# Chapter 13

# Time Series and Recurrent Networks

- Time Series
- Elman Networks
- Jordan Networks
- Deep Recurrent Networks

In this chapter, we will examine time series encoding and recurrent networks, two topics that are logical to put together because they are both methods for dealing with data that spans over time. Time series encoding deals with representing events that occur over time to a neural network. There are many different methods to encode data that occur over time to a neural network. This encoding is necessary because a feedforward neural network will always produce the same output vector for a given input vector. Recurrent neural networks do not require encoding of time series data because they are able to automatically handle data that occur over time.

The variation in temperature during the week is an example of time series data. For instance, if we know that today's temperature is 25 degrees, and tomorrow's temperature is 27 degrees, the recurrent neural networks and time series encoding provide another option to predict the correct temperature for the week. Conversely, a traditional feedforward neural network will always respond with the same output for a given input. If a feedforward neural network is trained to predict tomorrow's temperature, it should respond 27 for 25. The fact that it will always output 27 when given 25 might be a hindrance to its predictions. Surely the temperature of 27 will not always follow 25. It would be better for the neural network to consider the temperatures for a series of days before the day being predicted. Perhaps the temperature over the last week might allow us to predict tomorrow's temperature. Therefore, recurrent neural networks and time series encoding represent two different approaches to the problem of representing data over time to a neural network.

So far the neural networks that we've examined have always had forward connections. The input layer always connects to the first hidden layer. Each hidden layer always connects to the next hidden layer. The final hidden layer always connects to the output layer. This manner to connect layers is the reason that these networks are called "feedforward." Recurrent neural networks are not so rigid, as backward connections are also allowed. A recurrent connection links a neuron in a layer to either a previous layer or the neuron itself. Most recurrent neural network architectures maintain state in the recurrent connections. Feedforward neural networks don't maintain any state. A recurrent neural network's state acts as a sort of short-term memory for the neural network. Consequently, a recurrent neural network will not always produce the same output for a given input.

## 13.1 Time Series Encoding

As we saw in previous chapters, neural networks are particularly good at recognizing patterns, which helps them predict future patterns in data. We refer to a neural network that predicts future patterns as a predictive, or temporal, neural network. These predictive neural networks can anticipate future events, such as stock market trends and sun spot cycles.

Many different kinds of neural networks can predict. In this section, the feedforward neural network will attempt to learn patterns in data so it can predict future values. Like all problems applied to neural networks, prediction is a matter of intelligently determining how to configure input and interpret output neurons for a problem. Because the type of feedforward neural networks in this book always produce the same output for a given input, we need to make sure that we encode the input correctly.

A wide variety of methods can encode time series data for a neural network. The sliding window algorithm is one of the simplest and most popular encoding algorithms. However, more complex algorithms allow the following considerations:

- Weighting older values as less important than newer
- Smoothing/averaging over time
- Other domain-specific (e.g. finance) indicators

We will focus on the sliding window algorithm encoding method for time series. The sliding window algorithm works by dividing the data into two windows that represent the past and the future. You must specify the sizes of both windows. For example, if you want to predict future prices with the daily closing price of a stock, you must decide how far into the past and future that you wish to examine. You might want to predict the next two days using the last five closing prices. In this case, you would have a neural network with five input neurons and two output neurons.

### 13.1.1 Encoding Data for Input and Output Neurons

Consider a simple series of numbers, such as the sequence shown here:

```
1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1
```

A neural network that predicts numbers from this sequence might use three input neurons and a single output neuron. The following training set has a prediction window of size 1 and a past window size of 3:

```
[1 ,2 ,3] -> [4]
[2 ,3 ,4] -> [3]
[3 ,4 ,3] -> [2]
[4 ,3 ,2] -> [1]
```

As you can see, the neural network is prepared to receive several data samples in a sequence. The output neuron then predicts how the sequence will continue. The idea is that you can now feed any sequence of three numbers, and the neural network will predict the fourth number. Each data point is called a time slice. Therefore, each input neuron represents a known time slice. The output neurons represent future time slices.

