CHAPTER 2: MATRIX OPERATIONS

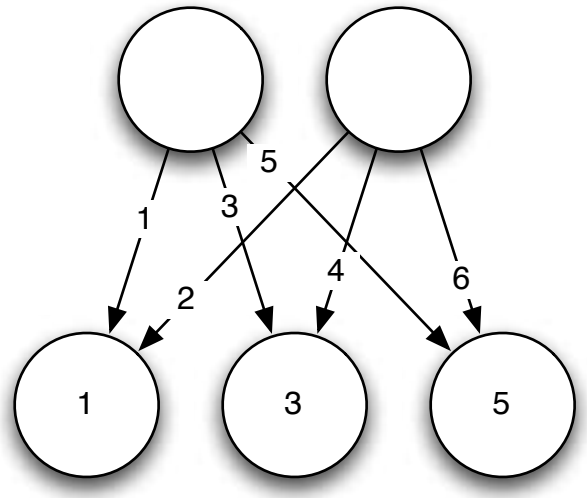
- Understanding Weight Matrixes
- Using the Matrix Classes
- Using Matrixes with Neural Networks
- Working with Bipolar Operations

Matrix mathematics are used to both train neural networks and calculate their outputs. Other mathematical operations are used as well; however, neural network programming is based primarily on matrix operations. This chapter will review the matrix operations that are of particular use to neural networks. Several classes will be developed to encapsulate the matrix operations used by the neural networks covered in this book. You will learn how to construct these matrix classes and how to use them. Future chapters will explain how to use the matrix classes with several different types of neural networks.

The Weight Matrix

In the last chapter, you learned that neural networks make use of two types of values: weights and thresholds. Weights define the interactions between the neurons. Thresholds define what it will take to get a neuron to fire. The weighted connections between neurons can be thought of as a matrix. For example, consider the connections between the following two layers of the neural network shown in Figure 2.1.

Figure 2.1: A two neuron layer connected to a three neuron layer.



You can see the weights in Figure 2.1. The weights are attached to the lines drawn between the neurons. Each of the two neurons in the first layer is connected to each of the three neurons in the second layer. There are a total of six connections. These connections can be represented as a 3x2 weight matrix, as described in Equation 2.1.

Equation 2.1: A Weight Matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

The weight matrix can be defined in Java as follows:

```
Matrix weightMatrix = new Matrix(3,2);
```

The threshold variable is not multidimensional, like the weight matrix. There is one threshold value per neuron. Each neuron in the second layer has an individual threshold value. These values can be stored in an array of Java **`double`** variables. The following code shows how the entire memory of the two layers can be defined.

```
Matrix weightMatrix = new Matrix(3,2);
double thresholds[] = new double[2];
```

These declarations include both the 3x2 matrix and the two threshold values for the second layer. There is no need to store threshold values for the first layer, since it is not connected to another layer. Weight matrix and threshold values are only stored for the connections between two layers, not for each layer.

The preferred method for storing these values is to combine the thresholds with the weights in a combined matrix. The above matrix has three rows and two columns. The thresholds can be thought of as the fourth row of the weight matrix, which can be defined as follows:

```
Matrix weightMatrix = new Matrix(4,2);
```

The combined threshold and weight matrix is described in Equation 2.2. In this equation, the variable **`w represents`** the cells used to store weights and the variable **`t`** represents the cells used to hold thresholds.

Equation 2.2: A Threshold and Weight Matrix

$$\begin{bmatrix} w & w \\ w & w \\ w & w \\ t & t \end{bmatrix}$$

Combining the thresholds and weights in one matrix has several advantages. This matrix now represents the entire memory of this layer of the neural network and you only have to deal with a single structure. Further, since many of the same mathematical operations performed on the weight matrix are also performed on the threshold values, having them contained in a single matrix allows these operations to be performed more efficiently.

Matrix Classes

This chapter presents several classes that can be used to create and manipulate matrixes. These matrix classes will be used throughout this book. These classes are summarized in Table 2.1.

