Basic RNNs in TensorFlow

First, let's implement a very simple RNN model, without using any of TensorFlow's RNN operations, to better understand what goes on under the hood. We will create an RNN composed of a layer of five recurrent neurons (like the RNN represented in Figure 14-2), using the tanh activation function. We will assume that the RNN runs over only two time steps, taking input vectors of size 3 at each time step. The following code builds this RNN, unrolled through two time steps:

```
n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

This network looks much like a two-layer feedforward neural network, with a few twists: first, the same weights and bias terms are shared by both layers, and second, we feed inputs at each layer, and we get outputs from each layer. To run the model, we need to feed it the inputs at both time steps, like so:

```
import numpy as np

# Mini-batch:        instance 0,instance 1,instance 2,instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

This mini-batch contains four instances, each with an input sequence composed of exactly two inputs. At the end, `Y0_val` and `Y1_val` contain the outputs of the network at both time steps for all neurons and all instances in the mini-batch:

```
>>> print(Y0_val)  # output at t = 0
[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]  # instance 0
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]  # instance 1
 [ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]  # instance 2
 [ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]] # instance 3
>>> print(Y1_val)  # output at t = 1
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]  # instance 0
 [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]  # instance 1
```

```
[-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]  # instance 2
[-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]] # instance 3
```

That wasn't too hard, but of course if you want to be able to run an RNN over 100 time steps, the graph is going to be pretty big. Now let's look at how to create the same model using TensorFlow's RNN operations.

Static Unrolling Through Time

The `static_rnn()` function creates an unrolled RNN network by chaining cells. The following code creates the exact same model as the previous one:

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(
                          basic_cell, [X0, X1], dtype=tf.float32)
Y0, Y1 = output_seqs
```

First we create the input placeholders, as before. Then we create a `BasicRNNCell`, which you can think of as a factory that creates copies of the cell to build the unrolled RNN (one for each time step). Then we call `static_rnn()`, giving it the cell factory and the input tensors, and telling it the data type of the inputs (this is used to create the initial state matrix, which by default is full of zeros). The `static_rnn()` function calls the cell factory's `__call__()` function once per input, creating two copies of the cell (each containing a layer of five recurrent neurons), with shared weights and bias terms, and it chains them just like we did earlier. The `static_rnn()` function returns two objects. The first is a Python list containing the output tensors for each time step. The second is a tensor containing the final states of the network. When you are using basic cells, the final state is simply equal to the last output.

If there were 50 time steps, it would not be very convenient to have to define 50 input placeholders and 50 output tensors. Moreover, at execution time you would have to feed each of the 50 placeholders and manipulate the 50 outputs. Let's simplify this. The following code builds the same RNN again, but this time it takes a single input placeholder of shape `[None, n_steps, n_inputs]` where the first dimension is the mini-batch size. Then it extracts the list of input sequences for each time step. `X_seqs` is a Python list of `n_steps` tensors of shape `[None, n_inputs]`, where once again the first dimension is the mini-batch size. To do this, we first swap the first two dimensions using the `transpose()` function, so that the time steps are now the first dimension. Then we extract a Python list of tensors along the first dimension (i.e., one tensor per time step) using the `unstack()` function. The next two lines are the same as before. Finally, we merge all the output tensors into a single tensor using the `stack()` function, and we swap the first two dimensions to get a final `outputs` tensor