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).

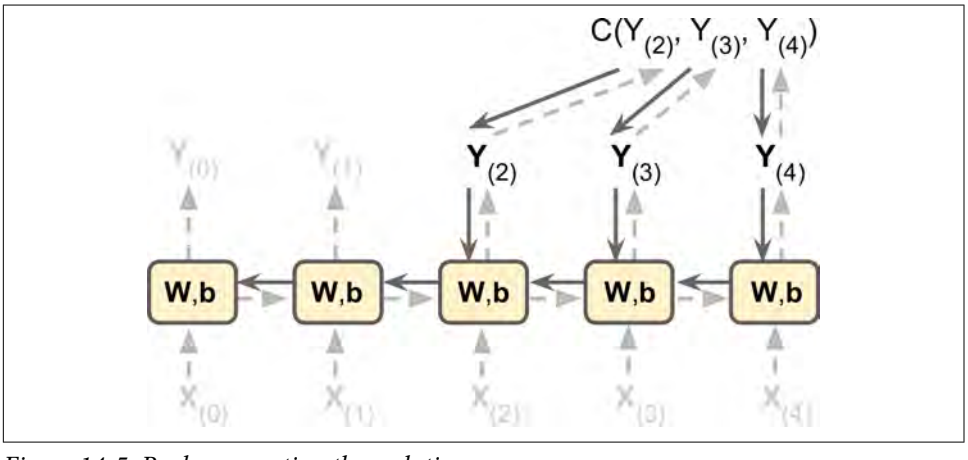

Figure 14-5. Backpropagation through time

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

(one per class) connected to the output of the last time step, followed by a softmax layer (see Figure 14-6).

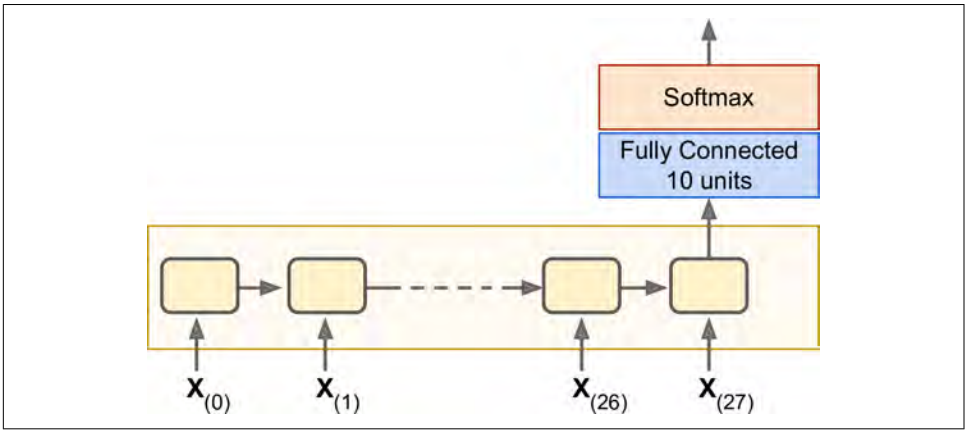

Figure 14-6. Sequence classifier

The construction phase is quite straightforward; it's pretty much the same as the MNIST classifier we built in Chapter 10 except that an unrolled RNN replaces the hidden layers. Note that the fully connected layer is connected to the `states` tensor, which contains only the final state of the RNN (i.e., the 28th output). Also note that `y` is a placeholder for the target classes.

```
from tensorflow.contrib.layers import fully_connected

n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

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

logits = fully_connected(states, n_outputs, activation_fn=None)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
              labels=y, logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

```
init = tf.global_variables_initializer()
```

Now let's load the MNIST data and reshape the test data to `[batch_size, n_steps, n_inputs]` as is expected by the network. We will take care of reshaping the training data in a moment.

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels
```

Now we are ready to train the RNN. The execution phase is exactly the same as for the MNIST classifier in Chapter 10, except that we reshape each training batch before feeding it to the network.

```
n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

The output should look like this:

```
0 Train accuracy: 0.713333 Test accuracy: 0.7299
1 Train accuracy: 0.766667 Test accuracy: 0.7977
...
98 Train accuracy: 0.986667 Test accuracy: 0.9777
99 Train accuracy: 0.986667 Test accuracy: 0.9809
```

We get over 98% accuracy—not bad! Plus you would certainly get a better result by tuning the hyperparameters, initializing the RNN weights using He initialization, training longer, or adding a bit of regularization (e.g., dropout).

You can specify an initializer for the RNN by wrapping its construction code in a variable scope (e.g., use `variable_scope("rnn", initializer=variance_scaling_initializer())` to use He initialization).

Training to Predict Time Series

Now let's take a look at how to handle time series, such as stock prices, air temperature, brain wave patterns, and so on. In this section we will train an RNN to predict the next value in a generated time series. Each training instance is a randomly selected sequence of 20 consecutive values from the time series, and the target sequence is the same as the input sequence, except it is shifted by one time step into the future (see Figure 14-7).

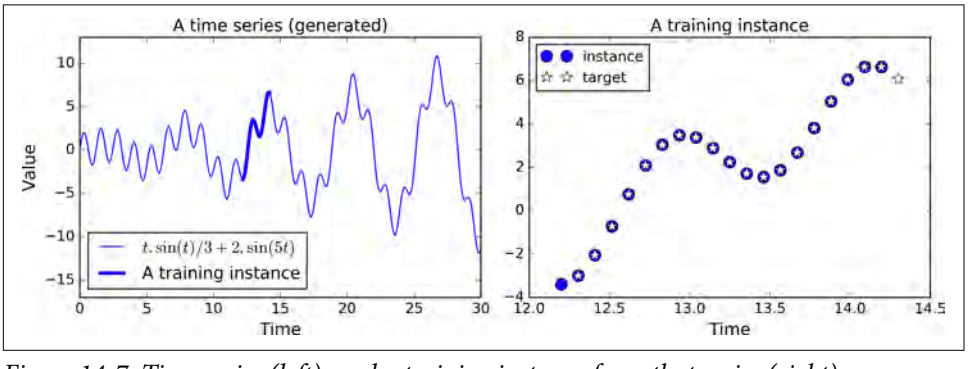

Figure 14-7. Time series (left), and a training instance from that series (right)

First, let's create the RNN. It will contain 100 recurrent neurons and we will unroll it over 20 time steps since each training instance will be 20 inputs long. Each input will contain only one feature (the value at that time). The targets are also sequences of 20 inputs, each containing a single value. The code is almost the same as earlier:

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

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, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

In general you would have more than just one input feature. For example, if you were trying to predict stock prices, you would likely have many other input features at each time step, such as prices of competing stocks, ratings from analysts, or any other feature that might help the system make its predictions.

At each time step we now have an output vector of size 100. But what we actually want is a single output value at each time step. The simplest solution is to wrap the cell in an `OutputProjectionWrapper`. A cell wrapper acts like a normal cell, proxying