Table 2.1: Matrix Classes

Class	Purpose
BiPolarUtil	A utility class to convert between Boolean and bipolar numbers.
Matrix	Holds a matrix.
MatrixMath	Performs mathematical operations on a matrix.

The next three sections will examine each of these classes.

The BiPolarUtil Class

The **`BiPolarUtil`** class is used to switch between a bipolar number and a **`boolean`** value. A **`boolean`** value is either **`true`** or **`false`**. A bipolar number is either 1 or -1. Using this class, the **`boolean`** value of **`false`** is expressed as a -1 bipolar value and the **`boolean`** value of **`true`** is expressed as a 1 bipolar value. The **`BiPolarUtil`** class is a collection of static methods. The signatures for these methods are shown here.

```
public static double bipolar2double(final boolean b)
public static double[] bipolar2double(final boolean b[])
public static double[][] bipolar2double(final boolean b[][])
public static boolean double2bipolar(final double d)
public static boolean[] double2bipolar(final double d[])
public static boolean[][] double2bipolar(final double d[][])
```

Table 2.2 summarizes the functions provided by the **`BiPolarUtil`** class.

Table 2.2: The BiPolarUtil Class

Method	Purpose
bipolar2double	Converts a Boolean bipolar to a double. For example, true is converted to 1.
double2bipolar	Converts a double value to a bipolar Boolean. For example, -1 is converted to false.

The above two methods are overloaded, so you can convert a single value, a single dimensional array, or a two dimensional array. Bipolar values are particularly useful for Hopfield neural networks. Hopfield neural networks will be discussed in the next chapter.

The Matrix Class

The **Matrix** class is used to construct two dimensional matrixes. The values contained in the matrixes are stored as Java **double** variables. The **Matrix** class provides the fundamental operations of numerical linear algebra. For operations involving two or more matrixes, the **MatrixMath** class is used. The **MatrixMath** class is discussed in the next section.

The signatures for the **Matrix** class members are shown here.

```
public static Matrix createColumnMatrix(final double input[])
public static Matrix createRowMatrix(final double input[])
public void add(final int row, final int col, final double value)
public void clear()
public Matrix clone()
public boolean equals(final Matrix matrix)
public boolean equals(final Matrix matrix, int precision)
public double get(final int row, final int col)
public Matrix getCol(final int col)
public int getCols()
public Matrix getRow(final int row)
public int getRows()
public boolean isVector()
public boolean isZero()
public void set(final int row, final int col, final double value)
public double sum()
public double[] toPackedArray()
```

The methods provided by the **Matrix** class are summarized in Table 2.3.

Table 2.3: The Matrix Class

Method	Purpose
createColumnMatrix	Static method which creates a matrix with a single column.
createRowMatrix	Static method which creates a matrix with a single row.
add	Adds the specified value to every cell in the matrix.
clear	Sets every cell in a matrix to zero.
clone	Creates an exact copy of a matrix.
equals	Determines if two matrixes are equal to each other.
get	Gets the value for a cell.
getCol	Gets one column of a matrix object as a new matrix object.
getCols	Determines the number of columns in a matrix object.
getRow	Gets one row of a matrix object as a new matrix object.
getRows	Determines the number of rows in a matrix object.
isVector	Determines if a matrix is a vector. A vector matrix has either a single row or a single column.
isZero	Determines if every cell in a matrix object is zero.
set	Sets the value of a cell.
sum	Returns the sum of every cell in a matrix object.
toPackedArray	Converts a two dimensional matrix array into a one dimensional array of Java double variables.

The **`Matrix`** class will be used to construct the weight matrixes for all neural networks presented in this book.