It is also possible to predict more than one value into the future. The following training set has a prediction window of size 2 and a past window size of 3:

```
[1 ,2 ,3] -> [4 ,3]
[2 ,3 ,4] -> [3 ,2]
[3 ,4 ,3] -> [2 ,1]
[4 ,3 ,2] -> [1 ,2]
```

The last two examples have only a single stream of data. It is possible to use multiple streams of data to predict. For example, you might predict the price with the price of a stock and its volume. Consider the following two streams:

```
Stream #1: 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1
Stream #2: 10, 20, 30, 40, 30, 20, 10, 20, 30, 40, 30, 20, 10
```

You can predict stream #1 with stream #1 and #2. You simply need to add the stream #2 values next to the stream #1 values. A training set can perform this calculation. The following set shows a prediction window of size 1 and a past window size of 3:

```
[1 ,10 ,2 ,20 ,3 ,30] -> [4]
[2 ,20 ,3 ,30 ,4 ,40] -> [3]
[3 ,30 ,4 ,40 ,3 ,30] -> [2]
[4 ,40 ,3 ,30 ,2 ,20] -> [1]
```

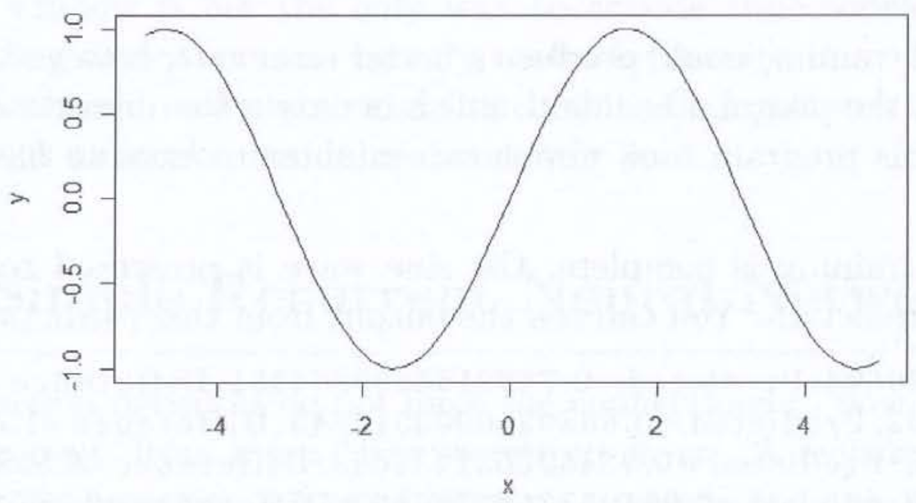
This same technique works for any number of streams. In this case, stream #1 helps to predict itself. For example, you can use the stock prices of IBM and Apple to predict Microsoft. This technique uses three streams. The stream that we're predicting doesn't need to be among the streams providing the data to form the prediction.

### 13.1.2 Predicting the Sine Wave

The example in this section is relatively simple. We present a neural network that predicts the sine wave, which is mathematically predictable. However, programmers can easily understand the sine wave, and it varies over time. These qualities make it a good introduction to predictive neural networks.

You can see the sine wave by plotting the trigonometric sine function. Figure 13.1 shows the sine wave:

**Figure 13.1:** The sine wave



The sine wave function trains the neural network. Backpropagation will adjust the weights to emulate the sine wave. When you first execute the sine wave example, you will see the results of the training process. Typical output from the sine wave predictor's training process follows:

```
Iteration #1 Error:0.48120350975475823 Iteration #2 Error:
0.36753445768855236 Iteration #3 Error:0.3212066601426759
Iteration #4 Error:0.2952410514715732 Iteration #5 Error:
0.2780102928778258 Iteration #6 Error:0.26556861969786527
Iteration #7 Error:0.25605359706505776 Iteration #8 Er236
Introduction to Neural Networks with Java, Second Edition
ror:0.24842242500053566 Iteration #9 Error:0.24204767544134156
  Iteration
#10 Error:0.23653845782593882
...
```

```
Iteration #4990 Error:0.023193976628974254 Iteration #4991 Error:
0.02319310934886356 Iteration #4992 Error:0.023192242246688515
Iteration #4993 Error:0.02319137532183077 Iteration #4994 Error:
0.023190508573672858 Iteration #4995 Error:0.0231896420015976l
Iteration #4996 Error:0.023188775604988862 Iteration #4997 Error:
0.023187909383229866 Iteration #4998 Error:0.023187043335705867
Iteration #4999 Error:0.023186177461801745
```

In the beginning, the error rate is fairly high at 48%. By the second iteration, this rate quickly begins to fall to 36.7%. By the time the 4,999th iteration has occurred, the error rate has fallen to 2.3%. The program is designed to stop before hitting the 5,000th iteration. This succeeds in reducing the error rate to less than 0.03.

