Chapter 10: Application to the Financial Markets

- Creating Input and Output Neurons for Prediction
- How to Create Training Sets for a Predictive Neural Network
- Predicting the Sine Wave

In the last chapter, you saw that neural networks can be used to predict trends in numeric data, as in the sine wave example. Predicting the sine wave was useful in demonstrating how to create neural networks that can predict, but it has little real world application. The purpose of chapter 9 was to introduce the fundamentals of how to predict with a neural network. This chapter builds upon the material presented in chapter 9 by providing you with a foundation for applying neural networks to financial market problems.

In this chapter, a relatively simple program is presented that attempts to predict the S&P 500 index. The keyword in the last sentence is "attempts." This chapter is for educational purposes only and is by no means an investment strategy, since past performance is no indication of future returns. The material presented here can be used as a starting point from which to adapt neural networks to augment your own investment strategy.

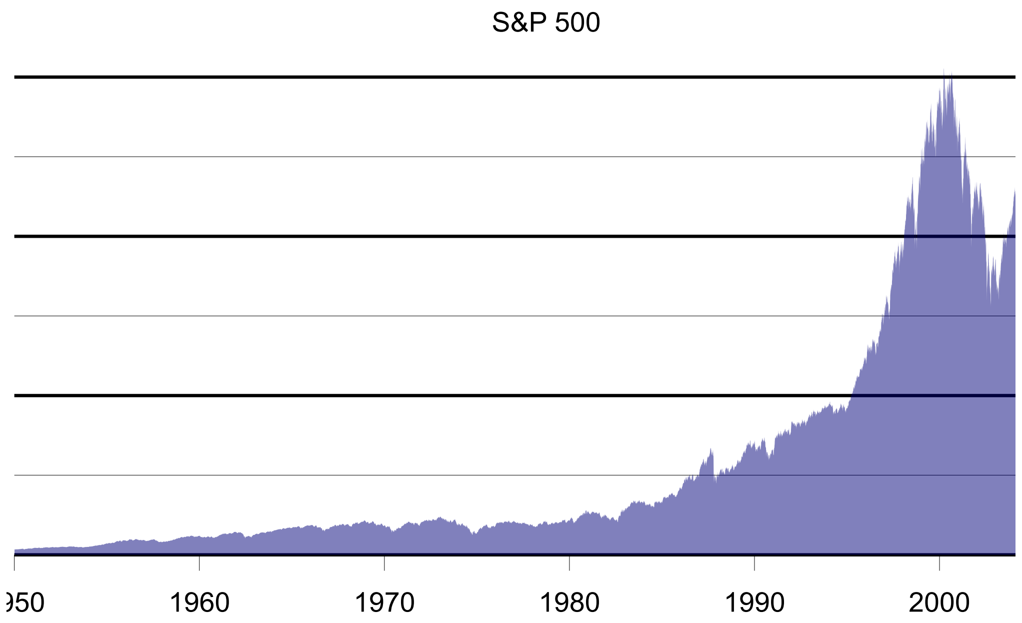
Collecting Data for the S&P 500 Neural Network

Before we discuss how to predict direction in the S&P 500, we should first clarify what the S&P 500 is and how it functions.

"The S&P 500 is a stock market index containing the stocks of 500 Large-Cap corporations, most of which are American. The index is the most notable of the many indices owned and maintained by Standard & Poor's, a division of McGraw-Hill. S&P 500 is used in reference not only to the index but also to the 500 companies that have their common stock included in the index." (From www.wikipedia.org)

Figure 10.1 shows movement in the S&P 500 since 1950.

Figure 10.1: The S&P 500 stock index (From www.wikipedia.org).

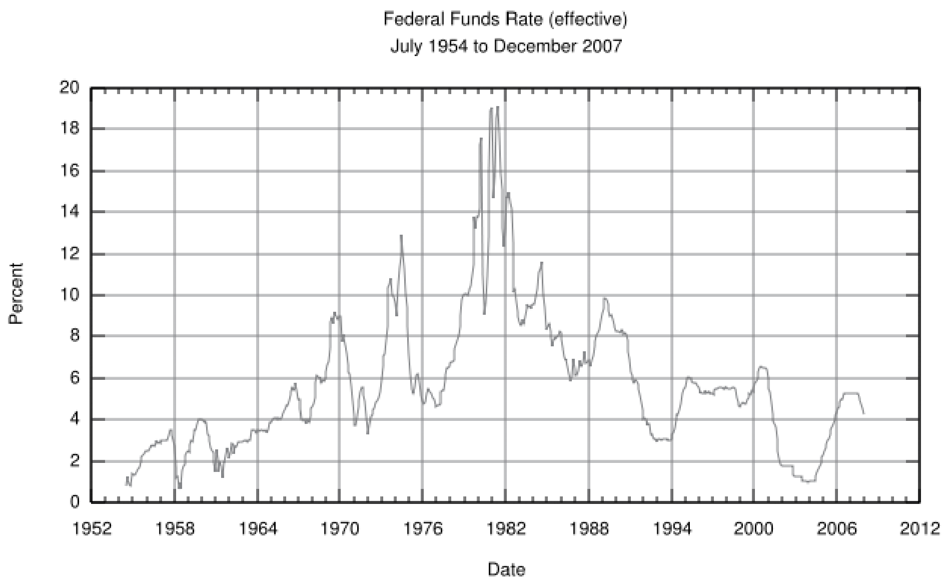


Historical S&P 500 values will be used to predict future S&P 500 values; however, the S&P 500 data will not be considered in a vacuum. The current prime interest rate will also be adjusted to aid in the detection of patterns. The prime interest rate is defined as follows:

“Prime rate is a term applied in many countries to a reference interest rate used by banks. The term originally indicated the rate of interest at which banks lent to favored customers, [...] though this is no longer always the case. Some variable interest rates may be expressed as a percentage above or below prime rate.” (From www.wikipedia.org)

Figure 10.2 shows the US prime interest rate over time.