The MatrixMath Class

Most mathematical operations on a **`Matrix`** class are accomplished using the **`MatrixMath`** class. All methods in the **`MatrixMath`** class are **`static`**. Further, they always return a new matrix and do not modify the matrixes passed to them. The signatures for the **`MatrixMath`** methods are shown here.

```
public static Matrix add(final Matrix a, final Matrix b)
public static Matrix divide(final Matrix a, final double b)
public static double dotProduct(final Matrix a, final Matrix b)
public static Matrix identity(final int size)
public static Matrix multiply(final Matrix a, final double b)
public static Matrix multiply(final Matrix a, final Matrix b)
public static Matrix subtract(final Matrix a, final Matrix b)
public static Matrix transpose(final Matrix input)
public static double vectorLength(final Matrix input)
```

These methods are summarized in Table 2.4.

Table 2.4: The MatrixMath Class

Method	Purpose
add	Adds two matrixes and produces a third matrix.
divide	Divides one matrix by a scalar and produces a second matrix.
dotProduct	Calculates the dot product of a matrix.
identity	Creates an identity matrix of a specified size.
multiply	Multiplies one matrix by another and produces a third matrix.
subtract	Subtracts one matrix from another and produces a third matrix.
transpose	Transposes a matrix and produces a new matrix.
vectorLength	Calculates the squared length of a vector.

These are the primary mathematical operations that neural networks need to perform. Each of these operations will be discussed at length later in this chapter.

Many Java neural network implementations build matrix operations directly into their neural network classes. The result being many nested **for** loops inside the neural network class. For example, the following code allows a neural network to learn.

```
public void learn(double learnRate, double momentum) {

  if (layer.hasMatrix() ) {
    for (int i1 = 0; i1 < layer.getNeuronCount(); i1++) {
      for (int i2 = 0; i2 < layer.getNext().getNeuronCount();
i2++) {
matrixDelta[i1][i2] = (learnRate * accMatrixDelta[i1][i2])
+ (momentum * matrixDelta[i1][i2]);
        layer.getMatrix().setMatrix(i1,i2,layer.getMatrix().
getMatrix(i1,i2) + matrixDelta[i1][i2]);
        accMatrixDelta[i1][i2] = 0;
      }
    }
  }
}
```

The above code performs several matrix operations; however, it is not completely obvious which matrix operations are being performed. By encapsulating the matrix operations inside several matrix classes, the above code can be simplified. Further, you will be able to tell, at a glance, which matrix operations are being performed. The following code accomplishes the same as the code above; however, it uses matrix classes.

```
public void learn(final double learnRate, final double momentum)
{
  if (this.layer.hasMatrix()) {

    Matrix m1 = MatrixMath.multiply( accMatrixDelta, learnRate );
    Matrix m2 = MatrixMath.multiply( matrixDelta, momentum);
    matrixDelta = MatrixMath.add(m1, m2);
    layer.setMatrix(MatrixMath.add(layer.
getMatrix(),matrixDelta));
    accMatrixDelta.clear();
  }
}
```

As you can see, several matrixes are constructed and then the **MatrixMath** class is used to perform operations upon them.

Constructing a Matrix

There are several ways to construct a **Matrix** object. To construct an empty matrix, use the following code:

```
Matrix matrix = new Matrix(3,2);
```

This will construct an empty matrix of three rows and two columns, which contains only zeros. This matrix is described in Equation 2.3.

Equation 2.3: An Empty Matrix

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

You can also construct a matrix using a two dimensional array, which allows you to initialize the cells of the matrix in the declaration. The following code creates an initialized 3x2 matrix.

```
double matrixData[][] = {
  {1.0,2.0,3.0},
  {4.0,5.0,6.0}
};

Matrix matrix = new Matrix(matrixData);
```

This matrix is described in Equation 2.4.

Equation 2.4: An Initialized Matrix

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 7.0 & 8.0 \end{bmatrix}$$

Matrixes can also be created using a single dimensional array. A single dimensional array can be used to create a row or a column matrix. Row and column matrixes contain either a single row or a single column, respectively. These matrixes are also called vectors. The following code can be used to create a row matrix.

```
double matrixData[] = {1.0,2.0,3.0,4.0};
Matrix matrix = Matrix.createRowMatrix(matrixData);
```

This will create a matrix that contains a single row. This matrix is described in Equation 2.5.

