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).

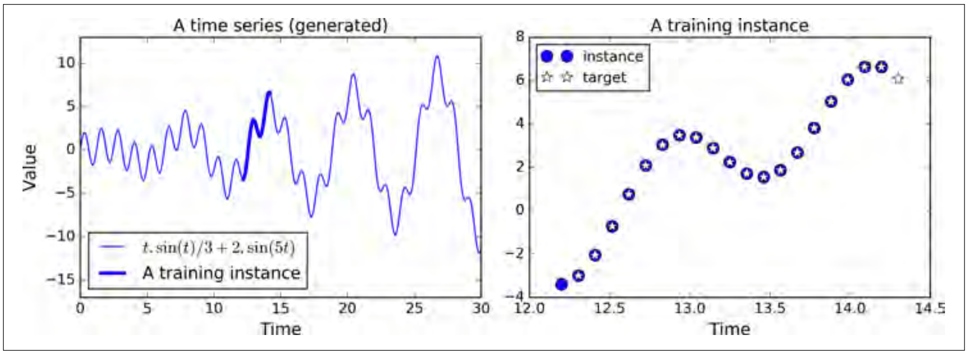

Figure 14-7. Time series (left), and a training instance from that series (right)

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 every method call to an underlying cell, but it also adds some functionality. The OutputProjectionWrapper adds a fully connected layer of linear neurons (i.e., without any activation function) on top of each output (but it does not affect the cell state). All these fully connected layers share the same (trainable) weights and bias terms. The resulting RNN is represented in Figure 14-8.

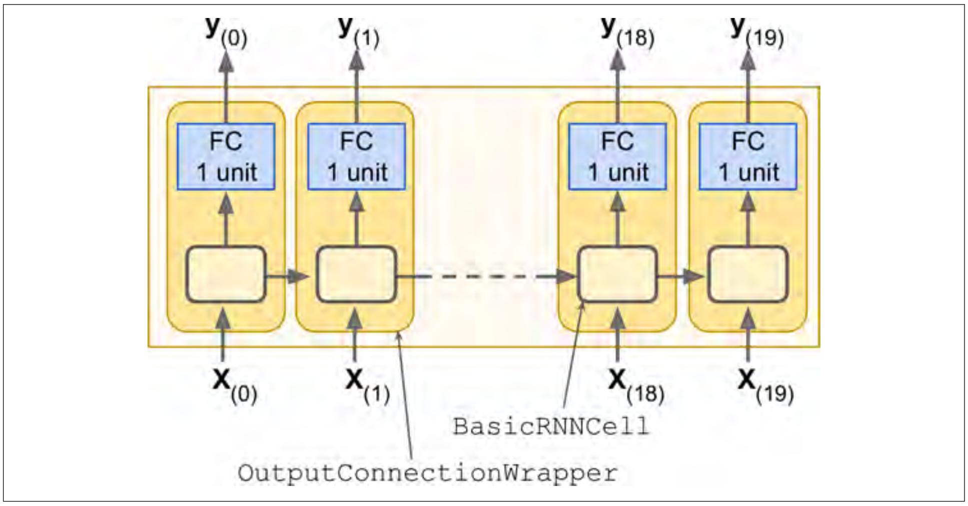

Figure 14-8. RNN cells using output projections

Wrapping a cell is quite easy. Let's tweak the preceding code by wrapping the BasicRNNCell into an OutputProjectionWrapper:

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```

So far, so good. Now we need to define the cost function. We will use the Mean Squared Error (MSE), as we did in previous regression tasks. Next we will create an Adam optimizer, the training op, and the variable initialization op, as usual:

```
learning_rate = 0.001

loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Now on to the execution phase:

```
n_iterations = 10000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = [...]  # fetch the next training batch
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)
```

The program's output should look like this:

```
0    MSE: 379.586
100  MSE: 14.58426
200  MSE: 7.14066
300  MSE: 3.98528
400  MSE: 2.00254
[...]
```

Once the model is trained, you can make predictions:

```
X_new = [...]  # New sequences
y_pred = sess.run(outputs, feed_dict={X: X_new})
```

Figure 14-9 shows the predicted sequence for the instance we looked at earlier (in Figure 14-7), after just 1,000 training iterations.

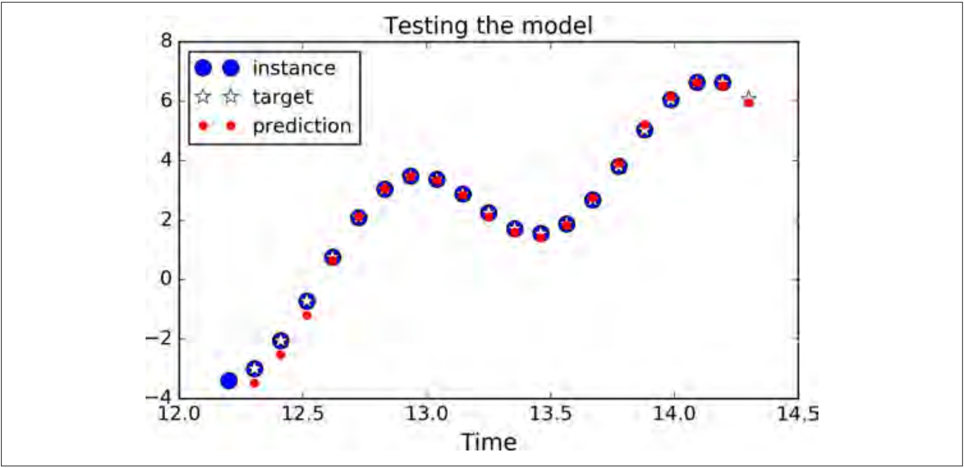

Figure 14-9. Time series prediction

Although using an `OutputProjectionWrapper` is the simplest solution to reduce the dimensionality of the RNN's output sequences down to just one value per time step (per instance), it is not the most efficient. There is a trickier but more efficient solution: you can reshape the RNN outputs from `[batch_size, n_steps, n_neurons]` to `[batch_size * n_steps, n_neurons]`, then apply a single fully connected layer with the appropriate output size (in our case just 1), which will result in an output tensor of shape `[batch_size * n_steps, n_outputs]`, and then reshape this tensor to [batch_size, n_steps, n_outputs]. These operations are represented in Figure 14-10.

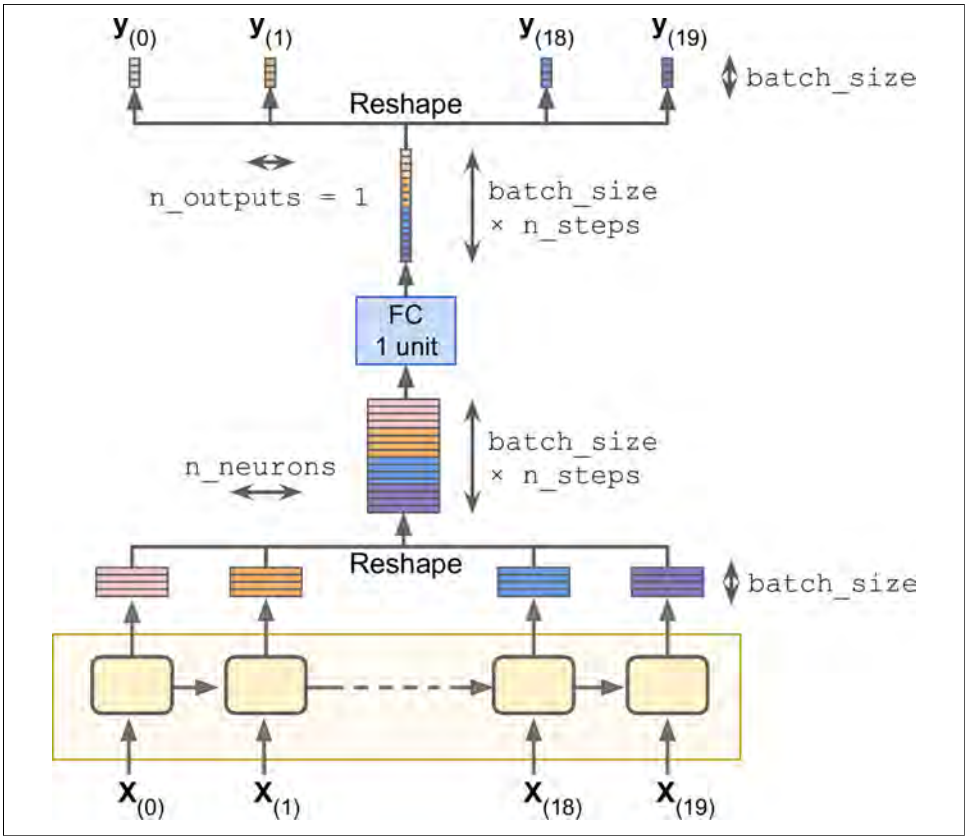

Figure 14-10. Stack all the outputs, apply the projection, then unstack the result

To implement this solution, we first revert to a basic cell, without the OutputProjectionWrapper:

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

Then we stack all the outputs using the reshape() operation, apply the fully connected linear layer (without using any activation function; this is just a projection), and finally unstack all the outputs, again using reshape():

