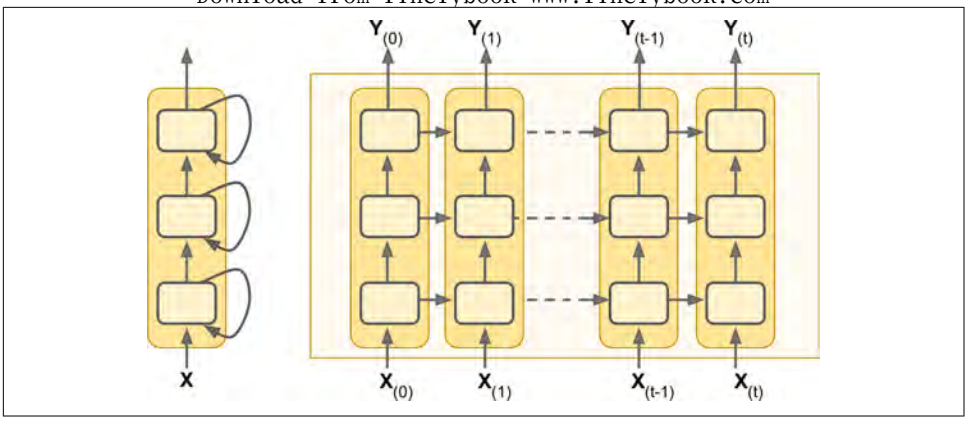

Figure 14-12. Deep RNN (left), unrolled through time (right)

To implement a deep RNN in TensorFlow, you can create several cells and stack them into a `MultiRNNCell`. In the following code we stack three identical cells (but you could very well use various kinds of cells with a different number of neurons):

```
n_neurons = 100
n_layers = 3

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] * n_layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

That's all there is to it! The `states` variable is a tuple containing one tensor per layer, each representing the final state of that layer's cell (with shape `[batch_size, n_neurons]`). If you set `state_is_tuple=False` when creating the `MultiRNNCell`, then `states` becomes a single tensor containing the states from every layer, concatenated along the column axis (i.e., its shape is `[batch_size, n_layers * n_neurons]`). Note that before TensorFlow 0.11.0, this behavior was the default.

Distributing a Deep RNN Across Multiple GPUs

Chapter 12 pointed out that we can efficiently distribute deep RNNs across multiple GPUs by pinning each layer to a different GPU (see Figure 12-16). However, if you try to create each cell in a different `device()` block, it will not work:

```
with tf.device("/gpu:0"):  # BAD! This is ignored.
    layer1 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)

with tf.device("/gpu:1"):  # BAD! Ignored again.
    layer2 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

This fails because a `BasicRNNCell` is a cell factory, not a cell *per se* (as mentioned earlier); no cells get created when you create the factory, and thus no variables do either.

The device block is simply ignored. The cells actually get created later. When you call `dynamic_rnn()`, it calls the `MultiRNNCell`, which calls each individual `BasicRNNCell`, which create the actual cells (including their variables). Unfortunately, none of these classes provide any way to control the devices on which the variables get created. If you try to put the `dynamic_rnn()` call within a device block, the whole RNN gets pinned to a single device. So are you stuck? Fortunately not! The trick is to create your own cell wrapper:

```
import tensorflow as tf

class DeviceCellWrapper(tf.contrib.rnn.RNNCell):
  def __init__(self, device, cell):
    self._cell = cell
    self._device = device

  @property
  def state_size(self):
    return self._cell.state_size

  @property
  def output_size(self):
    return self._cell.output_size

  def __call__(self, inputs, state, scope=None):
    with tf.device(self._device):
        return self._cell(inputs, state, scope)
```

This wrapper simply proxies every method call to another cell, except it wraps the `__call__()` function within a device block.[2] Now you can distribute each layer on a different GPU:

```
devices = ["/gpu:0", "/gpu:1", "/gpu:2"]
cells = [DeviceCellWrapper(dev,tf.contrib.rnn.BasicRNNCell(num_units=n_neurons))
         for dev in devices]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

Do not set `state_is_tuple=False`, or the `MultiRNNCell` will concatenate all the cell states into a single tensor, on a single GPU.

2 This uses the *decorator* design pattern.

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
```