Equation 2.5: A Row Matrix/Vector

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$

It is also possible to create a column matrix. This matrix takes a single dimensional array, just as before; however, this time a matrix with a single column is created. The code to do this is shown here.

```
double matrixData[] = {1.0,2.0,3.0,4.0};
Matrix matrix = Matrix.createColumnMatrix(matrixData);
```

This matrix is described in Equation 2.6.

Equation 2.6: A Column Matrix/Vector

$$\begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix}$$

In the next section you will see how mathematical operations can be performed on one or more matrixes.

Matrix Operations

This section will review some of the matrix operations that are provided by the **MatrixMath** class. Each matrix operation will be explained, as well as the code that is used to perform the operation. You can try some of the matrix operations online at the following URL.

http://www.heatonresearch.com/examples/math/matrix

Matrix Addition

Matrix addition is a relatively simple procedure. The corresponding cells of two matrixes of exactly the same size are summed. The results are returned in a new matrix of equal size. Equation 2.7 describes the process of matrix addition.

Equation 2.7: Matrix Addition

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}+\begin{bmatrix}4 & 5\\6 & 7\end{bmatrix}=\begin{bmatrix}5 & 7\\9 & 11\end{bmatrix}$$

The signature for the **add** method of the **MatrixMath** class is shown here.

```
public static Matrix add(final Matrix a, final Matrix b)
```

The process begins with the creation of a new array to hold the results of the operation.

```
final double result[][] = new double[a.getRows()][a.getCols()];
```

Next, we loop through the cells of each row and column in the source matrixes.

```
for (int resultRow = 0; resultRow < a.getRows(); resultRow++) {
  for (int resultCol = 0; resultCol < a.getCols(); resultCol++) {
```

The result of each summation is then placed in the corresponding cell in the **result** array.

```
    result[resultRow][resultCol] = a.get(resultRow, resultCol)
    + b.get(resultRow, resultCol);
  }
}
```

Finally, a new matrix is created from the **result** array.

```
return new Matrix(result);
```

This **add** method can be used to add any two matrixes of the same size.

Matrix Division by a Scalar

A matrix can be divided by a single number, or scalar. For example, to divide a matrix by the number 10, each cell in the matrix would be divided by 10. Equation 2.8 describes the process of dividing a matrix by a scalar.

Equation 2.8: Matrix Division by a Scalar

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} / 2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The signature for the matrix **divide** function is shown here.

```
public static Matrix divide(final Matrix a, final double b)
```

First, a **result** array is created to hold the results of the division. This array is exactly the same size as the array to be divided.

```
final double result[][] = new double[a.getRows()][a.getCols()];
```

To perform the division, the **divide** method must loop through every cell in the matrix.

```
for (int row = 0; row < a.getRows(); row++) {
  for (int col = 0; col < a.getCols(); col++) {
```

Every cell in the matrix is divided by the specified number and placed into the **result** array.

```
result[row][col] = a.get(row, col) / b;
}
}
```

Finally, a new **Matrix** object is created from the **result** array.

```
return new Matrix(result);
```

This matrix is returned. It is the result of the division.

Compute the Dot Product

The dot product can be computed from two vector matrixes. A vector matrix contains either a single row or a single column. To determine the dot product of two matrixes, they must have the same number of cells. It is not necessary that the cells in the two matrixes be oriented the same way. It is only necessary that they have the same number of cells.

The dot product is a scalar, a single number, not another matrix. To calculate the dot product, each cell, taken in order and beginning at the top-left side of the matrix, is multiplied by the corresponding cell in the other matrix. The results of these multiplication operations are then summed. Equation 2.9 describes the process of computing the dot product.

