**CHAPTER**  **2:**  **MATRIX**  **OPERACIONES**

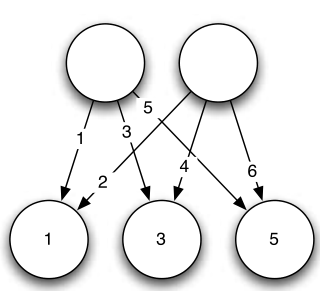
* Comprensión de las matrices de peso
* Uso de las clases matrix
* Uso de matrices con redes neuronales
* Trabajar con operaciones bipolares

Las matemáticas de matriz se utilizan tanto para entrenar redes neuronales como para calcular sus salidas. También se utilizan otras operaciones matemáticas; sin embargo, la programación de redes neuronales se basa principalmente en operaciones de matriz. Este capítulo revisará la matriz operationes que son de particular utilidad para las redes neuronales. Se desarrollarán varias clases para encapsular las operaciones de matriz utilizadas por las redes neuronales cubiertas en este libro. Aprenderá a construir estas clases de matriz y cómo usarlas. Los futuros chapters explicarán cómo utilizar las clases de matriz con varios tipos diferentes de redes neuronales.

# La matriz de peso

En el último capítulo, aprendió que las redes neuronales hacen uso de dos tipos de valores: pesos y umbrales. Los pesos definen las interacciones entre las neuronas. Los umbrales definen lo que se necesita para que una neurona se dispare. Las conexiones ponderadas entre neuronas se pueden considerar como una matriz. Por ejemplo, considere las conexiones entre las dos capas siguientes de la red neuronal que se muestran en la Figura 2.1.

**Figura 2.1:**  **Una capa** de **dos neuronas**  **conectada**  **a**  **una** capa de **tres**  **neuronas.**

****

Puede ver las ponderaciones en la Figura 2.1. Los pesos se unen a las líneas dibujadas entre las neuronas. Cada una de las dos neuronas de la primera capa está conectada a cada una de las tres neuronas de la segunda capa. Hay un total de seis conexiones. Estas conexiones se pueden representar como una matriz de peso 3x2, como se describe en la ecuación 2.1.

**Ecuación 2.1:**  **Una** matriz **de**  **peso**



La matriz de peso se puede definir en Java de la siguiente manera:

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

La variable de umbral no es multidimensional, como la matriz de peso. Hay un valor umbral por neurona. Cada neurona de la segunda capa tiene un valor de umbral individual. Estos values se pueden almacenar en una matriz de variables **double** de Java. El código siguiente muestra cómo se puede definir toda la memoria de las dos capas.

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

*double thresholds[] = new double[2];*

Estas declaraciones incluyen tanto la matriz 3x2 como los dos valores de umbral para la segunda capa. No es necesario almacenar valores de umbral para la primera capa, ya que no está conectada a otra capa. Los valores de matriz de peso y umbral solo se almacenan para las conexiones entre dos capas, no para cada capa.

El método preferido para almacenar estos valores es combinar los umbrales con las ponderaciones de una matriz combinada. La matriz anterior tiene tres filas y dos columnas. Los umbrales se pueden considerar como la cuarta fila de la matriz de peso, que se puede definir de la siguiente manera:

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

El umbral combinado y la matriz de peso se describen en la Ecuación 2.2. En esta ecuación, la variable  **w**  **representa** las celdas utilizadas para almacenar pesos y la vari- capaz  **t** representa las celdas utilizadas para contener umbrales.

**Ecuación 2.2: A Umbral Y Peso Matriz**

[ ]

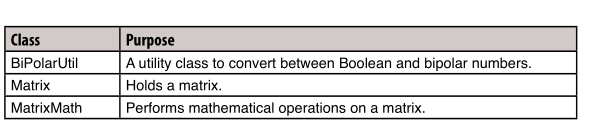
**

La combinación de los umbrales y pesos en una matriz tiene varias ventajas. Esta matriz ahora representa toda la memoria de esta capa de la red neuronal y sólo tiene que lidiar con una sola estructura. Además, dado que muchas de las mismas operaciones matemáticas realizadas en la matriz de peso también se realizan en el valor de umbral, tenerlas contenidas en una sola matriz permite que estas operaciones se realicen de manera más eficiente.