Additional training would produce a better error rate; however, by limiting the iterations, the program is able to finish in only a few minutes on a regular computer. This program took about two minutes to execute on an Intel I7 computer.

Once the training is complete, the sine wave is presented to the neural network for prediction. You can see the output from this prediction here:

```
5:Actual=0.76604:Predicted=0.7892166200864351:Difference=2.32% 6:A
ctual=0.86602:Predicted=0.8839210963512845:Difference=1.79% 7:Ac
tual=0.93969:Predicted=0.934526031234053:Difference=0.52% 8:Act
ual=0.9848:Predicted=0.9559577688326862:Difference=2.88% 9:Actu
al=1.0:Predicted=0.9615566601973113:Difference=3.84% 10:Actual=
0.9848:Predicted=0.9558060932656686:Difference=2.90% 11:Actual=
0.93969:Predicted=0.9354447787244102:Difference=0.42% 12:Actual
=0.86602:Predicted=0.8894014978439005:Difference=2.34% 13:Actua
l=0.76604:Predicted=0.801342405700056:Difference=3.53% 14:Actua
l=0.64278:Predicted=0.6633506809125252:Difference=2.06% 15:Actu
al=0.49999:Predicted=0.4910483600917853:Difference=0.89% 16:Act
ual=0.34202:Predicted=0.31286152780645105:Difference=2.92% 17:A
ctual=0.17364:Predicted=0.14608325263568134:Difference=2.76%
18:Actual=0.0:Predicted=-0.008360016796238434:Difference=0.84%
19:Actual=-0.17364:Predicted=-0.15575381460132823:Difference=1.79%
20:Actual=-0.34202:Predicted=-0.3021775158559559:Difference=3.98%
...
490:Actual=-0.64278:Predicted=-0.6515076637590029:Difference=0.87%
491:Actual=-0.76604:Predicted=-0.8133333939237001:Difference=4.73%
492:Actual=-0.86602:Predicted=-0.9076496572125671:Difference=4.16%
493:Actual=-0.93969:Predicted=-0.9492579517460149:Difference=0.96%
```

```
494:Actual=-0.9848:Predicted=-0.9644567437192423:Difference=2.03%
495:Actual=-1.0:Predicted=-0.9664801515670861:Difference=3.35%
496:Actual=-0.9848:Predicted=-0.9579489752650393:Difference=2.69%
497:Actual=-0.93969:Predicted=-0.9340105440194074:Difference=0.57%
498:Actual=-0.86602:Predicted=-0.8829925066754494:Difference=1.70%
499:Actual=-0.76604:Predicted=-0.7913823031308845:Difference=2.53%
```

As you can see, we present both the actual and predicted values for each element. We trained the neural network for the first 250 elements; however, the neural network is able to predict beyond the first 250. You will also notice that the difference between the actual values and the predicted values rarely exceeds 3%.

Sliding window is not the only way to encode time series. Other time series encoding algorithms can be very useful for specific domains. For example, many technical indicators exist that help to find patterns in the value of securities such as stocks, bonds, and currency pairs.

## 13.2 Simple Recurrent Neural Networks

Recurrent neural networks do not force the connections to flow only from one layer to the next, from input layer to output layer. A recurrent connection occurs when a connection is formed between a neuron and one of the following other types of neurons:

- The neuron itself
- A neuron on the same level
- A neuron on a previous level

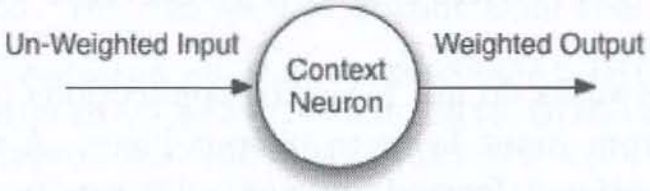
Recurrent connections can never target the input neurons or the bias neurons.