Figure 10.2: US prime interest rate (From www.wikipedia.org).



The neural network presented in this chapter must be provided with both the S&P 500 historical data and the prime interest rate historical data. The program is designed to receive both data inputs in comma separated value (CSV) files.

Obtaining S&P 500 Historical Data

When you download the examples for this book, you will also be downloading a set of S&P 500 historical data. The data provided was current as of May 2008. If you would like more current financial data, you can obtain it from many sites on the Internet, free of charge. One such site is Yahoo! Finance. Historical S&P 500 data from the 1950s to present can be accessed at the URL:

http://finance.yahoo.com/q/hp?s=%5EGSPC

The file **sp500.csv**, which is included with the companion download for this book, contains historical S&P 500 data from the 1950s to May 14, 2008. Data from this file is shown in Listing 10.1.

Listing 10.1: S&P 500 Historical Data (sp500.csv)

```
Date,Open,High,Low,Close,Volume,Adj Close 2008-04-18,1369.00,1395.
90,1369.00,1390.33,4222380000,1390.33 2008-04-17,1363.37,1368.60,1
357.25,1365.56,3713880000,1365.56 2008-04-16,1337.02,1365.49,1337.
02,1364.71,4260370000,1364.71 2008-04-15,1331.72,1337.72,1324.35,1
```

```
334.43,3581230000,1334.43 2008-04-14,1332.20,1335.64,1326.16,1328.
32,3565020000,1328.32 2008-04-11,1357.98,1357.98,1331.21,1332.83,3
723790000,1332.83 2008-04-10,1355.37,1367.24,1350.11,1360.55,36861
50000,1360.55 2008-04-09,1365.50,1368.39,1349.97,1354.49,35566700
00,1354.49 2008-04-08,1370.16,1370.16,1360.62,1365.54,3602500000,
1365.54 2008-04-07,1373.69,1386.74,1369.02,1372.54,3747780000,137
2.54 2008-04-04,1369.85,1380.91,1362.83,1370.40,3703100000,1370.40
...
1950-01-23,16.92,16.92,16.92,16.92,1340000,16.92 1950-01-20,16.90
,16.90,16.90,16.90,1440000,16.90 1950-01-19,16.87,16.87,16.87,16.
87,1170000,16.87 1950-01-18,16.85,16.85,16.85,16.85,1570000,16.85
1950-01-17,16.86,16.86,16.86,16.86,1790000,16.86 1950-01-16,16.72
,16.72,16.72,16.72,1460000,16.72 1950-01-13,16.67,16.67,16.67,16.
67,3330000,16.67 1950-01-12,16.76,16.76,16.76,16.76,2970000,16.76
1950-01-11,17.09,17.09,17.09,17.09,2630000,17.09 1950-01-10,17.03
,17.03,17.03,17.03,2160000,17.03 1950-01-09,17.08,17.08,17.08,17.
08,2520000,17.08 1950-01-06,16.98,16.98,16.98,16.98,2010000,16.98
1950-01-05,16.93,16.93,16.93,16.93,2550000,16.93 1950-01-04,16.85
,16.85,16.85,16.85,1890000,16.85 1950-01-03,16.66,16.66,16.66,16.
66,1260000,16.66
```

A CSV file contains data such that each line is a record and commas separate individual fields within each line. As mentioned earlier, the example presented in this chapter also uses prime interest rate data.

Obtaining Prime Interest Rate Data

There are many Internet sites that provide historical prime interest rate data. The companion download for this book contains a file named **`prime.csv`**. This file contains prime interest rates from approximately the same time period as the S&P 500 data provided. The contents of **`prime.csv`** are shown in Listing 10.2.

Listing 10.2: Prime Interest Rate Historical Data

```
date,prime
1955-08-04,3.25
1955-10-14,3.50
1956-04-13,3.75
1956-08-21,4.00
1957-08-06,4.50
1958-01-22,4.00
1958-04-21,3.50
1958-09-11,4.00
1959-05-18,4.50
...
2005-12-13,7.25
2006-01-31,7.50
2006-03-28,7.75
```

```
2006-05-10,8.00
2006-06-29,8.25
2007-09-18,7.75
2007-10-31,7.50
2007-12-11,7.25
2008-01-22,6.50
2008-01-30,6.00
2008-03-18,5.25
```

The data in this file will be combined with the S&P 500 data to form the actual data to be used to train the S&P 500 neural network.

Running the S&P 500 Prediction Program

There are two modes of operation for the S&P 500 prediction application. The mode of operation depends upon the command line arguments provided to the program. If no command line arguments are specified, then the neural network is loaded from the file **sp500.net**. If the command argument **FULL** is specified, then the neural network will train a new neural network and save it to disk under the name **sp500.net**.

It can take many hours to completely train the neural network. Therefore, you will not want to run it in full mode every time. However, if you choose to change some of the training parameters, you should retrain the neural network and generate a new **sp500.net** file. The companion download contains an **sp500.net** file that has been trained within 2% accuracy of the training sets.

If you run the program in full training mode, the following output will be produced:

Listing 10.3: Training the SP500 Neural Network

```
Samples read: 14667 Iteration(Backprop) #1
Error:0.6999154401150052 Iteration(Backprop) #2
Error:0.6464464887928701 Iteration(Backprop) #3
Error:0.584286620498403 Iteration(Backprop) #4
Error:0.5161413540009822 Iteration(Backprop) #5
Error:0.44688028770366317 Iteration(Backprop) #6
Error:0.3832980672593392 Iteration(Backprop) #7
Error:0.33189098575632436 Iteration(Backprop) #8
Error:0.2958585679317178 Iteration(Backprop) #9
Error:0.2738894563079073 Iteration(Backprop) #10
Error:0.2619015539956993
...
Iteration(Backprop) #2038 Error:0.020032706833329087
Iteration(Backprop) #2039 Error:0.02002936831637675
Iteration(Backprop) #2040 Error:0.020026031153749693
Iteration(Backprop) #2041 Error:0.020022695344982695
Iteration(Backprop) #2042 Error:0.02001936088961063
Iteration(Backprop) #2043 Error:0.02001602778716852
Iteration(Backprop) #2044 Error:0.0200126960371914
Iteration(Backprop) #2045 Error:0.020009365639214557
Iteration(Backprop) #2046 Error:0.020006036592773283
Iteration(Backprop) #2047 Error:0.02000270889740304
Iteration(Backprop) #2048 Error:0.019999382552639385
```