# Matrix Classes

Este capítulo presenta varias clases que se pueden utilizar para crear y manipular matrices. Estas clases de matriz se utilizarán a lo largo de este libro. Estas clases se resumen en la Tabla 2.1.

**Tabla 2.1:**  **Clases de**  **matriz**

****

Las tres secciones siguientes examinarán cada una de estas clases.

**the BiPolarUtil Class**

La clase **BiPolarUtil** se utiliza para cambiar entre un número bipolar y un valor **booleano.** Un valor **booleano** es **true**  o **false**. Un número bipolar es 1 o -1. Con esta clase, el valor  **booleano** de  **false** se expresa como un

-1 valor bipolar y el valor **booleano** de **true**  se expresa como un valor bipolar 1. La clase **BiPolarUtil** es una colección de métodos estáticos. Las firmas para estos métodos se muestran aquí.

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[ ][ ])

El Cuadro 2.2 resume las funciones proporcionadas por la clase **BiPolarUtil.**

**Tabla 2.2:**  **La**  **clase**  **BiPolarUtil**

|  |  |
| --- | --- |
| **Método** | **Propósito** |
| bipolar2double | Convierte un bipolar booleano en un doble. Por ejemplo, true se convierte a 1. |
| double2bipolar | Convierte un valor double en un booleano bipolar. Por ejemplo, -1 se convierte en false. |

Los dos métodos anteriores están sobrecargados, por lo que puede convertir un único valor, una matriz unidimensional o una matriz bidimensional. Los valores bipolares son particularmente útiles para las redes neuronales hopfield. Las redes neuronales hopfield se discutirán en el próximo chapter.

## La clase Matrix

La clase  **Matrix** se utiliza para construir matrices bidimensionales. Los valores contenidos en las matrices se almacenan como variables **dobles** Java. La clase  **Matrix** provee las operaciones fundamentales del álgebra lineal numérica. Para las operaciones envolv- ing dos o más matrices, el  **MatrixMath** clase se utiliza. La clase  **MatrixMath** se describe en la siguiente sección.

Las firmas para los miembros de la clase  **Matrix** se muestran aquí.

*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()*

Los métodos proporcionados por la clase  **Matrix** se resumen en la Tabla 2.3.

**Tabla 2.3:**  **La**  **clase**  **Matrix**

|  |  |
| --- | --- |
| **Método** | **Propósito** |
| crearColumnMatrix | Método estático que crea una matriz con una sola columna. |
| createRowMatrix | Método estático que crea una matriz con una sola fila. |
| add | Agrega el valor especificado a cada celda de la matriz. |
| clear | Establece cada celda de una matriz en cero. |
| clone | Crea una copia exacta de una matriz. |
| equals | Determina si dos matrices son iguales entre sí. |
| get | Obtiene el valor de una celda. |
| getCol | Obtiene una columna de un objeto matrix como un nuevo objeto de matriz. |
| getCols | Determina el número de columnas de un objeto de matriz. |
| getRow | Obtiene una fila de un objeto de matriz como un nuevo objeto de matriz. |
| getRows | Determina el número de filas de un objeto de matriz. |
| isVector | Determina si una matriz es un vector. Una matriz vectorial tiene una sola fila o una sola columna. |
| isZero | Determina si cada celda de un objeto de matriz es cero. |
| set | Establece el valor de una celda. |
| sum | Devuelve la suma de cada celda de un objeto de matriz. |
| toPackedArray | Convierte una matriz de matriz bidimensional en una matriz unidimensional de variables dobles de Java. |

La clase **Matrix**  se utilizará para construir las matrices de peso para todas las redes neuronales presentadas en este libro.

## La clase MatrixMath

La mayoría de las operaciones matemáticas de una clase **Matrix**  se realizan mediante la clase **MatrixMath.** Todos los métodos de la clase **MatrixMath** son **estáticos.** Además, siempre devuelven una nueva matriz y no modifican las matrices que se les pasan. Las firmas para los métodos **MatrixMath** se muestran aquí.

