Table 9-1. Open source Deep Learning libraries (not an exhaustive list)

Library	API	Platforms	Started by	Year
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	A. Gibson, J.Patterson	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014
MXNet	Python, C++, others	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	University of Montreal	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

Installation

Let's get started! Assuming you installed Jupyter and Scikit-Learn by following the installation instructions in Chapter 2, you can simply use pip to install TensorFlow. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH               # Your ML working directory (e.g., $HOME/ml)
$ source env/bin/activate
```

Next, install TensorFlow:

```
$ pip3 install --upgrade tensorflow
```

For GPU support, you need to install `tensorflow-gpu` instead of `tensorflow`. See Chapter 12 for more details.

To test your installation, type the following command. It should output the version of TensorFlow you installed.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)'
1.0.0
```

Creating Your First Graph and Running It in a Session

The following code creates the graph represented in Figure 9-1:

```
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

That's all there is to it! The most important thing to understand is that this code does not actually perform any computation, even though it looks like it does (especially the last line). It just creates a computation graph. In fact, even the variables are not initialized yet. To evaluate this graph, you need to open a TensorFlow *session* and use it to initialize the variables and evaluate `f`. A TensorFlow session takes care of placing the operations onto *devices* such as CPUs and GPUs and running them, and it holds all the variable values.[3] The following code creates a session, initializes the variables, and evaluates, and `f` then closes the session (which frees up resources):

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Having to repeat `sess.run()` all the time is a bit cumbersome, but fortunately there is a better way:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Inside the `with` block, the session is set as the default session. Calling `x.initializer.run()` is equivalent to calling `tf.get_default_session().run(x.initializer)`, and similarly `f.eval()` is equivalent to calling `tf.get_default_session().run(f)`. This makes the code easier to read. Moreover, the session is automatically closed at the end of the block.

Instead of manually running the initializer for every single variable, you can use the `global_variables_initializer()` function. Note that it does not actually perform the initialization immediately, but rather creates a node in the graph that will initialize all variables when it is run:

```
init = tf.global_variables_initializer()  # prepare an init node

with tf.Session() as sess:
    init.run()  # actually initialize all the variables
    result = f.eval()
```

Inside Jupyter or within a Python shell you may prefer to create an `InteractiveSession`. The only difference from a regular `Session` is that when an `InteractiveSession` is created it automatically sets itself as the default session, so you don't need a

3 In distributed TensorFlow, variable values are stored on the servers instead of the session, as we will see in Chapter 12.

with block (but you do need to close the session manually when you are done with it):

```
>>> sess = tf.InteractiveSession()
>>> init.run()
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()
```

A TensorFlow program is typically split into two parts: the first part builds a computation graph (this is called the *construction phase*), and the second part runs it (this is the *execution phase*). The construction phase typically builds a computation graph representing the ML model and the computations required to train it. The execution phase generally runs a loop that evaluates a training step repeatedly (for example, one step per mini-batch), gradually improving the model parameters. We will go through an example shortly.

Managing Graphs

Any node you create is automatically added to the default graph:

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

In most cases this is fine, but sometimes you may want to manage multiple independent graphs. You can do this by creating a new `Graph` and temporarily making it the default graph inside a `with` block, like so:

```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```

In Jupyter (or in a Python shell), it is common to run the same commands more than once while you are experimenting. As a result, you may end up with a default graph containing many duplicate nodes. One solution is to restart the Jupyter kernel (or the Python shell), but a more convenient solution is to just reset the default graph by running `tf.reset_default_graph()`.