As you can see, it took a considerable number of training iterations to train the neural network to the desired level. Immediately after the training for this network was complete, the neural network was run in prediction mode and the following output was produced:

Listing 10.4: Predicting the SP500 Neural Network

```
2007-01-03:Start=1416.6,Actual % Change=-0.12%,Predicted % Change=
1.78%:Difference=1.90%
2007-01-04:Start=1418.34,Actual % Change=0.12%,Predicted % Change=
1.39%:Difference=1.27%
2007-01-05:Start=1409.71,Actual % Change=-0.61%,Predicted %
Change= 1.06%:Difference=1.67%
2007-01-08:Start=1412.84,Actual % Change=0.22%,Predicted % Change=
1.28%:Difference=1.06%
2007-01-09:Start=1412.11,Actual % Change=-0.05%,Predicted %
Change= 1.35%:Difference=1.41%
2007-01-10:Start=1414.85,Actual % Change=0.19%,Predicted % Change=
0.87%:Difference=0.67%
2007-01-11:Start=1423.82,Actual % Change=0.63%,Predicted % Change=
0.53%:Difference=0.11%
2007-01-12:Start=1430.73,Actual % Change=0.49%,Predicted % Change=
1.42%:Difference=0.94%
2007-01-16:Start=1431.9,Actual % Change=0.08%,Predicted % Change=
1.71%:Difference=1.63%
...
2008-03-27:Start=1325.76,Actual % Change=-1.15%,Predicted %
Change= 2.85%:Difference=3.99%
2008-03-28:Start=1315.22,Actual % Change=-0.80%,Predicted %
Change= 3.20%:Difference=4.00%
2008-03-31:Start=1322.7,Actual % Change=0.57%,Predicted % Change=
-0.78%:Difference=1.34%
2008-04-01:Start=1370.18,Actual % Change=3.59%,Predicted % Change=
-0.43%:Difference=4.02%
2008-04-02:Start=1367.53,Actual % Change=-0.19%,Predicted %
Change= 1.30%:Difference=1.49%
2008-04-03:Start=1369.31,Actual % Change=0.13%,Predicted % Change=
-0.56%:Difference=0.69%
2008-04-04:Start=1370.4,Actual % Change=0.08%,Predicted % Change=
1.30%:Difference=1.22%
2008-04-07:Start=1372.54,Actual % Change=0.16%,Predicted % Change=
2.25%:Difference=2.09%
2008-04-08:Start=1365.54,Actual % Change=-0.51%,Predicted %
Change= 2.09%:Difference=2.60%
2008-04-09:Start=1354.49,Actual % Change=-0.81%,Predicted %
Change= 2.23%:Difference=3.04%
2008-04-10:Start=1360.55,Actual % Change=0.45%,Predicted % Change=
1.50%:Difference=1.05%
2008-04-11:Start=1332.83,Actual % Change=-2.04%,Predicted %
Change= -0.94%:Difference=1.10%
2008-04-14:Start=1328.32,Actual % Change=-0.34%,Predicted %
Change= -1.25%:Difference=0.91%
2008-04-15:Start=1334.43,Actual % Change=0.46%,Predicted % Change=
0.43%:Difference=0.03%
2008-04-16:Start=1364.71,Actual % Change=2.27%,Predicted % Change=
-0.03%:Difference=2.30%
2008-04-17:Start=1365.56,Actual % Change=0.06%,Predicted % Change=
-0.02%:Difference=0.08%
2008-04-18:Start=1390.33,Actual % Change=1.81%,Predicted % Change=
1.04%:Difference=0.77%
```

As you can see from the above data, the prediction is far from perfect, although it does generally stay within an accuracy range of 10%. It could easily fare much better, or far worse, as additional data becomes available.

Creating the Actual S&P 500 Data

As mentioned earlier, the actual neural network input data is composed of both the prime interest rate and S&P 500 historical data. This data is stored in a class named **SP500Actual**. This class is shown in Listing 10.5.

