Applying Dropout

If you build a very deep RNN, it may end up overfitting the training set. To prevent that, a common technique is to apply dropout (introduced in Chapter 11). You can simply add a dropout layer before or after the RNN as usual, but if you also want to apply dropout between the RNN layers, you need to use a `DropoutWrapper`. The following code applies dropout to the inputs of each layer in the RNN, dropping each input with a 50% probability:

```
keep_prob = 0.5

cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
cell_drop = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell_drop] * n_layers)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

Note that it is also possible to apply dropout to the outputs by setting `output_keep_prob`.

The main problem with this code is that it will apply dropout not only during training but also during testing, which is not what you want (recall that dropout should be applied only during training). Unfortunately, the `DropoutWrapper` does not support an `is_training` placeholder (yet?), so you must either write your own dropout wrapper class, or have two different graphs: one for training, and the other for testing. The second option looks like this:

```
import sys
is_training = (sys.argv[-1] == "train")

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
if is_training:
    cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell] * n_layers)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
[...] # build the rest of the graph
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    if is_training:
        init.run()
        for iteration in range(n_iterations):
            [...] # train the model
        save_path = saver.save(sess, "/tmp/my_model.ckpt")
    else:
        saver.restore(sess, "/tmp/my_model.ckpt")
        [...] # use the model
```

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.