```
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = fully_connected(stacked_rnn_outputs, n_outputs,
                                  activation_fn=None)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
```

The rest of the code is the same as earlier. This can provide a significant speed boost since there is just one fully connected layer instead of one per time step.

Creative RNN

Now that we have a model that can predict the future, we can use it to generate some creative sequences, as explained at the beginning of the chapter. All we need is to provide it a seed sequence containing `n_steps` values (e.g., full of zeros), use the model to predict the next value, append this predicted value to the sequence, feed the last `n_steps` values to the model to predict the next value, and so on. This process generates a new sequence that has some resemblance to the original time series (see Figure 14-11).

```
sequence = [0.] * n_steps
for iteration in range(300):
    X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    sequence.append(y_pred[0, -1, 0])
```

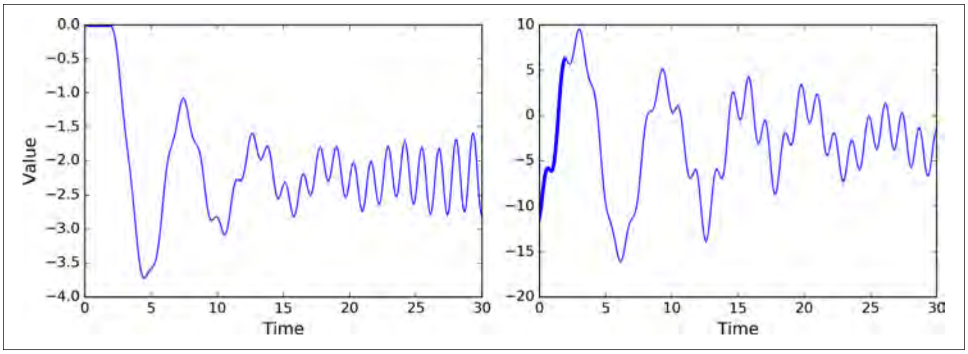

Figure 14-11. Creative sequences, seeded with zeros (left) or with an instance (right)

Now you can try to feed all your John Lennon albums to an RNN and see if it can generate the next "Imagine." However, you will probably need a much more powerful RNN, with more neurons, and also much deeper. Let's look at deep RNNs now.

Deep RNNs

It is quite common to stack multiple layers of cells, as shown in Figure 14-12. This gives you a *deep RNN*.