```
    [-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 of shape [None, n_steps, n_neurons] (again the first dimension is the mini-batch size).

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(
                        basic_cell, X_seqs, dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

Now we can run the network by feeding it a single tensor that contains all the mini-batch sequences:

```
X_batch = np.array([
         # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 0
        [[3, 4, 5], [0, 0, 0]], # instance 1
        [[6, 7, 8], [6, 5, 4]], # instance 2
        [[9, 0, 1], [3, 2, 1]], # instance 3
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

And we get a single outputs_val tensor for all instances, all time steps, and all neurons:

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]
  [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]]

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]]
```

However, this approach still builds a graph containing one cell per time step. If there were 50 time steps, the graph would look pretty ugly. It is a bit like writing a program without ever using loops (e.g., Y0=f(0, X0); Y1=f(Y0, X1); Y2=f(Y1, X2); ...; Y50=f(Y49, X50)). With such as large graph, you may even get out-of-memory (OOM) errors during backpropagation (especially with the limited memory of GPU cards), since it must store all tensor values during the forward pass so it can use them to compute gradients during the reverse pass.

Fortunately, there is a better solution: the dynamic_rnn() function.

Dynamic Unrolling Through Time

The dynamic_rnn() function uses a while_loop() operation to run over the cell the appropriate number of times, and you can set swap_memory=True if you want it to swap the GPU's memory to the CPU's memory during backpropagation to avoid OOM errors. Conveniently, it also accepts a single tensor for all inputs at every time step (shape [None, n_steps, n_inputs]) and it outputs a single tensor for all outputs at every time step (shape [None, n_steps, n_neurons]); there is no need to stack, unstack, or transpose. The following code creates the same RNN as earlier using the dynamic_rnn() function. It's so much nicer!

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

During backpropagation, the while_loop() operation does the appropriate magic: it stores the tensor values for each iteration during the forward pass so it can use them to compute gradients during the reverse pass.

Handling Variable Length Input Sequences

So far we have used only fixed-size input sequences (all exactly two steps long). What if the input sequences have variable lengths (e.g., like sentences)? In this case you should set the sequence_length parameter when calling the dynamic_rnn() (or static_rnn()) function; it must be a 1D tensor indicating the length of the input sequence for each instance. For example:

```
seq_length = tf.placeholder(tf.int32, [None])

[...]
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                    sequence_length=seq_length)
```

For example, suppose the second input sequence contains only one input instead of two. It must be padded with a zero vector in order to fit in the input tensor X (because the input tensor's second dimension is the size of the longest sequence—i.e., 2).

```
X_batch = np.array([
        # step 0     step 1
        [[0, 1, 2], [9, 8, 7]], # instance 0
        [[3, 4, 5], [0, 0, 0]], # instance 1 (padded with a zero vector)
        [[6, 7, 8], [6, 5, 4]], # instance 2
        [[9, 0, 1], [3, 2, 1]], # instance 3
    ])
seq_length_batch = np.array([2, 1, 2, 2])
```