Equation 2.9: Dot Product

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} * \begin{bmatrix}5\\6\\7\\8\end{bmatrix} = (1*5)+(2*6)+(3*7)+(4*8) = 5+12+21+32 = 70$$

The signature for the **dotProduct** method is shown here.

```
public static double dotProduct(final Matrix a, final Matrix b)
```

First, two temporary arrays are created to hold the values of the packed matrix. The packed matrix is a simple one dimensional array that holds both dimensions, so the matrix is a flat linear list.

```
final double aArray[] = a.toPackedArray();
final double bArray[] = b.toPackedArray();
```

A **result** variable is declared to hold the sum of the dot product, as it will be computed.

```
double result = 0;
final int length = aArray.length;
```

The method then loops through the corresponding cells of the two matrixes, adding the result of each multiplication operation to the **result variable**.

```
for (int i = 0; i < length; i++) {
  result += aArray[i] * bArray[i];
}
```

Finally, the resulting number is returned.

```
return result;
```

This **result** variable is assigned the dot product of the two matrixes.

Matrix Multiplication and the Identity Matrix

Matrix multiplication can only be performed if two matrixes have compatible dimensions. Compatible dimensions mean that the number of columns of the first matrix must be equal to the number of rows of the second matrix. This means that it is legal to multiply a 2x3 matrix by a 3x2 matrix. However, it is not legal to multiply a 2x3 matrix by a 2x6 matrix!

Next, we will see how to multiply two matrixes. Equation 2.10 describes how a 2x3 matrix is multiplied by a 3x2 matrix.

Equation 2.10: Matrix Multiplication

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} (1*7)+(4*10) & (1*8)+(4*11) & (1*9)+(4*12) \\ (2*7)+(5*10) & (2*8)+(5*11) & (2*9)+(5*12) \\ (3*7)+(6*10) & (3*8)+(6*11) & (3*9)+(6*12) \end{bmatrix} = \begin{bmatrix} 47 & 52 & 57 \\ 64 & 71 & 78 \\ 81 & 90 & 99 \end{bmatrix}$$

It is also important to note that matrix multiplication is not commutative. The result of 2*6 is the same as 6*2. This is because multiplication is commutative when dealing with scalars—not so with matrixes. The result of multiplying a 1x3 matrix by a 3x2 matrix is not at all the same as multiplying a 3x2 matrix by a 1x3 matrix. In fact, it is not even valid to multiply a 3x2 matrix by a 1x3. Recall in Equation 2.10 we multiplied a 2x3 matrix by a 3x2 matrix. Equation 2.11 illustrates the results of multiplying a 3x2 matrix by a 2x3 matrix. This operation produces a completely different result than Equation 2.10.

Equation 2.11: Non-Commutative Matrix Multiplication

$$\begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} * \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} (7*1)+(8*2)+(9*3) & (7*4)+(8*5)+(9*6) \\ (10*1)+(11*2)+(12*3) & (10*4)+(11*5)+(12*6) \end{bmatrix} = \begin{bmatrix} 50 & 122 \\ 68 & 167 \end{bmatrix}$$

An identity matrix is a matrix that when multiplied by another matrix produces the same matrix. Think of this as multiplying a number by 1. Equation 2.12 describes the identity matrix.

Equation 2.12: Identity Matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

An identity matrix is always perfectly square. A matrix that is not square does not have an identity matrix. As you can see from Equation 2.12, the identity matrix is created by starting with a matrix that has only zero values. The cells in the diagonal from the northwest corner to the southeast corner are then set to one.

Equation 2.13 describes an identity matrix being multiplied by another matrix.

Equation 2.13: Multiply by an Identity Matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (1*1)+(2*0)+(3*0) & (1*0)+(2*1)+(3*0) & (1*0)+(2*0)+(3*1) \\ (4*1)+(5*0)+(6*0) & (4*0)+(5*1)+(6*0) & (4*0)+(5*0)+(6*1) \\ (7*1)+(8*0)+(9*0) & (7*0)+(8*1)+(9*0) & (7*0)+(8*0)+(9*1) \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The resulting matrix in Equation 2.13 is the same as the matrix that was multiplied by the identity matrix.