The processing of recurrent connections can be challenging. Because the recurrent links create endless loops, the neural network must have some way to know when to stop. A neural network that entered an endless loop would not be useful. To prevent endless loops, we can calculate the recurrent connections with the following three approaches:

- Context neurons
- Calculating output over a fixed number of iterations
- Calculating output until neuron output stabilizes

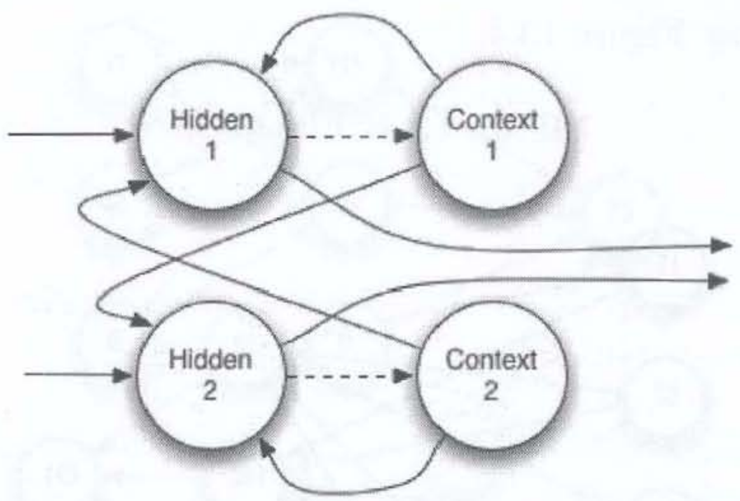
We refer to neural networks that use context neurons as a simple recurrent network (SRN). The context neuron is a special neuron type that remembers its input and provides that input as its output the next time that we calculate the network. For example, if we gave a context neuron 0.5 as input, it would output 0. Context neurons always output 0 on their first call. However, if we gave the context neuron a 0.6 as input, the output would be 0.5. We never weight the input connections to a context neuron, but we can weight the output from a context neuron just like any other connection in a network. Figure 13.2 shows a typical context neuron:

**Figure 13.2:** Context Neuron



Context neurons allow us to calculate a neural network in a single feed-forward pass. Context neurons usually occur in layers. A layer of context neurons will always have the same number of context neurons as neurons in its source layer, as demonstrated by Figure 13.3:

**Figure 13.3:** Context Layer



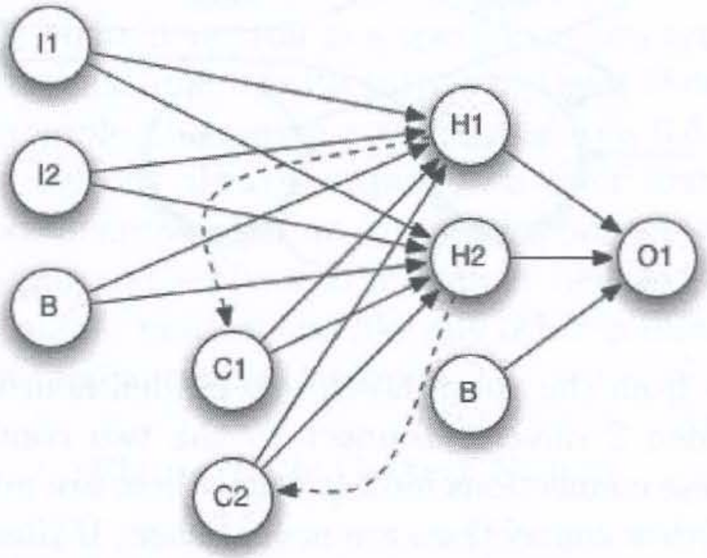
As you can see from the above layer, two hidden neurons that are labeled hidden 1 and hidden 2 directly connect to the two context neurons. The dashed lines on these connections indicate that these are not weighted connections. These weightless connections are never dense. If these connections were dense, hidden 1 would be connected to both hidden 1 and hidden 2. However, the direct connection simply joins each hidden neuron to its corresponding context neuron. The two context neurons form dense, weighted connections to the two hidden neurons. Finally, the two hidden neurons also form dense connections to the neurons in the next layer. The two context neurons would form two connections to a single neuron in the next layer, four connections to two neurons, six connections to three neurons, and so on.

