Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on. This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}$, $\mathbf{X}_{(1)}$, …, $\mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very *basic cell*, but later in this chapter we will look at some more complex and powerful types of cells.

In general a cell's state at time step t, denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Its output at time step t, denoted $\mathbf{y}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is simply equal to the state, but in more complex cells this is not always the case, as shown in Figure 14-3.

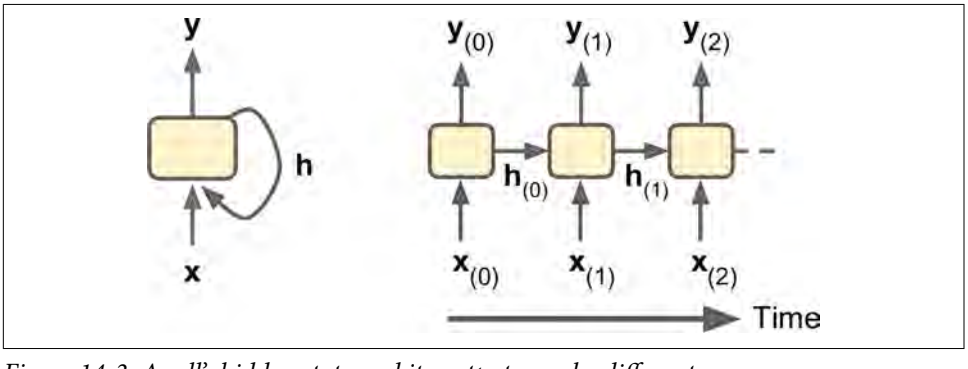

Figure 14-3. A cell's hidden state and its output may be different

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see Figure 14-4, top-left network). For example, this type of network is useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from $N - 1$ days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs, and ignore all outputs except for the last one (see the top-right network). In other words, this is a sequence-to-vector network. For example, you could feed the network a sequence of words cor-

responding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).

Conversely, you could feed the network a single input at the first time step (and zeros for all other time steps), and let it output a sequence (see the bottom-left network). This is a vector-to-sequence network. For example, the input could be an image, and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network). For example, this can be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an Encoder–Decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented on the top left), since the last words of a sentence can affect the first words of the translation, so you need to wait until you have heard the whole sentence before translating it.

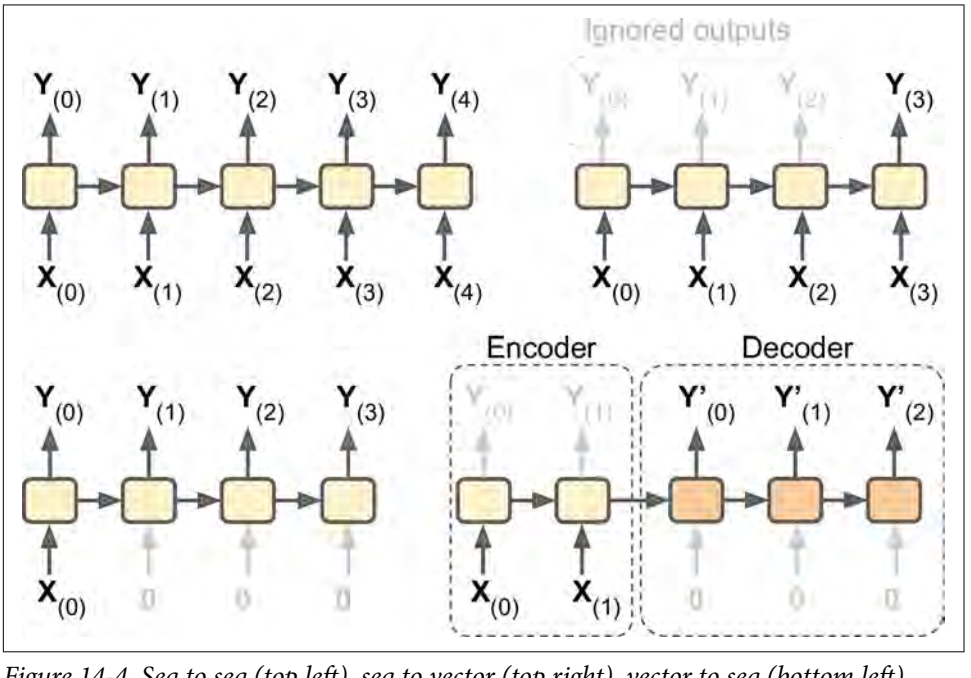

Figure 14-4. Seq to seq (top left), seq to vector (top right), vector to seq (bottom left), delayed seq to seq (bottom right)

Sounds promising, so let's start coding!

Basic RNNs in TensorFlow

First, let's implement a very simple RNN model, without using any of TensorFlow's RNN operations, to better understand what goes on under the hood. We will create an RNN composed of a layer of five recurrent neurons (like the RNN represented in Figure 14-2), using the tanh activation function. We will assume that the RNN runs over only two time steps, taking input vectors of size 3 at each time step. The following code builds this RNN, unrolled through two time steps:

```
n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

This network looks much like a two-layer feedforward neural network, with a few twists: first, the same weights and bias terms are shared by both layers, and second, we feed inputs at each layer, and we get outputs from each layer. To run the model, we need to feed it the inputs at both time steps, like so:

```
import numpy as np

# Mini-batch:        instance 0,instance 1,instance 2,instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

This mini-batch contains four instances, each with an input sequence composed of exactly two inputs. At the end, `Y0_val` and `Y1_val` contain the outputs of the network at both time steps for all neurons and all instances in the mini-batch:

```
>>> print(Y0_val)  # output at t = 0
[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]  # instance 0
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]  # instance 1
 [ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]  # instance 2
 [ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]] # instance 3
>>> print(Y1_val)  # output at t = 1
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]  # instance 0
 [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]  # instance 1
```