The signature for the **identity** method is shown here.

```
public static Matrix identity(final int size)
```

This method will create an identity matrix of the size specified by the **size** parameter. First, a new matrix is created that corresponds to the specified size.

```
final Matrix result = new Matrix(size, size);
```

Next, a for loop is used to set the northwest to southeast diagonal to one.

```
for (int i = 0; i < size; i++) {
  result.set(i, i, 1);
}
```

Finally, the resulting identity matrix is returned.

```
return result;
```

The method returns the **result**.

Matrix Multiplication by a Scalar

Matrixes can also be multiplied by a scalar. Matrix multiplication by a scalar is very simple to perform—every cell in the matrix is multiplied by the specified scalar. Equation 2.14 shows how this is done.

Equation 2.14: Matrix Multiplication by a Scalar

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * 2 = \begin{bmatrix} 1*2 & 2*2 & 3*2 \\ 4*2 & 5*2 & 6*2 \\ 7*2 & 8*2 & 9*2 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

The signature for the **`multiply`** by a scalar method is shown here.

```
public static Matrix multiply(final Matrix a, final double b)
```

First, a **`result`** array is created to hold the results of the multiplication operation.

```
final double result[][] = new   double[a.getRows()][a.getCols()];
```

The **`multiply`** method then loops through every cell in the original array, multiplies it by the scalar and then stores the result in the new **`result`** array.

```
  for (int row = 0; row < a.getRows(); row++) {
    for (int col = 0; col < a.getCols(); col++) {
      result[row][col] = a.get(row, col) * b;
    }
  }
```

Finally, a new **`Matrix`** object is created from the result array.

```
return new Matrix(result);
```

This **`Matrix`** object is then returned to the calling method.

Matrix Subtraction

Matrix subtraction is a relatively simple procedure, also. The two matrixes on which the subtraction operation will be performed must be exactly the same size. Each cell in the resulting matrix is the difference of the two corresponding cells from the source matrixes. Equation 2.15 describes the process of matrix subtraction.

Equation 2.15: Matrix Subtraction

$$\begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} (4-1) & (5-2) \\ (6-3) & (7-4) \end{bmatrix} = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$$

The signature for the **subtract** method of the **MatrixMath** class is shown here.

```
public static Matrix subtract(final Matrix a, final Matrix b)
```

First, a new array is created to hold the results of the subtraction operations.

```
final double result[][] = new double[a.getRows()][a.getCols()];
```

Next, we must loop through each row and column in the source matrixes.

```
for (int resultRow = 0; resultRow < a.getRows(); resultRow++) {
  for (int resultCol = 0; resultCol < a.getCols(); resultCol++) {
```

The results of the subtraction operation for a particular pair of cells in the source matrixes are placed in the corresponding cell in the **result** array.

```
    result[resultRow][resultCol] = a.get(resultRow, resultCol)
    - b.get(resultRow, resultCol);
  }
}
```

Finally, a new matrix is created from the **result** array.

```
return new Matrix(result);
```

Transpose a Matrix

Matrix transposition occurs when the rows and columns of a matrix are interchanged. Equation 2.16 describes the transposition of a matrix.

Equation 2.16: Matrix Transpose

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

The signature for the **transpose** method is shown here.

```
public static Matrix transpose(final Matrix input)
```

A new **inverseMatrix** array is created with rows and columns equal in size to the inverse of those of the original matrix.

```
final double inverseMatrix[][] = new double[input.getCols()][input
                        .getRows()];
```

Next, we loop through all of the cells in the original matrix. The value of each cell is copied to the location in the result array identified by the inverse row and column of the original cell.

```
  for (int r = 0; r < input.getRows(); r++) {
    for (int c = 0; c < input.getCols(); c++) {
      inverseMatrix[c][r] = input.get(r, c);
    }
  }
```

Finally, a new **Matrix** object is created from the **inverseMatrix** array.

```
return new Matrix(inverseMatrix);
```

This newly created **Matrix** object is returned to the calling method.