*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)*

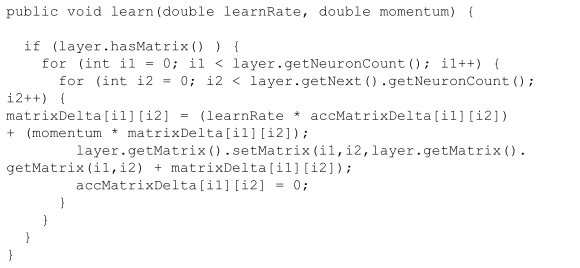
Estos métodos se resumen en la Tabla 2.4.

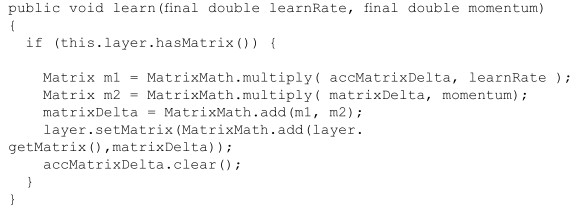
**Tabla 2.4:**  **La**  **clase**  **MatrixMath**

|  |  |
| --- | --- |
| **Método** | **Propósito** |
| add | Agrega dos matrices y produce una tercera matriz. |
| Divide | Divide una matriz por un escalar y produce una segunda matriz. |
| dotProduct | Calcula el producto punto de una matriz. |
| Identity | Crea una matriz de identidad de un tamaño especificado. |
| Multiply | Multiplica una matriz por otra y produce una tercera matriz. |
| Subtract | Resta una matriz de otra y produce una tercera matriz. |
| Traspose | Transpone una matriz y produce una nueva matriz. |
| vectorLength | Calcula la longitud cuadrada de un vector. |

Estas son las principales operaciones matemáticas que las redes neuronales deben realizar. Cada una de estas operaciones se discutirá extensamente más adelante en este capítulo.

Muchas implementaciones de redes neuronales Java crean operaciones de matriz directamente en sus clases de red neuronal. El resultado son muchos bucles anidados  dentro de la clase de red neu-ral. Por ejemplo, el código siguiente permite que una red neuronal aprenda.

El código anterior realiza varias operaciones de matriz; sin embargo, no es completamente obvio qué operaciones de matriz se están realizando. Al encapsular las operaciones de matriz dentro de varias clases de matriz, se puede simplificar el código anterior. Además, usted será capaz de decir, de un vistazo, qué operaciones de matriz se están realizando. El código siguiente realiza lo mismo que el código anterior; sin embargo, usa clases de matriz.

Como puede ver, se construyen varias matrices y, a continuación,  **matrixmath**

clase se utiliza para realizar operaciones sobre ellos.

# Construcción de una matriz

Hay varias maneras de construir un **matrix** objeto. Para construir un ma- trixvacío, utilice el código siguiente:

Matriz = nueva Matriz (3,2);

Esto construirá una matriz vacía de tres filas y dos columnas, que contiene solo ceros. Esta matriz se describe en la ecuación 2.3.

**Ecuación 2.3:**  **Una**  **matriz**  **vacía**

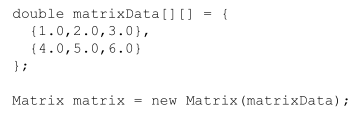
0 0 0

[ ]

0 0 0

### 0 0 0

También puede construir una matriz utilizando una matriz bidimensional, que le permite inicializar las celdas de la matriz en la declaración. El código siguiente crea una matriz 3x2 inicializada.



Esta matriz se describe en la ecuación 2.4.