You can combine context neurons with the input, hidden, and output layers of a neural network in many different ways. In the next two sections, we explore two common SRN architectures.

### 13.2.1 Elman Neural Networks

In 1990, Elman introduced a neural network that provides pattern recognition to time series. This neural network type has one input neuron for each stream that you are using to predict. There is one output neuron for each time slice you are trying to predict. A single-hidden layer is positioned between the input and output layer. A layer of context neurons takes its input from the hidden layer output and feeds back into the same hidden layer. Consequently, the context layers always have the same number of neurons as the hidden layer, as demonstrated by Figure 13.4:

**Figure 13.4:** Elman SRN

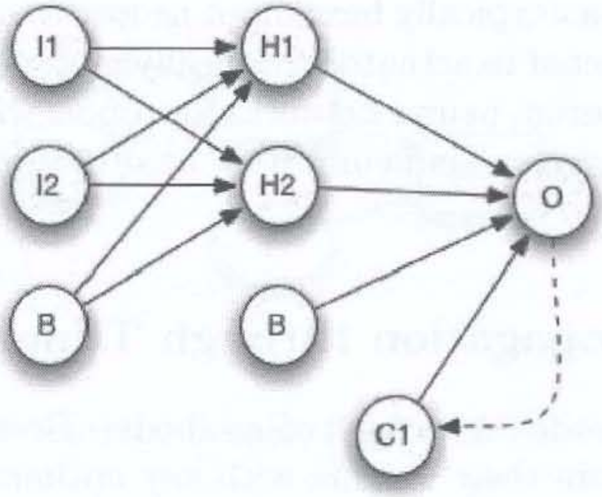


The Elman neural network is a good general-purpose architecture for simple recurrent neural networks. You can pair any reasonable number of input neurons to any number of output neurons. Using normal weighted connections, the two context neurons are fully connected with the two hidden neurons. The two context neurons receive their state from the two non-weighted connections (dashed lines) from each of the two hidden neurons.

## 13.2.2 Jordan Neural Networks

In 1993, Jordan introduced a neural network to control electronic systems. This style of SRN is similar to Elman networks. However, the context neurons are fed from the output layer instead of the hidden layer. We also refer to the context units in a Jordan network as the state layer. They have a recurrent connection to themselves with no other nodes on this connection, as seen in Figure 13.5:

**Figure 13.5:** Jordan SRN



The Jordan neural network requires the same number of context neurons and output neurons. Therefore, if we have one output neuron, the Jordan network will have a single context neuron. This equality can be problematic if you have only a single output neuron because you will be able to have just one single-context neuron.

The Elman neural network is applicable to a wider array of problems than the Jordan network because the large hidden layer creates more context neurons. As a result, the Elman network can recall more complex patterns because it captures the state of the hidden layer from the previous iteration. This state is never bipolar since the hidden layer represents the first line of feature detectors.

Additionally, if we increase the size of the hidden layer to account for a more complex problem, we also get more context neurons with an Elman network. The Jordan network doesn't produce this effect. To create more context neurons with the Jordan network, we must add more output neurons. We cannot add output neurons without changing the definition of the problem.

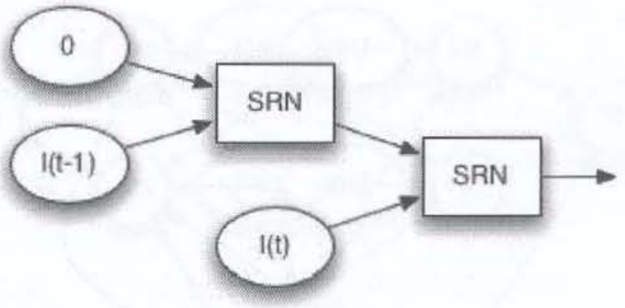
When to use a Jordan network is a common question. Programmers originally developed this network type for robotics research. Neural networks that are designed for robotics typically have input neurons connected to sensors and output neurons connected to actuators (typically motors). Because each motor has its own output neuron, neural networks for robots will generally have more output neurons than regression neural networks that predict a single value.