Vector Length

The length of a vector matrix is defined to be the square root of the squared sums of every cell in the matrix. Equation 2.17 describes how the vector length for a vector matrix is calculated.

Equation 2.17: Calculate Vector Length

$$[x_1 \quad x_2 \quad \ldots \quad x_n] = \sqrt{(x_1)^2 + (x_2)^2 + \ldots + (x_n)^2}$$

The **MatrixMath** class provides the **vectorLength** function which can be used to calculate this length. The signature for the **vectorLength** function is shown here.

```
public  static double vectorLength( final Matrix input )
```

First, an array of the individual cells is created using the **toPackedArray** function. This function returns the matrix as a simple array of scalars. This allows either a column or row-based vector matrix length to be calculated, since they will both become simple arrays of scalars.

```
double v[] = input.toPackedArray();
double rtn = 0.0 ;
```

Next, we loop through the packed array and sum the square of every number.

```
for ( int i=0;i<v.length;i++ ) {
  rtn += Math.pow(v[i],2);
}
```

Finally, the square root of the sum is returned.

```
return Math.sqrt(rtn);
```

The length of a vector will be particularly important for the self-organizing maps that will be presented in chapter 8.

Bipolar Operations

In binary format, **`true`** is represented with the number one and **`false`** is represented with zero. Bipolar notation is another way to represent binary states. In bipolar notation **`true`** is represented by one and **`false`** is represented by negative one.

Equation 2.18 describes how a Boolean number is converted into a bipolar number.

Equation 2.18: Boolean to Bipolar

$$f(x)=2x-1$$

Equation 2.19 does the opposite. This equation shows how to convert a bipolar number into a **`boolean`** number.

Equation 2.19: Bipolar to Boolean

$$f(x)=\frac{(x+1)}{2}$$

Neural networks that operate on **`boolean`** numbers will usually require these **`boolean`** values to be expressed as bipolar numbers. To assist with this conversion, the **`BiPolarUtil`** class is provided. Table 2.2 summarized the functions provided by the **`BiPolarUtil`** class.

The following code uses the **`BiPolarUtil`** class to construct a bipolar matrix.

```
boolean booleanData2[][] = {{true,false},{false,true}};

Matrix matrix2 = new Matrix(BiPolarUtil.
```

```
bipolar2double(booleanData2));
```

This code will produce the matrix described in Equation 2.20.

Equation 2.20: A Bipolar Matrix

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

This will be particularly useful in the next chapter in which a Hopfield neural network is presented. A Hopfield neural network makes use of bipolar matrixes.

Chapter Summary

Matrix mathematics is very important to neural networks. Java does not include built-in support for matrixes; thus, three matrix classes have been presented here. These matrix classes are used to both create matrixes and perform various mathematical operations on them.

Bipolar notation is a special way of representing binary values. In bipolar notation, the binary value of **`true`** is represented with a one, and the binary value of **`false`** is represented with a negative one. Most neural networks that work with binary values will make use of bipolar notation. The Hopfield neural network, which will be introduced in chapter 3, makes use of bipolar numbers.

This book will make use of the matrix classes presented in this chapter for all neural networks that will be introduced. This will allow you to quickly see the mathematical underpinnings of each neural network.

Vocabulary

Bipolar

Boolean

Column Matrix

Dot Product

Identity Matrix

Matrix

Row Matrix

Scalar

Vector

Weight Matrix

Questions for Review

1. What is the purpose of using bipolar numbers, as opposed to Boolean numbers?

2. Is matrix multiplication commutative?

3. What are the dimensions of a weight matrix used to connect a two neuron layer to a three neuron layer?

4. Perform the following multiplication.

$$\begin{bmatrix} 3 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = ?$$

5. Perform the following dot product.

$$\begin{bmatrix} 6 & 2 & 7 & 4 \end{bmatrix} \bullet \begin{bmatrix} 6 \\ 6 \\ 7 \\ 9 \end{bmatrix} = ?$$