**Ecuación 2.4: Una**  **matriz**  **inicializada**

### 1.0 2.0 3.0 4.0

[ ]

5.0 6.0 7.0 8.0

Las matrices también se pueden crear mediante una matriz unidimensional. Se puede usar una única matriz de dimen- sional para crear una fila o una matriz de columnas. Las matrices de fila y columna contienen una sola fila o una sola columna, respectivamente. Estas matrices también se denominan vectores. El código siguiente se puede usar para crear una matriz de filas.

matriz dobleData[] = {1.0,2.0,3.0,4.0};

Matriz = Matrix.createRowMatrix(matrixData);

Este will crea una matriz que contiene una sola fila. Esta matriz se describe en la ecuación 2.5.

**Ecuación 2.5:**  **Una** matriz de **fila/vector**

### [1.0 2.0 3.0 4. 0 ]

También es posible crear una matriz de columnas. Esta matriz toma una única matriz de dimensión al, al igual que antes; sin embargo, esta vez se crea una matriz con una sola columna. El código para hacer esto se muestra aquí.

matriz dobleData[] = {1.0,2.0,3.0,4.0};

Matriz = Matrix.createColumnMatrix(matrixData);

Esta matriz se describe en la ecuación 2.6.

**Ecuación 2.6: A Columna Matriz/Vector**

[ ]

### 1.0

2.0

### 3.0

4.0

En la siguiente sección verá cómo se pueden realizar las operaciones matemáticas en una o más matrices.

# Operaciones de matriz

En esta sección se revisarán algunas de las operaciones de matriz proporcionadas por la clase  **MatrixMath.** Se explicará cada operación de matriz, así como el código que se utiliza para realizar la operación. Puede probar algunas de las operaciones de matriz en línea en la siguiente dirección URL.

[**http://www.heatonresearch.com/examples/math/matrix**](http://www.heatonresearch.com/examples/math/matrix)

**Adición de** matriz

La adición de matrices es un procedimiento relativamente simple. Se suman las células correspondientes de dos matrices del mismo tamaño. Los resultados se devuelven en un nuevo ma-trix de igual tamaño. Equation 2.7 describe el proceso de adición de matriz.

**Ecuación 2.7:**  **Adición de**  **matriz**

[1 2]+[4 5]=[5 7 ]

3 4

6 7

9 11

La firma para el método **add** de la clase **MatrixMath** se muestra aquí.

adición de matriz estática pública (matriz final a, matriz final b)

El proceso comienza con la creación de una nueva matriz para contener los resultados de la ópera- tion.

resultado doble final[][] = nuevo doble[a.getRows()][a.getCols()];

A continuación, recorremos las celdas de cada fila y columna en las matrices de origen.

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

A continuación, el resultado de cada suma se coloca en la celda correspondiente en el

matriz de resultados.

resultado[resultRow][resultCol] = a.get(resultRow, resultCol)

+ b.get(resultRow, resultCol);

}

}

Por último, se crea una nueva matriz a partir de la matriz de  **resultados.**

devolver nuevo Matrix(result);

Este método **add** se puede utilizar para agregar dos matrices del mismo tamaño.

## Matrix Division por un escalar

Una matriz se puede dividir por un solo número o escalar. Por ejemplo, para dividir una matriz por el número 10, cada celda de la matriz se dividiría en 10. La ecuación 2.8 describe el proceso de dividir una matriz por un escalar.

**Ecuación 2.8:**  **División**  **matrix**  **por**  **un**  **escalar**

[2 4]/ 2=[1 2]

6 8

3 4

La firma de la función de división **de** matriz se muestra aquí.

división de matriz estática pública (matriz final a, doble b final)

En primer lugar, se crea una matriz de **resultados**  para contener los resultados de la división. Esta matriz tiene exactamente el mismo tamaño que la matriz que se va a dividir.

resultado doble final[][] = nuevo doble[a.getRows()][a.getCols()];

Para realizar la división, el método  **divide** debe recorrer en bucle todas las celdas de la matriz.

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

Cada celda de la matriz se divide por el número especificado y se coloca en el

matriz de resultados.

resultado[fila][col] = a.get(fila, col) / b;

}

}

Por último, se crea un nuevo objeto **Matrix** a partir de la matriz **de resultados.**

devolver nuevo Matrix(result);

Se devuelve esta matriz. Es el resultado de la división.