Listing 10.5: Storing Actual S&P 500 Data (SP500Actual.java)

```
package com.heatonresearch.book.introneuralnet.ch10.sp500;

import java.io.IOException;
import java.text.ParseException;
import java.util.Date;
import java.util.Set;
import java.util.TreeSet;

import com.heatonresearch.book.introneuralnet.common.ReadCSV;

/**
 * Chapter 10: Application to the Financial Markets
 *
 * SP500Actual: Holds actual SP500 data and prime interest rates.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class SP500Actual {

      private final Set<InterestRate> rates =
            new TreeSet<InterestRate>();
      private final Set<FinancialSample> samples =
            new TreeSet<FinancialSample>();
      private final int inputSize;
      private final int outputSize;

      public SP500Actual(final int inputSize, final int outputSize)
{
            this.inputSize = inputSize;
            this.outputSize = outputSize;
      }

      public void calculatePercents() {
            double prev = -1;
            for (final FinancialSample sample : this.samples) {
                  if (prev != -1) {
                        final double movement =
                        sample.getAmount() - prev;
                        final double percent = movement / prev;
                        sample.setPercent(percent);
                  }
                  prev = sample.getAmount();
            }
      }

      public void getInputData(final int offset,
            final double[] input) {
            final Object[] samplesArray = this.samples.toArray();
            // get SP500 & prime data
            for (int i = 0; i < this.inputSize; i++) {
                  final FinancialSample sample =
                  (FinancialSample) samplesArray[offset
                              + i];
                  input[i] = sample.getPercent();
                  input[i + this.outputSize] = sample.getRate();
            }
      }

      public void getOutputData(final int offset,
            final double[] output) {
```

```
		final Object[] samplesArray = this.samples.toArray();
		for (int i = 0; i < this.outputSize; i++) {
			final FinancialSample sample =
			(FinancialSample) samplesArray[offset
					+ this.inputSize + i];
			output[i] = sample.getPercent();
		}

	}

	public double getPrimeRate(final Date date) {
		double currentRate = 0;

		for (final InterestRate rate : this.rates) {
			if (rate.getEffectiveDate().after(date)) {
				return currentRate;
			} else {
				currentRate = rate.getRate();
			}
		}
		return currentRate;
	}

	/**
	 * @return the samples
	 */
	public Set<FinancialSample> getSamples() {
		return this.samples;
	}

	public void load(final String sp500Filename,
		final String primeFilename)
			throws IOException, ParseException {
		loadSP500(sp500Filename);
		loadPrime(primeFilename);
		stitchInterestRates();
		calculatePercents();
	}

	public void loadPrime(final String primeFilename)
		throws IOException,
			ParseException {
		final ReadCSV csv = new ReadCSV(primeFilename);

		while (csv.next()) {
			final Date date = csv.getDate("date");
```

```
                final double rate = csv.getDouble("prime");
                final InterestRate ir =
                        new InterestRate(date, rate);
                this.rates.add(ir);
            }

            csv.close();
        }

        public void loadSP500(final String sp500Filename)
                throws IOException,
                ParseException {
            final ReadCSV csv = new ReadCSV(sp500Filename);
            while (csv.next()) {
                final Date date = csv.getDate("date");
                final double amount = csv.getDouble("adj close");
                final FinancialSample sample =
                        new FinancialSample();
                sample.setAmount(amount);
                sample.setDate(date);
                this.samples.add(sample);
            }
            csv.close();
        }

        public int size() {
            return this.samples.size();
        }

        public void stitchInterestRates() {
            for (final FinancialSample sample : this.samples) {
                final double rate =
                getPrimeRate(sample.getDate());
                sample.setRate(rate);
            }
        }

}
```

There are several functions that this class provides. These functions will be explored in the next sections.

Financial Samples

The primary purpose for the **`SP500Actual`** class is to provide an SP500 quote and the prime interest rate for any given day that the US stock market was open. Additionally, the percent change between the current quote and the previous quote is stored. Together, these values are called a sample. Samples are stored in the **`FinancialSample`** class. The **`FinancialSample`** class is shown in Listing 10.6.

Listing 10.6: Financial Samples (FinancialSample.java)

```
package com.heatonresearch.book.introneuralnet.ch10.sp500;

import java.text.NumberFormat;
import java.util.Date;

import com.heatonresearch.book.introneuralnet.common.ReadCSV;

/**
 * Chapter 10: Application to the Financial Markets
 *
 * FinancialSample: Holds a sample of financial data at the
 * specified date.  This includes the close of the SP500 and
 * the prime interest rate.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class FinancialSample implements
Comparable<FinancialSample> {
      private double amount;
      private double rate;
      private Date date;
      private double percent;

      public int compareTo(final FinancialSample other) {
            return getDate().compareTo(other.getDate());
      }

      /**
       * @return the amount
       */
      public double getAmount() {
            return this.amount;
      }

      /**
```

```
	 * @return the date
	 */
	public Date getDate() {
		return this.date;
	}

	/**
	 * @return the percent
	 */
	public double getPercent() {
		return this.percent;
	}

	/**
	 * @return the rate
	 */
	public double getRate() {
		return this.rate;
	}

	/**
	 * @param amount
	 *            the amount to set
	 */
	public void setAmount(final double amount) {
		this.amount = amount;
	}

	/**
	 * @param date
	 *            the date to set
	 */
	public void setDate(final Date date) {
		this.date = date;
	}

	/**
	 * @param percent
	 *            the percent to set
	 */
	public void setPercent(final double percent) {
		this.percent = percent;
	}

	/**
	 * @param rate
	 *		the rate to set
	 */
	public void setRate(final double rate) {
		this.rate = rate;
	}

	@Override
	public String toString() {
		final NumberFormat nf =
			NumberFormat.getPercentInstance();
		nf.setMinimumFractionDigits(2);
		nf.setMaximumFractionDigits(2);
		final StringBuilder result = new StringBuilder();
		result.append(ReadCSV.displayDate(this.date));
		result.append(", Amount: ");
		result.append(this.amount);
		result.append(", Prime Rate: ");
		result.append(this.rate);
		result.append(", Percent from Previous: ");
		result.append(nf.format(this.percent));
		return result.toString();
	}

}
```

Another important feature of the **FinancialSample** class is that it implements the **Comparable** interface. This allows the **FinancialSample** objects to be added to a sorted Java collection and be ordered by their dates.

Get Prime Rate for a Day

The prime interest rate data file, named **prime.csv**, only contains the prime interest rate for days on which the interest rate changed. Therefore, a special method is required to determine what the interest rate was on a specific date. This method is called **getPrimeRate**. The signature for **getPrimeRate** is shown here:

```
public double getPrimeRate(final Date date)
```

First, a variable is defined to hold the last rate found, which is the current rate. This variable is named **currentRate**.

```
double currentRate = 0;
```

Next, the method loops through all of the interest rates. These interest rates are stored in a sorted list, so the interest rates for the earliest dates occur first.

```
for (final InterestRate rate : this.rates) {
```

As soon as the first interest rate is found with a date beyond the date of interest, then the rate stored in **`currentRate`** is the interest rate for the specified date. If the variable **`currentRate`** has not yet been set, then the specified date is earlier than the dates for our data. If this is the case, then we have no interest rate data for the specified date and a value of **`null`** is returned.

```
  if (rate.getEffectiveDate().after(date)) {
      return currentRate;
  } else {
```

Otherwise, the specified date has not yet been reached, so the **`currentRate`** variable is updated.

```
      currentRate = rate.getRate();
  }
}
```

If we reach the end of the list, then the final interest rate is simply returned. We assume that the rate has not changed since our last data value and specified date. As long as our interest rate file is up to date, and the specified date is not in the future, this is a valid assumption.

```
return currentRate;
}
```

Since the **`getPrimeRate`** method must iterate to find the interest rate, calling it is somewhat expensive; therefore, each S&P 500 sample must be "stitched" to the correct interest rate.

