mobile devices (using *TensorFlow Lite*), including both iOS and Android (see ???). If you do not want to use the Python API, there are also 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 just 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 ???). Google also launched *TensorFlow Hub*, 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, or *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 implementation, 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, and 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.

Okay, it's time to start coding!

## Using TensorFlow like NumPy

TensorFlow's API revolves around *tensors*, hence the name Tensor-Flow. A tensor is usually a multidimensional array (exactly like a NumPy `ndarray`), 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 easily create a tensor, using `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 `-`, `*`, etc.) 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 (e.g., `tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`…), and more generally most operations that you can find in NumPy (e.g., `tf.reshape()`, `tf.squeeze()`, `tf.tile()`), but sometimes with a different name (e.g., `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()`, `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 it does not do exactly the same thing: 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, this means that 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 operations[4], while preserving well organized packages.

## Keras' Low-Level API

The Keras API actually has its own low-level API, located in `keras.backend`. It includes functions like `square()`, `exp()`, `sqrt()` and so on. 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 as 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, and 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)
```

4 A notable exception is `tf.math.log()` which is commonly used but there is no `tf.log()` alias (as it might be confused with logging).

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

So far, we have used constant tensors: as their name suggests, you cannot modify them. However, the weights in a neural network need to be tweaked by backpropagation, and 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 constant 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), using the cell's (or slice's) `assign()` method (direct item assignment will not work), or 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 notebook or ??? for more details):

- *Sparse tensors* (`tf.SparseTensor`) efficiently represent tensors containing mostly 0s. 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 actually represent byte strings, not Unicode strings, so if you create a string tensor using a Unicode string (e.g., a regular Python 3 string like `"café"`\`), then it will get encoded to UTF-8 automatically (e.g., `b"caf\xc3\xa9"`). Alternatively, you can represent Unicode strings using tensors of type `tf.int32`, where each item represents a Unicode codepoint (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).
- *Sets* are just represented as regular tensors (or sparse tensors) containing one or more sets, and you can manipulate them using operations from the `tf.sets` package.
- *Queues*, including First In, First Out (FIFO) queues (`FIFOQueue`), queues that can prioritize some items (`PriorityQueue`), queues that shuffle their items (`RandomShuffleQueue`), and queues that can 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 it 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, so your model will end up being 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).

Next, you can just 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 we save the model?

## Saving and Loading Models That Contain Custom Components

Saving a model containing a custom loss function actually works fine, as Keras just saves the name of the function. However, whenever you load it, you 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 we 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 we 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 implement 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 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.

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, possibly weighted by the sample weights, if any, and then divide the result 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")
```

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 just creates an instance of the class, passing `**config` to the constructor.

That's it for losses! It was not too hard, was it? Well it's just as simple for custom activation functions, initializers, regularizers, and constraints. Let's look at these now.

5 It would not be a good idea to use a weighted mean: if we 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.

## 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. For example, 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 $\ell_1$ 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 some 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). For example, much like we did for the custom loss, here is a simple class for $\ell_1$ 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 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 okay if they are not easily interpretable by humans (e.g. cross-entropy). In contrast, metrics are used to *evaluate* a model, they must be more easily interpretable, and they can be non-differentiable or have 0 gradients everywhere (e.g., accuracy).

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"`):

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

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch. Most of the time, this is exactly what you want. But not always! Consider a binary classifier's precision, for example. As 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 5 positive predictions in the first batch, 4 of which were correct: that's 80% precision. Then suppose the model made 3 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, this is *not* the model's precision over these two batches! Indeed, there were a total of 4 true positives (4 + 0) out of 8 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 num-

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

ber of false positives, and 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 the 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) using the `variables` attribute, and reset these variables using the `reset_states()` method:

```
>>> p.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> p.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> p.reset_states() # both variables get reset to 0.0
```

If you need to create such a streaming metric, you can just 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 just ignore them).
- The `result()` method computes and returns the final result, in this case just 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 just resets all variables to 0.0 (but you can override it if needed).

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 thoses 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!

7 This class is for illustration purposes only. A simpler and better implementation would just subclass the `keras.metrics.Mean` class, see the notebook for an example.

## 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 sometimes 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, and 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:

```python
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 just use `activation=tf.exp`, or `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., 1000.).

As you 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:

```python
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 just `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`, `name`, and so on. 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 `build()` method: this tells Keras that the layer is built (it just sets `self.built = True`).
- The `call()` method actually 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, 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 earlier. 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!

8 This function is specific to tf.keras. You could use `keras.activations.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()`.

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:

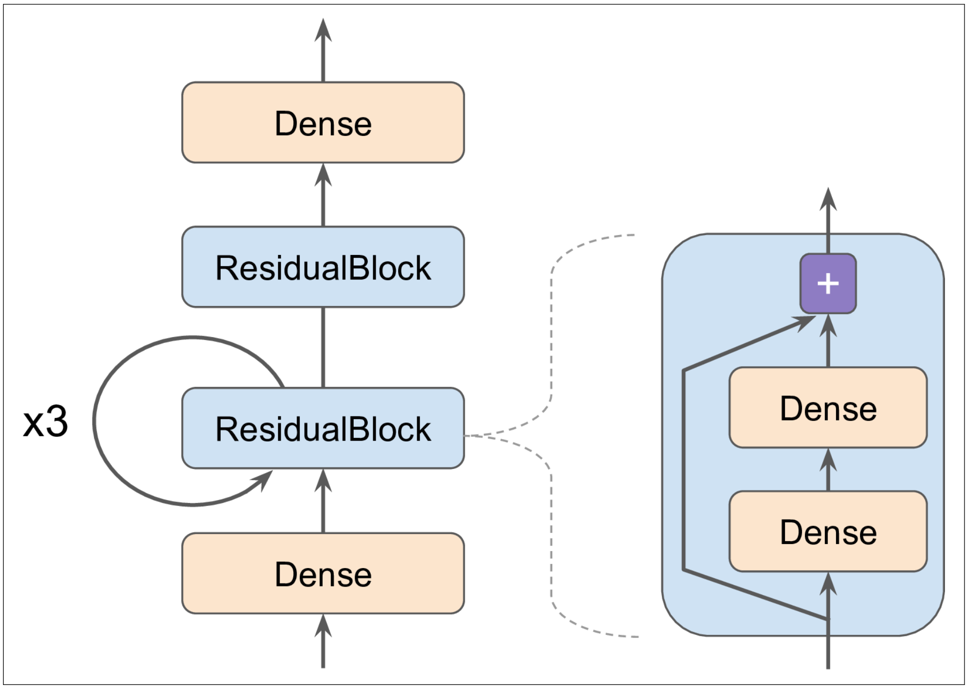


*Figure 12-3. Custom Model Example*

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 variants, 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:

```python
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 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
10.000000003174137
```

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.

With that, let's get down to business! First, we need to define some hyperparameters, 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 regards to each trainable variable (*not* all variables!), and we apply them to the optimizer to perform a Gradient Descent step.
- Next 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 complete[11] 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 hyperparameters, 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.[12]

As you can see, there are quite a lot of things you need to get right, it is 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 models[13] 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 ???).

## TensorFlow Functions and Graphs

In TensorFlow 1, graphs were unavoidable (as were the complexities that came with them): they were a central part of TensorFlow's API. In TensorFlow 2, they are still

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

12 Alternatively, check out K.learning_phase(), K.set_learning_phase() and K.learning_phase_scope().

13 With the exception of optimizers, as very few people ever customize these: see the notebook for an example.

there, but not as central, and much (much!) simpler to use. To demonstrate this, let's start with a trivial function that just computes the cube of its input:

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

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

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: 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.[14] Most of the time you will not really need to know more than that: when you want to boost a Python function, just transform it into a TF Function. That's all!

Moreover, 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.

TF Function generates a new graph for every unique set of input shapes and data types, and it 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.

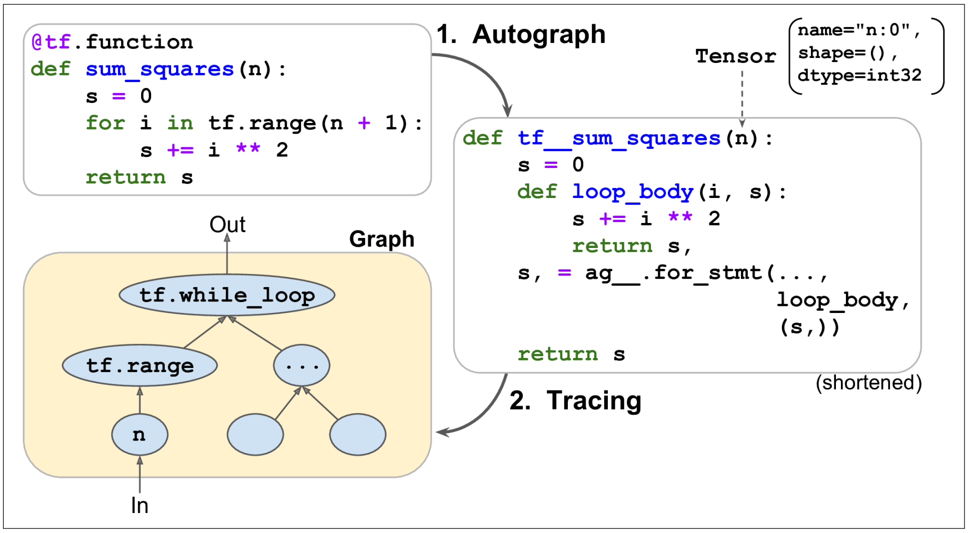
If you call a TF Function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM. Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per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

## Autograph and Tracing

So how does TensorFlow generate graphs? Well, first it starts by analyzing the Python function's source code to capture all the control flow statements, such as `for` loops and `while` loops, `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__()` or `__mul__()` to capture operators like

14 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()`.

`+` 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.



```python
@tf.function
def sum_squares(n):
    s = 0
    for i in tf.range(n + 1):
        s += i ** 2
    return s
```

```python
def tf__sum_squares(n):
    s = 0
    def loop_body(i, s):
        s += i ** 2
        return s,
    s, = ag__.for_stmt(...,
                       loop_body,
                       (s,))
    return s
```

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

Next, TensorFlow calls this "upgraded" function, but instead of passing the actual argument, it passes a *symbolic tensor*, meaning 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 actually be called with a symbolic tensor of type int32 and shape []. The function will run in *graph mode*, meaning that each TensorFlow operation will just 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 actual 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 argument (in this case, an int32 tensor of shape []), and the final graph generated during tracing. The ellipses 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: just 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()`, and `tf.sort()` instead of the built-in `sorted()` function, and so on (unless you really want the code to run only during tracing).
  - For example, 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 that side-effect to occur every time you call the TF Function, as it will only occur when the function is traced.
  - You can wrap arbitrary Python code in a `tf.py_function()` operation, but this will hinder performance, as TensorFlow will not be able to do any graph optimization on this code, and it will also reduce portability, as the graph will only run on platforms where Python is available (and the right libraries installed).
- You can call other Python functions or TF Functions, but they should follow the same rules, as TensorFlow will also 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).

- 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 Python files `*.pyc` 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(10)` rather than `for i in range(10)`, 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.
- And 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 further technical details in ???).

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