### 13.2.3 Backpropagation through Time

You can train SRNs with a variety of methods. Because SRNs are neural networks, you can train their weights with any optimization algorithm, such as simulated annealing, particle swarm optimization, Nelder-Mead or others. Regular backpropagation-based algorithms can also train of the SRN. Mozer (1995), Robinson & Fallside (1987) and Werbos (1988) each invented an algorithm specifically designed for SRNs. Programmers refer to this algorithm as backpropagation through time (BPTT). Sjoberg, Zhang, Ljung, et al. (1995) determined that backpropagation through time provides superior training performance than general optimization algorithms, such as simulated annealing. Backpropagation through time is even more sensitive to local minima than standard backpropagation.

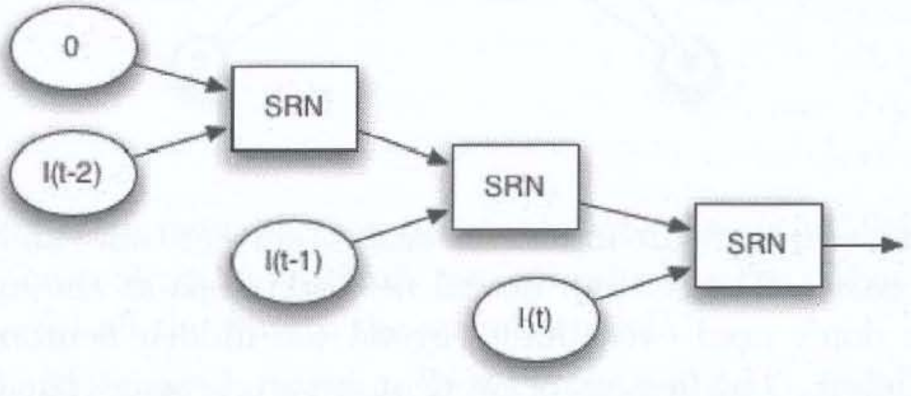
Backpropagation through time works by unfolding the SRN to become a regular neural network. To unfold the SRN, we construct a chain of neural networks equal to how far back in time we wish to go. We start with a neural network that contains the inputs for the current time, known as *t*. Next we replace the context with the entire neural network, up to the context neuron's input. We continue for the desired number of time slices and replace the final context neuron with a 0. Figure 13.6 illustrates this process for two time slices.

**Figure 13.6:** Unfolding to Two Time Slices



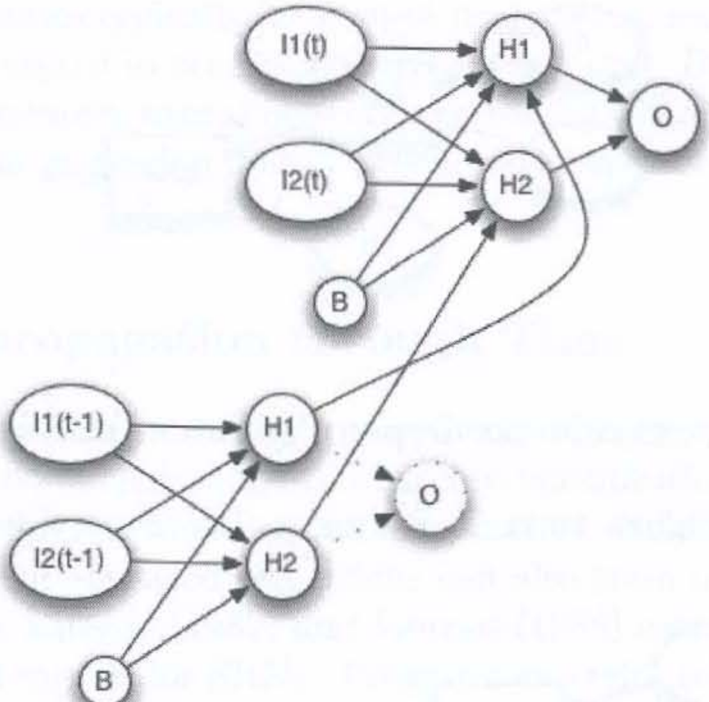
This unfolding can continue deeper; Figure 13.7 shows three time slices:

**Figure 13.7:** Unfolding to Two Time Slices



You can apply this abstract concept to the actual SRNs. Figure 13.8 illustrates a two-input, two-hidden, one-output Elman neural network unfolded to two time slices:

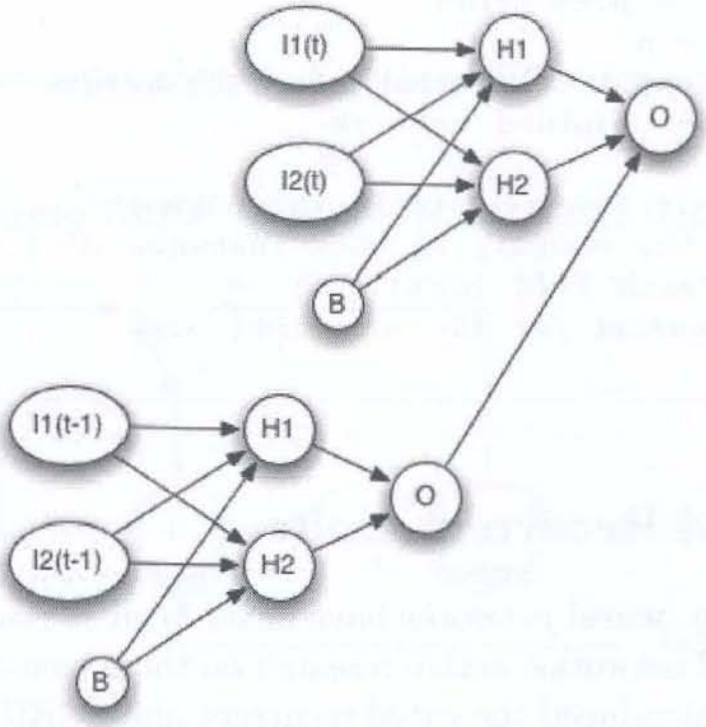
**Figure 13.8:** Elman Unfolded to Two Time Slices



As you can see, there are inputs for both *t* (current time) and *t-1* (one time slice in the past). The bottom neural network stops at the hidden neurons because you don't need everything beyond the hidden neurons to calculate the context input. The bottom network structure becomes the context to the top network structure. Of course, the bottom structure would have had a context as well that connects to its hidden neurons. However, because the output neuron above does not contribute to the context, only the top network (current time) has one.

It is also possible to unfold a Jordan neural network. Figure 13.9 shows a two-input, two-hidden, one-output Jordan network unfolded to two time slices.

**Figure 13.9:** Jordan Unfolded to Two Time Slices



Unlike the Elman network, you must calculate the entire Jordan network to determine the context. As a result, we can calculate the previous time slice (bottom network) all the way to the output neuron.

To train the SRN, we can use regular backpropagation to train the unfolded network. However, at the end of the iteration, we average the weights of all folds to obtain the weights for the SRN. Listing 13.1 describes the BPTT algorithm:

**Listing 13.1:** Backpropagation Through Time (BPTT):

```
def bptt(a, y)
# a[t] is the input at time t. y[t] is the output
   .. unfold the network to contain k instances of f
   .. see above figure..
   while stopping criteria no met:
# x is the current context
      x = []
      for t from 0 to n Ű 1:
# t is time. n is the length of the training sequence
         .. set the network inputs to x, a[t], a[t+1], ..., a[t+k-1]
```

```
    p = .. forward-propagation of the inputs
          .. over the whole unfolded network
# error = target - prediction
    e = y[t+k] - p
    .. Back-propagate the error, e, back across
    .. the whole unfolded network

    .. Update all the weights in the network
    .. Average the weights in each instance of f together,
    .. so that each f is identical
# compute the context for the next time-step
    x = f(x)
```

### 13.2.4 Gated Recurrent Units

Although recurrent neural networks have never been as popular as the regular feedforward neural networks, active research on them continues. Chung, Hyun & Bengio (2014) introduced the gated recurrent unit (GRU) to allow recurrent neural networks to function in conjunction with deep neural network by solving some inherent limitations of recurrent neural networks. GRUs are neurons that provide a similar role to the context neurons seen previously in this chapter.