Stitching the Rates to Ranges

The **`stitchInterestRates`** function is called to find the appropriate interest rate for each of the **`FinancialSample`** objects. The signature for the **`stitchInterestRates`** method is shown here:

```
public void stitchInterestRates()
```

We begin by looping through all the **`FinancialSample`** objects.

```
for (final FinancialSample sample : this.samples) {
```

For each **`FinancialSample`** object, we obtain the prime interest rate.

```
  final double rate = getPrimeRate(sample.getDate());
  sample.setRate(rate);
}
```

This process is continued until all the **`FinancialSample`** objects have been processed.

To train the neural network, input and ideal data must be created. The next two sections discuss how this is done.

Creating the Input Data

To create input data for the neural network, the **getInputDate** method of the **SP500Actual** class is used. The signature for the **getInputData** method is shown here:

```
public void getInputData(final int offset, final double[] input)
```

Two arguments are passed to the **getInputData** method. The **offset** argument specifies the zero-based index at which the input data is to be extracted. The **input** argument provides a **double** array into which the financial samples will be copied. This array also specifies the number of **FinancialSample** objects to process. Sufficient **FinancialSample** objects will be processed to fill the array.

First, an array of references to the samples is obtained.

```
final Object[] samplesArray = this.samples.toArray();
```

Next, we loop forward, according to the size of the **input** array.

```
for (int i = 0; i < this.inputSize; i++) {
```

Each **FinancialSample** object is then obtained.

```
  final FinancialSample sample = (FinancialSample)
samplesArray[offset
        + i];
```

Both the percent change and rate for each sample are copied. The neural network then uses these two values to make a prediction.

```
  input[i] = sample.getPercent();
  input[i + this.outputSize] = sample.getRate();
}
```

As you can see, the input to the neural network consists of percentage changes and the current level of the prime interest rate. Using the percentage changes is different than how input was handled for the neural network presented in chapter 9. In chapter 9, the actual numbers were added to the neural network. The program in this chapter will instead track percentage moves. In general, the S&P 500 has increased over its history, and has not often revisited ranges. Therefore, more patterns can be found by tracking the percent changes, rather than actual point values.

Creating the Ideal Output Data

For supervised training, the ideal outputs must also be calculated from known data. While the inputs include both interest rate and quote data, the outputs only contain quote percentage data. We are attempting to predict percentage movement in the S&P 500; we are not attempting to predict fluctuations in the prime interest rate.

The ideal output data is created by calling the **getOutputData** method. The signature for the **getOutputData** method is shown here:

```
public void getOutputData(final int offset, final double[] output)
```

Two arguments are passed to the **getInputData**. The **offset** argument specifies the zero-based index at which the output data is to be extracted. The **output** argument provides a **double** array into which the financial samples will be copied. This array also specifies the number of **FinancialSample** objects to be processed. Sufficient **FinancialSample** objects will be processed to fill the array.

First, an array of references to the samples is obtained.

```
final Object[] samplesArray = this.samples.toArray();
```

Next, we loop through the samples.

```
for (int i = 0; i < this.outputSize; i++) {
```

For each sample, we copy only the percentage change to the output array.

```
  final FinancialSample sample = (FinancialSample)
samplesArray[offset
        + this.inputSize + i];
  output[i] = sample.getPercent();
}
```

The prime interest rate is not copied, because the neural network is not trying to predict fluctuations in the prime interest rate. The neural network is only predicting fluctuations in the S&P 500 index.

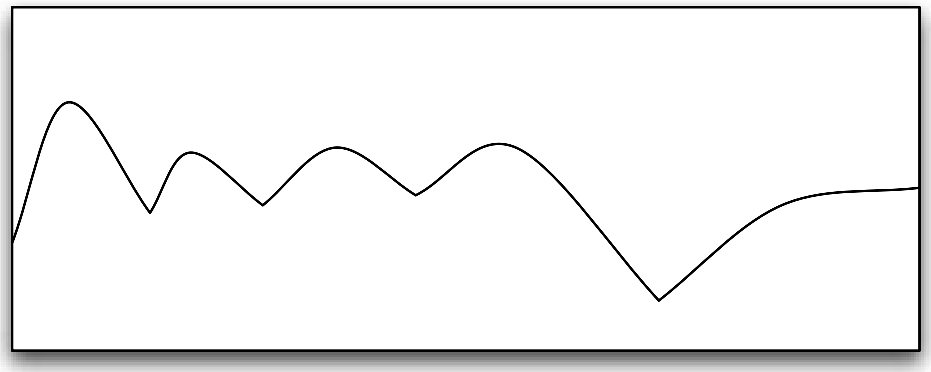
Training the S&P 500 Network

A combination of backpropagation and simulated annealing is used to train this neural network. A complex network such as this often benefits from a hybrid training approach.

Local Minima

The error for a backpropagation training algorithm has a tendency to encounter the local minima problem. Ideally, you should train until the global minimum is reached. However, the backpropagation algorithm can sometimes mistake a local minimum for the global minimum. Consider Figure 10.3.

Figure 10.3: Global and Local Minima



As you can see in Figure 10.3, there are four local minima. Only one of these local minima is the global minimum. To avoid the local minima problem, a hybrid training algorithm is used.