## Calcular el producto punto

El producto de puntos se puede calcular a partir de dos matrices vectoriales. Una matriz vectorial contiene una sola fila o una sola columna. Para determinar el producto punto de dos ma- trixes, deben tener el mismo número de celdas. No es necesario que las células de las dos matrices estén orientadas de la misma manera. Sólo es necesario que tengan el mismo número de células.

El producto punto es un escalar, un solo número, no otra matriz. Para calcular el producto punto, cada celda, tomada en orden y comenzando en el lado superior izquierdo de la matriz, se multiplica por la celda correspondiente en la otra matriz. A continuación, se resumen los resultados de estas operaciones multipli- cation. La ecuación 2.9 describe el proceso de cálculo del producto punto.

**Ecuación 2.9: Punto Producto**

[ ]

5

[1 2 3 4]∗ 6 =(1∗5)+( 2∗6)+(3∗7)+(4∗8)=5+12+21+32=70

7

### 8

La firma para el **dotProduct** método se muestra aquí.

public static double dotProduct(matriz final a, matriz final b)

En primer lugar, se crean dos matrices temporales para contener los valores de la matriz empaquetada. La matriz empaquetada es una matriz unidimensional simple que contiene ambas dimensiones, por lo que la matriz es una lista lineal plana.

último doble aArray[] = a.toPackedArray(); bArray doble final[] = b.toPackedArray();

Se declara una variable **de resultado**  para contener la suma del producto de punto, ya que se calculará.

doble resultado = 0;

longitud final int = aArray. longitud;

A continuación, el método recorre en bucle las celdas correspondientes de las dos matrices, agregando el resultado de cada operación de multiplicación a la  **variable** **de resultado.**

para (int i = 0; i < longitud; i++) { result += aArray[i] \* bArray[i];

}

Por último, se devuelve el número resultante.

resultado de retorno;

A esta variable de **resultado** se le asigna el producto de punto de las dos matrices.

## Multiplicación de matrices y la matriz de identidad

La multiplicación de matrices solo se puede realizar si dos matrices tienen di- mensions compatibles. Las dimensiones compatibles significan que el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. Esto significa que es legal trituraruna matriz 2x3 por una matriz 3x2. Sin embargo, no es legal multiplicar una matriz 2x3 por una matriz 2x6!

A continuación, veremos cómo multiplicar dos matrices. La ecuación 2.10 describe cómo una matriz 2x3 se multiplica por una matriz 3x2.

**Ecuación 2.10:**  **Multiplicación de**  **matriz**

[1 4]∗[ 7 8 9 ]=[(1∗7)+(4∗10) (1∗8)+( 4∗11) (1∗9)+( 4∗12)]=[47 52 57]

2 5

3 6

10 11 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)

64 71 78

81 90 99

También es importante tener en cuenta que la multiplicación de matriz no es conmutante. El resultado de 2\*6 es el mismo que 6\*2. Esto se debe a que la multiplicación es conmutativa cuando se trata de scalars, no así con matrices. El resultado de multiplicar una matriz 1x3 por una matriz 3x2 no es en absoluto lo mismo que multiplicar una matriz 3x2 por una matriz 1x3. De hecho, ni siquiera es válido multiplicar una matriz 3x2 por un 1x3. Recuerdo en la ecuación 2.10 multiplicamos una matriz 2x3 por una matriz 3x2. La ecuación 2.11 ilustra los resultados de mul- tiplying una matriz 3x2 por una matriz 2x3. Esta operación produce un resultado completamente diferente de la ecuación 2.10.

**Ecuación 2.11: No conmutativo Matriz Multiplicación**

[68 167 ]

7 8 9 ∗ 1 4 =

[ ]

[10 11 12]

2 5

3 6

[(10∗1)+(11∗2)+(12∗3) (10∗4)+(11∗5)+(12∗6)]

(7∗1)+(8∗2)+(9∗3) (7∗4)+(8∗5)+(9∗6)

= 50 122

Una matriz de identidad es una matriz que cuando se multiplica por otra matriz produce la misma matriz. Piense en esto como multiplicar un número por 1. La ecuación 2.12 describe la matriz de identidad.

