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?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then simply use regular backpropagation (see Figure 14-5). This strategy is called *backpropagation through time* (BPTT).

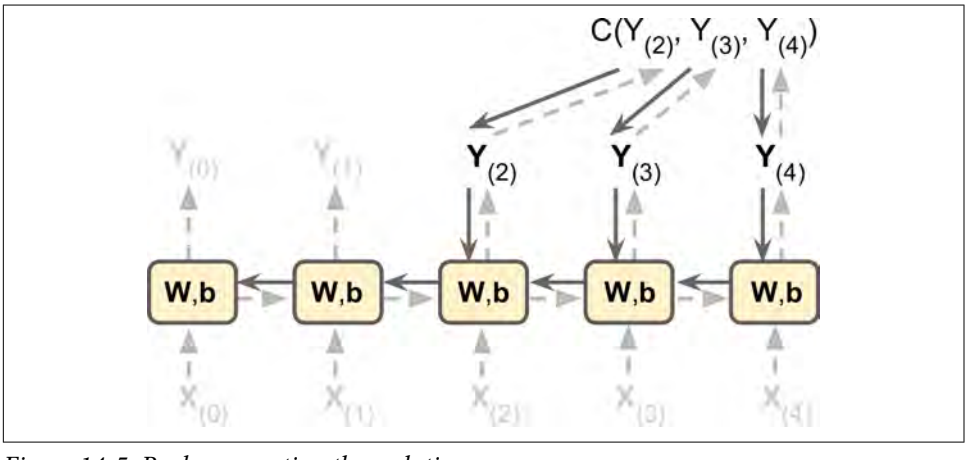

Figure 14-5. Backpropagation through time

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows); then the output sequence is evaluated using a cost function $C\left(\mathbf{Y}_{(t_{\text{min}})}, \mathbf{Y}_{(t_{\text{min}}+1)}, \cdots, \mathbf{Y}_{(t_{\text{max}})}\right)$ (where t_{min} and t_{max} are the first and last output time steps, not counting the ignored outputs), and the gradients of that cost function are propagated backward through the unrolled network (represented by the solid arrows); and finally the model parameters are updated using the gradients computed during BPTT. Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output (for example, in Figure 14-5 the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs, but not through $\mathbf{Y}_{(0)}$ and $\mathbf{Y}_{(1)}$). Moreover, since the same parameters $\mathbf{W}$ and $\mathbf{b}$ are used at each time step, backpropagation will do the right thing and sum over all time steps.

Training a Sequence Classifier

Let's train an RNN to classify MNIST images. A convolutional neural network would be better suited for image classification (see Chapter 13), but this makes for a simple example that you are already familiar with. We will treat each image as a sequence of 28 rows of 28 pixels each (since each MNIST image is 28 × 28 pixels). We will use cells of 150 recurrent neurons, plus a fully connected layer containing 10 neurons