Hybrid Training

The main class for the S&P 500 prediction example is **`PredictSP500`**. This class should be executed to run this example. The **`PredictSP500`** class is shown in Listing 10.7:

Listing 10.7: Try to Predict the S&P 500 (PredictSP500.java)

```
package com.heatonresearch.book.introneuralnet.ch10.sp500;

import java.io.IOException;
import java.text.NumberFormat;
import java.util.Date;

import com.heatonresearch.book.introneuralnet.common.ReadCSV;
import com.heatonresearch.book.introneuralnet.neural.activation.
ActivationFunction;
import com.heatonresearch.book.introneuralnet.neural.activation.
```

```
ActivationTANH;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardLayer;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.Train;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.anneal.NeuralSimulatedAnnealing;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.backpropagation.Backpropagation;
import com.heatonresearch.book.introneuralnet.neural.util.
ErrorCalculation;
import com.heatonresearch.book.introneuralnet.neural.util.
SerializeObject;

/**
 * Chapter 10: Application to the Financial Markets
 *
 * PredictSP500: Attempt to predict the SP500 using a predictive
 * feedforward neural network.  The key word is "attempt".  The
 * neural network can guess some basic trends in the SP500, and is
 * meant only as a starting point.
 *
 * This class is in no way investment advice!
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class PredictSP500 {

      public final static int TRAINING_SIZE = 500;
      public final static int INPUT_SIZE = 10;
      public final static int OUTPUT_SIZE = 1;
      public final static int NEURONS_HIDDEN_1 = 20;
      public final static int NEURONS_HIDDEN_2 = 0;
      public final static double MAX_ERROR = 0.02;
      public final static Date PREDICT_FROM =
            ReadCSV.parseDate("2007-01-01");
      public final static Date LEARN_FROM =
            ReadCSV.parseDate("1980-01-01");

      public static void main(final String args[]) {
            final PredictSP500 predict = new PredictSP500();
            if (args.length > 0 &&
                  args[0].equalsIgnoreCase("full"))
```

```
                predict.run(true);
            else
                predict.run(false);
        }

        private double input[][];

        private double ideal[][];
        private FeedforwardNetwork network;

        private SP500Actual actual;

        public void createNetwork() {
            final ActivationFunction threshold =
                new ActivationTANH();
            this.network = new FeedforwardNetwork();
            this.network.addLayer(new FeedforwardLayer(threshold,
                    PredictSP500.INPUT_SIZE * 2));
            this.network.addLayer(new FeedforwardLayer(threshold,
                    PredictSP500.NEURONS_HIDDEN_1));
            if (PredictSP500.NEURONS_HIDDEN_2 > 0) {
                this.network.addLayer(
                    new FeedforwardLayer(threshold,
                        PredictSP500.NEURONS_HIDDEN_2));
            }
            this.network.addLayer(new FeedforwardLayer(threshold,
                    PredictSP500.OUTPUT_SIZE));
            this.network.reset();
        }

        public void display() {
            final NumberFormat percentFormat =
                NumberFormat.getPercentInstance();
            percentFormat.setMinimumFractionDigits(2);

            final double[] present = new double[INPUT_SIZE * 2];
            double[] predict = new double[OUTPUT_SIZE];
            final double[] actualOutput = new double[OUTPUT_SIZE];

            int index = 0;
            for (final FinancialSample sample :
                this.actual.getSamples()) {
                if (sample.getDate().after(
                    PredictSP500.PREDICT_FROM)) {
                    final StringBuilder str =
                    new StringBuilder();
```

```
                        str.append(ReadCSV.displayDate(
                        sample.getDate()));
                        str.append(":Start=");
                        str.append(sample.getAmount());

                        this.actual.getInputData(index -
                              INPUT_SIZE, present);
                        this.actual.getOutputData(index -
                              INPUT_SIZE, actualOutput);

                        predict =
                        this.network.computeOutputs(present);
                        str.append(",Actual % Change=");
                        str.append(
                        percentFormat.format(actualOutput[0]));
                        str.append(",Predicted % Change= ");
                        str.append(
                        percentFormat.format(predict[0]));

                        str.append(":Difference=");

                        final ErrorCalculation error =
                        new ErrorCalculation();
                        error.updateError(predict, actualOutput);
                        str.append(percentFormat.format(
                        error.calculateRMS()));

                        //

                        System.out.println(str.toString());
                  }

                  index++;
            }
      }

      private void generateTrainingSets() {
            this.input = new double[TRAINING_SIZE][INPUT_SIZE * 2];
            this.ideal = new double[TRAINING_SIZE][OUTPUT_SIZE];

            // find where we are starting from
            int startIndex = 0;
            for (final FinancialSample sample : this.actual.
                  getSamples()) {
                  if (sample.getDate().after(LEARN_FROM)) {
                        break;
            }
            startIndex++;
        }

        // create a sample factor across the training area
        final int eligibleSamples = TRAINING_SIZE - startIndex;
        if (eligibleSamples == 0) {
            System.out
                        .println(
            "Need an earlier date for LEARN_FROM or a "
            +"smaller number for TRAINING_SIZE.");
            System.exit(0);
        }
        final int factor = eligibleSamples / TRAINING_SIZE;

        // grab the actual training data from that point
        for (int i = 0; i < TRAINING_SIZE; i++) {
            this.actual.getInputData(startIndex +
            (i * factor), this.input[i]);
            this.actual.getOutputData(startIndex +
            (i * factor), this.ideal[i]);
        }
    }

    public void loadNeuralNetwork() throws IOException,
        ClassNotFoundException {
        this.network = (FeedforwardNetwork)
            SerializeObject.load("sp500.net");
    }

    public void run(boolean full) {
        try {
            this.actual = new SP500Actual(
            INPUT_SIZE, OUTPUT_SIZE);
            this.actual.load("sp500.csv", "prime.csv");

            System.out.println("Samples read: "
            + this.actual.size());

            if (full) {
                createNetwork();
                generateTrainingSets();

                trainNetworkBackprop();

                saveNeuralNetwork();
```

```
			} else {
				loadNeuralNetwork();
			}

			display();

		} catch (final Exception e) {
			e.printStackTrace();
		}
	}

	public void saveNeuralNetwork() throws IOException {
		SerializeObject.save("sp500.net", this.network);
	}

	private void trainNetworkBackprop() {
		final Train train = new Backpropagation(
		this.network, this.input,
				this.ideal, 0.00001, 0.1);
		double lastError = Double.MAX_VALUE;
		int epoch = 1;
		int lastAnneal = 0;

		do {
			train.iteration();
			double error = train.getError();

			System.out.println("Iteration(Backprop) #"
			+ epoch + " Error:"
					+ error);

			if( error>0.05 )
			{
				if( (lastAnneal>100)
	&& (error>lastError || Math.abs(error-lastError)<0.0001) )
				{
					trainNetworkAnneal();
					lastAnneal = 0;
				}
			}

			lastError = train.getError();
			epoch++;
			lastAnneal++;
		} while (train.getError() > MAX_ERROR);
	}
```

```
      private void trainNetworkAnneal() {
            System.out.println(
"Training with simulated annealing for 5 iterations");
            // train the neural network
            final NeuralSimulatedAnnealing train =
            new NeuralSimulatedAnnealing(
                        this.network, this.input, this.ideal,
                        10, 2, 100);

            int epoch = 1;

            for(int i=1;i<=5;i++) {
                  train.iteration();
                  System.out.println("Iteration(Anneal) #"
                  + epoch + " Error:"
                  + train.getError());
                  epoch++;
            }
      }
}
```