It is difficult to train RNNs to capture long-term dependencies because the gradients tend to either vanish (most of the time) or explode (rarely, but with severe effects), as demonstrated by Chung, Hyun & Bengio (2015).

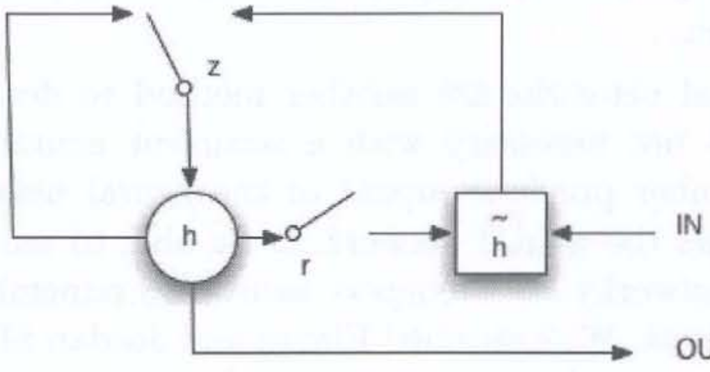
As of the 2015 publication of this book, GRUs are less than a year old. Because of the cutting edge nature of GRUs, most major neural network frameworks do not currently include them. If you would like to experiment with GRUs, the Python Theano-based framework Keras includes them. You can find the framework at the following URL:

**https://github.com/fchollet/keras**

Though we usually use Lasange, Keras is one of many Theano-based frameworks for Python, and it is also one of the first to support GRUs. This section contains a brief, high-level introduction to GRU, and we will update the book's examples as needed to support this technology as it becomes available. Refer to the book's example code for up-to-date information on example availability for GRU.

A GRU uses two gates to overcome these limitations, as shown in Figure 13.10:

**Figure 13.10:** Gated Recurrent Unit (GRU)



The gates are indicated by *z*, the update gate, and *r*, the reset gate. The values *h* and *tilde-h* represent the activation (output) and candidate activation. It is important to note that the switches specify ranges, rather than simply being on or off.

The primary difference between the GRU and traditional recurrent neural networks is that the entire context value does not change its value each iteration as it does in the SRN. Rather, the update gate governs the degree of update to the context activation that occurs. Additionally, the program provides a reset gate that allows the context to be reset.

## 13.3 Chapter Summary

In this chapter, we introduced several methods that can handle time series data with neural networks. A feedforward neural network produces the same output when provided the same input. As a result, feedforward neural networks are said to be deterministic. This quality does not allow a feedforward neural network the ability to produce output, given a series of inputs. If your application must provide output based on a series of inputs, you have two choices. You can encode the time series into an input feature vector or use a recurrent neural network.

Encoding a time series is a way of capturing time series information in a feature vector that is fed to a feedforward neural network. A number of methods encode time series data. We focused on sliding window encoding. This method specifies two windows. The first window determines how far into the past to use for prediction. The second window determines how far into the future to predict.

Recurrent neural networks are another method to deal with time series data. Encoding is not necessary with a recurrent neural network because it is able to remember previous inputs to the neural network. This short-term memory allows the neural network to be able to see patterns in time. Simple recurrent networks use a context neuron to remember the state from previous computations. We examined Elman and Jordan SRNs. Additionally, we introduced a very new neuron type called the gated recurrent unit (GRU). This neuron type does not immediately update its context value like the Elman and Jordan networks. Two gates govern the degree of update.

Hyper-parameters define the structure of a neural network and ultimately determine its effectiveness for a particular problem. In the previous chapters of this book, we introduced hyper-parameters such as the number of hidden layers and neurons, the activation functions, and other governing attributes of neural networks. Determining the correct set of hyper-parameters is often a difficult task of trial and error. However, some automated processes can make this process easier. Additionally, some rules of thumb can help architect these neural networks. We cover these pointers, as well as automated processes, in the next chapter.

[illegible]