**Ecuación 2.12:**  **Matriz**  **de identidad**

1 0 0

[ ]

0 1 0

### 0 0 1

Una matriz de identidad siempre es perfectamente cuadrada. Una matriz que no es cuadrada no tiene una matriz de identidad. Como se puede ver en la ecuación 2.12, la matriz de identidad es cre- ated comenzando con una matriz que tiene sólo cero valores. Las celdas en la diagonal desde la esquina noroeste hasta la esquina sureste se establecen en una.

La ecuación 2.13 describe una matriz de identidad que se multiplica por otra matriz.

**Ecuación 2.13:**  **Multiplicar**  **por**  **una** matriz **de**  **identidad**

[1 2 3]

4 5 6 ∗ 0 1 0 = (4∗1)+(5∗0)+(6∗0) ( 4∗0)+5∗1)+(6∗0) (4∗0)+(5∗0)+(6∗1) = 4 5 6

7 8 9

0 0 1

[1 0 0]

[(1∗1)+(2∗0)+(3∗0) (1∗0)+(2∗1)+(3∗0) (1∗0)+( 2∗0)+(3∗1)

[1 2 3]

el Resultante Matriz En Ecuación 2.13 Es el Mismo Como el Matriz ese Fue multi- Ejercía Por el matriz de identidad.

(7∗1)+(8∗0)+( 9∗0) (7∗0)+(8∗1)+(9∗0) (7∗0)+(8∗0)+(9∗1)

7 8 9

La firma para el método de **identidad** se muestra aquí.

identidad matrix estática pública (tamaño int final)

Este método creará una matriz de identidad del tamaño especificado por el  **tamaño** pa- rameter. En primer lugar, se crea una nueva matriz que corresponde al tamaño especificado.

resultado final de Matrix = nueva Matriz(tamaño, tamaño);

A continuación, se utiliza un bucle for para establecer el noroeste a la diagonal sureste en uno.

para (int i = 0; i < tamaño; i++) { result.set(i, i, 1);

}

Por último, se devuelve la matriz de identidad resultante.

resultado de retorno;

El método devuelve el  **resultado**.

## Multiplicación de matriz por un escalar

Las matrices también se pueden multiplicar por un escalar. La multiplicación de matrices por un escalar es muy simple de realizar: cada celda de la matriz se multiplica por el escalar especificado. La ecuación 2.14 muestra cómo se hace esto.

**Ecuación 2.14: Matriz Multiplicación Por a Escalar**

8 10 12

[1 2 3]

4 5 6 ∗2= 4∗2 5∗2 6∗2 =

7 8 9

[1∗2 2∗2 3∗2]

[ 2 4 6 ]

el Firma Para el **Multiplicar** Por a método escalar se muestra aquí.

7∗2 8∗2 9∗2

14 16 18

Matriz estática pública multiplica(matriz final a, doble b final)

En primer lugar, se crea una matriz de **resultados** para contener los resultados de la operación de multiplicación.

resultado doble final[][] = nuevo doble[a.getRows()][a.getCols()];

A continuación, el método  **multiply** recorre en bucle todas las celdas de la matriz original, lo multiplica por el escalar y, a continuación, almacena el resultado en la nueva matriz **de resultados.**

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

resultado[fila][col] = a.get(fila, col) \* b;

}

}

Por último, se crea un nuevo objeto  **Matrix** a partir de la matriz de resultados.

devolver nuevo Matrix(result);

A continuación,  **este** objeto Matrix se devuelve al método de llamada.

## Resta de matriz

La resta de matriz también es un procedimiento relativamente simple. Las dos matrices en las que se realizará la operación de resta deben tener exactamente el mismo tamaño. Cada celda de la matriz resultante es la diferencia de las dos celdas correspondientes de las matrices de origen. La ecuación 2.15 describe el proceso de resta de matriz.