The **PredictSP500** class implements the hybrid training, as well as attempts to predict future S&P 500 values with the newly trained neural network. The hybrid training works by alternating between backpropagation training and simulated annealing. The training begins with backpropagation and switches to simulated annealing when backpropagation is no longer efficiently training the network. Once simulated annealing has been used for a number of cycles, the program switches back to backpropagation.

Listing 10.8 shows the output from hybrid training.

Listing 10.8: Hybrid Training Output

```
Iteration(Backprop) #1511 Error:0.1023912889542664
Iteration(Backprop) #1512 Error:0.10237590385794164
Iteration(Backprop) #1513 Error:0.10236112842990429
Iteration(Backprop) #1514 Error:0.10234696743296834
Iteration(Backprop) #1515 Error:0.10233342565770161
Iteration(Backprop) #1516 Error:0.10232050792236635
Iteration(Backprop) #1517 Error:0.10230821907285384
...
Iteration(Backprop) #1518 Error:0.10229656398261411
Iteration(Backprop) #1519 Error:0.10228554755257999
Iteration(Backprop) #1520 Error:0.10227517471108645
Iteration(Backprop) #1521 Error:0.10226545041378378 Train-
ing with simulated annealing for 5 iterations Iteration(Anneal)
```

```
#1 Error:0.042124954651261835 Iteration(Anneal) #2 Er-
ror:0.042124954651261835 Iteration(Anneal) #3 Er-
ror:0.042124954651261835 Iteration(Anneal) #4
Error:0.042124954651261835 Iteration(Anneal) #5 Er-
ror:0.042124954651261835 Iteration(Backprop) #1522 Er-
ror:0.04137291563937421 Iteration(Backprop) #1523
Error:0.04079595880076687 Iteration(Backprop) #1524 Er-
ror:0.04031691771522145 Iteration(Backprop) #1525 Er-
ror:0.03987729279067175 Iteration(Backprop) #1526
Error:0.03945727030649545 Iteration(Backprop) #1527 Er-
ror:0.03905037926667383 Iteration(Backprop) #1528 Er-
ror:0.038654238218587864 Iteration(Backprop) #1529
Error:0.038267815849145556 Iteration(Backprop) #1530 Er-
ror:0.037890572580577805 Iteration(Backprop) #1531 Er-
ror:0.03752216093273823 Iteration(Backprop) #1532
Error:0.03716231245175569 Iteration(Backprop) #1533 Er-
ror:0.036810793350444744
...
```

As you can see from the above code, backpropagation is used through iteration 1,521. The improvement between iterations 1,520 and 1,521 was not sufficient, so simulated annealing was employed for five iterations. Before the simulated annealing was used, the error rate was around 10%. After the simulated annealing, the error rate dropped rapidly to around 4%. Simulated annealing was successful in avoiding the local minimum that the above training session was approaching.

The hybrid training algorithm is implemented in the **trainNeuralNetworkHybrid** method. The signature for the **trainNeuralNetworkHybrid** method is shown here:

```
private void trainNetworkHybrid()
```

The hybrid training begins just like a regular backpropagation training session. A backpropagation trainer is implemented with a low training rate and a low momentum.

```
final Train train = new Backpropagation(this.network, this.input,
      this.ideal, 0.00001, 0.1);
```

We keep track of the last error, so we can gauge the performance of the training algorithm. Initially, this last error value is set very high so that it will be properly initialized during the first iteration.

```
double lastError = Double.MAX_VALUE;
int epoch = 1;
```

We only use simulated annealing every 100 iterations; otherwise, as the improvements become very small towards the end of the training, simulated annealing would constantly be invoked. We use the **`lastAnneal`** variable to track how many epochs it has been since the last simulated annealing attempt.

```
int lastAnneal = 0;

do {
  train.iteration();
  double error = train.getError();
```

For every epoch, we update the progress.

```
  System.out.println("Iteration(Backprop) #" + epoch + " Error:"
        + error);
```

If the error is greater than 5%, then we will consider using simulated annealing. Once the error is less than 5%, it is usually best to just let backpropagation finish out the training.

```
  if( error>0.05 )
  {
```

We must now consider if we would like to use simulated annealing. If it has been 100 iterations since we last used simulated annealing and the error rate has not improved by one hundredth of a percent, then we will try to train the neural network using simulated annealing.

```
      if( (lastAnneal>100) && (error>lastError ||
            Math.abs(error-lastError)<0.0001) )
      {
```

To train using simulated annealing, we call the **`trainNetworkAnneal`** method. This method works very much like our previous examples of simulated annealing, so it will not be repeated here. After the simulated annealing training has completed, the **`lastAnneal`** variable is set to zero so that we can once again keep track of how many epochs it has been since simulated annealing training was last used.

```
        trainNetworkAnneal();
        lastAnneal = 0;
      }
  }
```

We keep track of the last error and the epoch number.

```
  lastError = train.getError();
  epoch++;
  lastAnneal++;
} while (train.getError() > MAX_ERROR);
```

The **MAX_ERROR** constant for this example is set to 2%. It is possible to train this example to less than 1%, but it takes nearly one million epochs and several days of training.

