4. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What are about MC dropout?
8. Deep Learning.
    a. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
    b. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
    c. Tune the hyperparameters using cross-validation and see what precision you can achieve.
    d. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
    e. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?
9. Transfer learning.
    a. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.
    b. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
    c. Try caching the frozen layers, and train the model again: how much faster is it now?
    d. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
    e. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?
10. Pretraining on an auxiliary task.
    a. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little

training data. Start by building two DNNs (let's call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use a `keras.layers.Concatenate` layer to concatenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.

b. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.

c. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.

d. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in ???.

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.

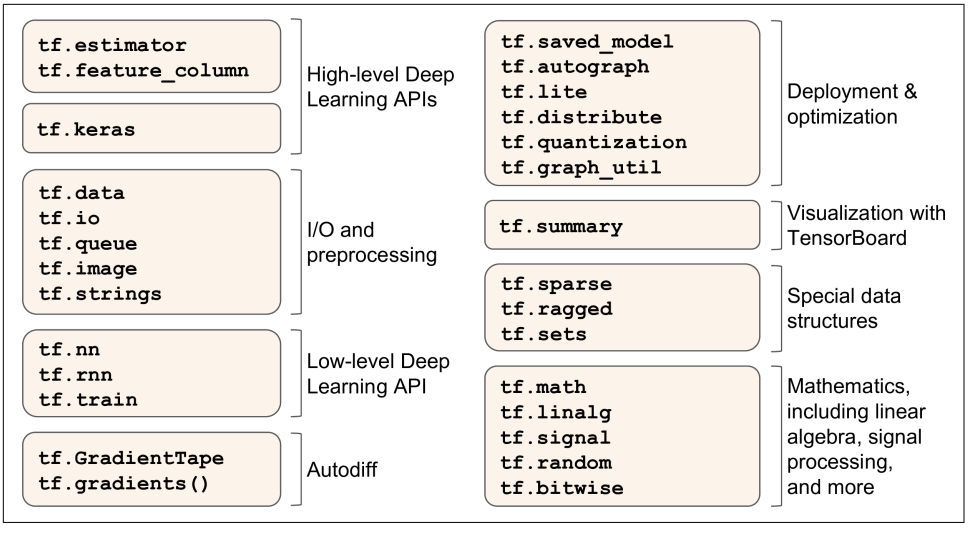

Figure 12-1. TensorFlow's Python API

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

At the lowest level, each TensorFlow operation 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.

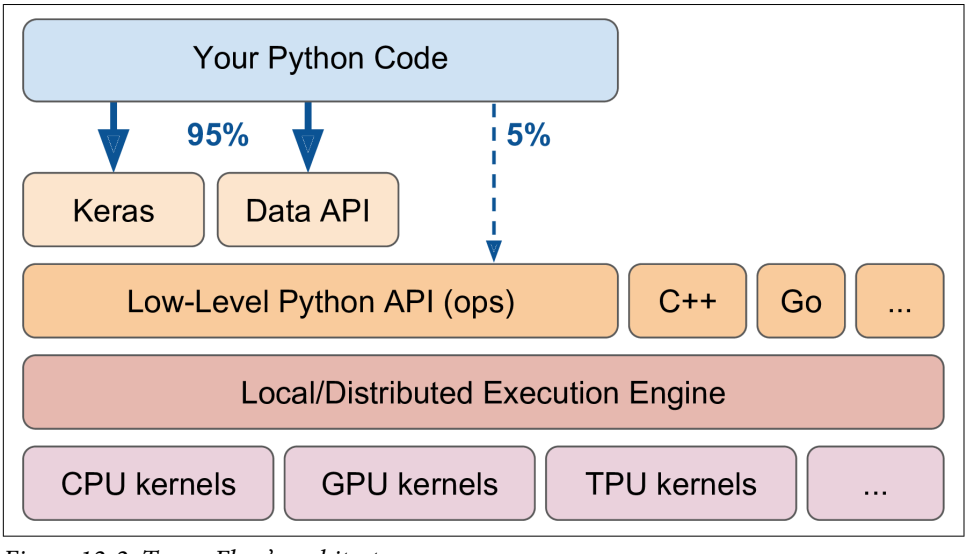

Figure 12-2. TensorFlow's architecture

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.]]
```