With that you should be able to train all sorts of RNNs! Unfortunately, if you want to train an RNN on long sequences, things will get a bit harder. Let's see why and what you can do about it.

The Difficulty of Training over Many Time Steps

To train an RNN on long sequences, you will need to run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the vanishing/exploding gradients problem (discussed in Chapter 11) and take forever to train. Many of the tricks we discussed to alleviate this problem can be used for deep unrolled RNNs as well: good parameter initialization, nonsaturating activation functions (e.g., ReLU), Batch Normalization, Gradient Clipping, and faster optimizers. However, if the RNN needs to handle even moderately long sequences (e.g., 100 inputs), then training will still be very slow.

The simplest and most common solution to this problem is to unroll the RNN only over a limited number of time steps during training. This is called *truncated backpropagation through time*. In TensorFlow you can implement it simply by truncating the input sequences. For example, in the time series prediction problem, you would simply reduce `n_steps` during training. The problem, of course, is that the model will not be able to learn long-term patterns. One workaround could be to make sure that these shortened sequences contain both old and recent data, so that the model can learn to use both (e.g., the sequence could contain monthly data for the last five months, then weekly data for the last five weeks, then daily data over the last five days). But this workaround has its limits: what if fine-grained data from last year is actually useful? What if there was a brief but significant event that absolutely must be taken into account, even years later (e.g., the result of an election)?

Besides the long training time, a second problem faced by long-running RNNs is the fact that the memory of the first inputs gradually fades away. Indeed, due to the transformations that the data goes through when traversing an RNN, some information is lost after each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. For example, say you want to perform sentiment analysis on a long review that starts with the four words "I loved this movie," but the rest of the review lists the many things that could have made the movie even better. If the RNN gradually forgets the first four words, it will completely misinterpret the review. To solve this problem, various types of cells with long-term memory have been introduced. They have proved so successful that the basic cells are not much used anymore. Let's first look at the most popular of these long memory cells: the LSTM cell.

LSTM Cell

The *Long Short-Term Memory* (LSTM) cell was proposed in 1997[3] by Sepp Hochreiter and Jürgen Schmidhuber, and it was gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,[4] Wojciech Zaremba,[5] and many more. If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster and it will detect long-term dependencies in the data. In TensorFlow, you can simply use a `BasicLSTMCell` instead of a `BasicRNNCell`:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

LSTM cells manage two state vectors, and for performance reasons they are kept separate by default. You can change this default behavior by setting `state_is_tuple=False` when creating the `BasicLSTMCell`.

So how does an LSTM cell work? The architecture of a basic LSTM cell is shown in Figure 14-13.

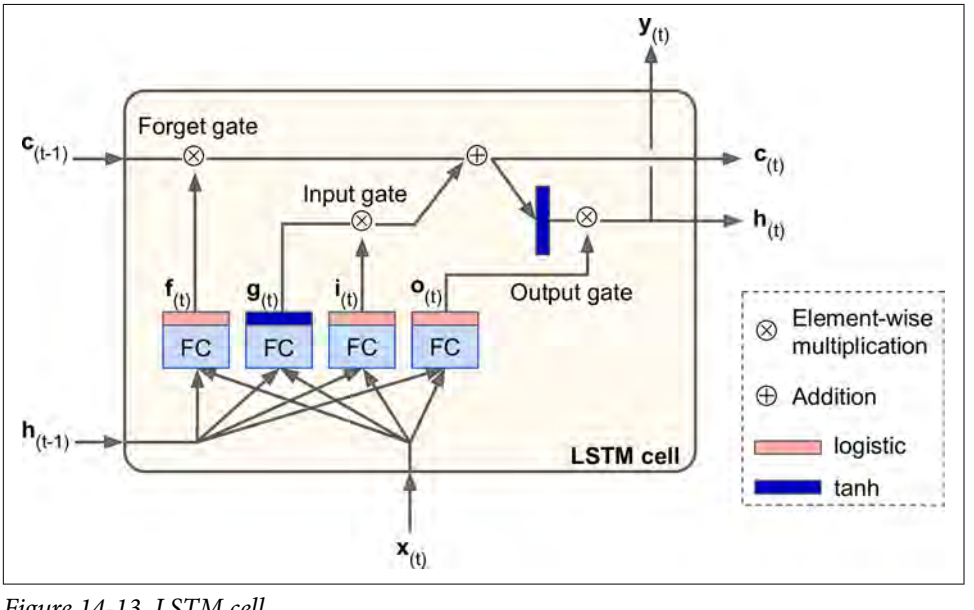

Figure 14-13. LSTM cell

3 "Long Short-Term Memory," S. Hochreiter and J. Schmidhuber (1997).

4 "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling," H. Sak et al. (2014).

5 "Recurrent Neural Network Regularization," W. Zaremba et al. (2015).