Attempting to Predict the S&P 500

This example uses the **predict** method to attempt to predict the S&P 500. The signature for the **predict** method is shown here:

```
public void predict()
```

First, we create a **NumberFormat** object designed to format percentages.

```
final NumberFormat percentFormat = NumberFormat.getPercentIn-
stance();
percentFormat.setMinimumFractionDigits(2);
```

Three arrays are created. The **present** array holds the "present values" upon which the prediction will be based. The **predict** array will hold the predicted S&P 500 values. The **actualOutput** array will contain the actual values from the historical S&P 500 data. The **actualOutput** array will be compared against the **predicted** array to determine the effectiveness of the neural network.

```
final double[] present = new double[INPUT_SIZE * 2];
double[] predict = new double[OUTPUT_SIZE];
final double[] actualOutput = new double[OUTPUT_SIZE];
```

We loop through all of the **FinancialSample** objects that fall in the range for which this prediction is to be made.

```
int index = 0;
for (final FinancialSample sample : this.actual.getSamples()) {
```

If this **FinancialSample** object falls in the range after the **PREDICT_FROM** constant, then we should attempt to predict based on it.

```
  if (sample.getDate().after(PredictSP500.PREDICT_FROM)) {
```

We create a **StringBuilder** that will build the line of text to be displayed and append the date on which the sample was taken.

```
      final StringBuilder str = new StringBuilder();
      str.append(ReadCSV.displayDate(sample.getDate()));
```

The starting value of the S&P 500 for this time slice is then displayed.

```
      str.append(":Start=");
      str.append(sample.getAmount());
```

The input values for the neural network are obtained, as well as the ideal output values.

```
      this.actual.getInputData(index - INPUT_SIZE, present);
      this.actual.getOutputData(index - INPUT_SIZE, actualOutput);
```

The outputs are then computed using the present values. This is the neural network's prediction.

```
      predict = this.network.computeOutputs(present);
```

The actual change in percent for the S&P 500 is displayed.

```
      str.append(",Actual % Change=");
      str.append(percentFormat.format(actualOutput[0]));
```

Next, the predicted change in percent is displayed.

```
      str.append(",Predicted % Change= ");
      str.append(percentFormat.format(predict[0]));
```

The difference between the actual and predicted values is then displayed.

```
      str.append(":Difference=");
```

The error between the actual and the difference is then calculated using root mean square.

```
      final ErrorCalculation error = new ErrorCalculation();
      error.updateError(predict, actualOutput);
      str.append(percentFormat.format(error.calculateRMS()));
```

Finally, the **StringBuffer** is displayed.

```
      System.out.println(str.toString());
  }

  index++;
}
```

This same procedure is followed for every **FinancialSample** object provided.

This example serves as a basic introduction to financial prediction with neural networks. An entire book could easily be written about how to use neural networks with financial markets. There are many options available that will allow you to create more advanced financial neural networks. For example, additional inputs can be provided; individual stocks, and their relations to other stocks can be used; and hybrid approaches using neural networks and other forms of statistical analyses can be used. This example serves as a starting point.

Chapter Summary

Predicting the movement of financial markets is a very common area of interest for predictive neural networks. Application of neural networks to financial forecasting could easily fill a book. This book provides a brief introduction by presenting the basics of how to construct a neural network that attempts to predict price movement in the S&P 500 index.

To attempt to predict the S&P 500 index, both the prime interest rate and previous values of the S&P 500 are used. This attempts to find trends in the S&P 500 data that might be used to predict future price movement.

This chapter also introduced hybrid training. The hybrid training algorithm used in this chapter made use of both backpropgation and simulated annealing. Backpropagation is used until the backpropagation no longer produces a satisfactory reduction in the error rate. At this time, simulated annealing is used to help free the neural network from what might be a local minimum. A local minimum is a low point on the training chart, but not necessarily the lowest point. Backpropagation has a tendency to get stuck at a local minimum.

Though the feedforward neural network is one of the most common forms of neural networks, there are other neural network architectures that are also worth considering. In the next chapter, you will be introduced to a self-organizing map. A self-organizing map is often used to classify input into groups.

Vocabulary

Comma Separated Value (CSV) File

Global Minimum

Hybrid Training

Local Minima

Prime Interest Rate

S&P 500

Sample

Questions for Review

1. This chapter explained how to use simulated annealing and backpropagation to form a hybrid training algorithm. How can a genetic algorithm be easily added to the mix? A genetic algorithm uses many randomly generated neural networks. How can this be used without discarding the previous work done by the backpropagation and simulated annealing algorithms?

2. You would like to create a predictive neural network that predicts the price of an individual stock for the next five days. You will use the stock's prices from the previous ten days and the current prime interest rate. How many input neurons and how many output neurons will be used?

3. Explain what a local minimum is and how it can be detrimental to neural network training. How can this be overcome?

4. How can simulated annealing be used to augment backpropagation? How does the hybrid training algorithm presented in this chapter know when to engage simulated annealing?

5. Why is it preferable to input percentage changes into financial neural network changes rather than actual stock prices? Which activation function would work well for percentage changes? Why?