## 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])
```

Of course, you now need to feed values for both placeholders X and seq_length:

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Now the RNN outputs zero vectors for every time step past the input sequence length (look at the second instance's output for the second time step):

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

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]   # final state
  [ 0.          0.          0.          0.          0.        ]]  # zero vector

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

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

Moreover, the states tensor contains the final state of each cell (excluding the zero vectors):

```
>>> print(states_val)
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]    # t = 1
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]    # t = 0 !!!
 [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]    # t = 1
 [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]   # t = 1
```

## Handling Variable-Length Output Sequences

What if the output sequences have variable lengths as well? If you know in advance what length each sequence will have (for example if you know that it will be the same length as the input sequence), then you can set the sequence_length parameter as described above. Unfortunately, in general this will not be possible: for example, the length of a translated sentence is generally different from the length of the input sentence. In this case, the most common solution is to define a special output called an *end-of-sequence token* (EOS token). Any output past the EOS should be ignored (we will discuss this later in this chapter).

Okay, now you know how to build an RNN network (or more precisely an RNN network unrolled through time). But how do you train it?