**Ecuación 2.15:**  **Resta de**  **matriz**

[4 5 ]−[1 2 ]=[(4−1 ) (5−2 )]=[3 3 ]

6 7

3 4

(6−3 ) (7−4 )

3 3

La firma para el método  **subtract** de la clase  **MatrixMath** se muestra aquí.

matriz estática pública resta (matrizfinal a, matriz final b)

En primer lugar, se crea una nueva matriz para contener los resultados de las operaciones de resta.

resultado doble final[][] = nuevo doble[a.getRows()][a.getCols()];

A continuación, debemos recorrer en bucle cada fila y columna de las matrices de origen.

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

Los resultados de la operación de resta para un par determinado de celdas en las matrices de origen se colocan en la celda correspondiente en la matriz de **resultados.**

resultado[resultRow][resultCol] = a.get(resultRow, resultCol)

- b.get(resultRow, resultCol);

}

}

Por último, se crea una nueva matriz a partir de la matriz de  **resultados.**

devolver nuevo Matrix(result);

## Transponer una matriz

La transposición de matriz se produce cuando se cambian entre filas y columnas de una matriz. La ecuación 2.16 describe la transposición de una matriz.

**Ecuación 2.16:**  **Transposición de**  **matriz**

3 4 [2 4 6]

5 6

1 2 1 3 5[ ]

La firma para el método de **transposición** se muestra aquí.

transposición de matriz estática pública (entradafinal de Matrix)

Se crea una nueva matriz  **inverseMatrix** con filas y columnas iguales en tamaño a las inversas de las de la matriz original.

doble inversa finalMatrix[][] = nuevo doble[input.getCols()][entrada

. getRows()];

A continuación, recorremos todas las celdas de la matriz original. El valor de cada celda se copia en la ubicación de la matriz de resultados identificada por la fila inversa y la columna de la celda original.

para (int r = 0; r < input.getRows(); r++) { para (int c = 0; c < input.getCols(); c++) {

inverseMatrix[c][r] = input.get(r, c);

}

}

Por último, se crea un nuevo objeto **Matrix** a partir de la matriz **inverseMatrix.**

devolver nueva Matriz (inverseMatrix);

Este objeto **Matrix** recién creado se devuelve al método de llamada.

## Longitud del vector

La longitud de una matriz vectorial se define como la raíz cuadrada de las sumas cuadradas de cada celda de la matriz. La ecuación 2.17 describe cómo se calcula la longitud vectorial de una matriz vectorial.

**Ecuación 2.17:**  **Calcular** la **longitud**  **vectorial**

[ *x x* ... *X* ]=√( *X* )2+( *x* )2+...+( *X* )2

1 2 *n* 1 2 *n*

La clase  **MatrixMath** proporciona la función  **vectorLength** que se puede utilizar para calcular esta longitud. La firma de la función **vectorLength** se muestra aquí.

vector doble estático públicoLength( entrada final matrix )

En primer lugar, se crea una matriz de celdas individuales mediante la función **toPackedArray.** Esta función devuelve la matriz como una matriz simple de scalars. Esto permite calcular una longitud de matriz vectorial basada en columnas o filas, ya que ambos se convertirán en matrices simples de scalars.

doble v[] = input.toPackedArray(); doble rtn = 0,0 ;

A continuación, recorremos la matriz empaquetada y sumamos el cuadrado de cada número.

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

}

Por último, se devuelve la raíz cuadrada de la suma.

devolver Math.sqrt(rtn);

La longitud de un vector será particularmente importante para los mapas autoorganización que se presentarán en el capítulo 8.

# Operaciones bipolares

En formato binario,  **true** se representa con tél número uno y  **false** se vuelve a enviar con cero. La notación bipolar es otra forma de representar estados binarios. En la notación bipolar, **true** está representada por una y  **false** está representada por una negativa.

La ecuación 2.18 describe cómo un número booleano se convierte en un num-

ber.

**Ecuación 2.18:**  **Boolean**  **a**  **Bipolar**

*ḥ* (  *x* )=2x−1

La ecuación 2.19 hace lo contrario. Esta ecuación muestra cómo convertir un número bipolar en un número **booleano.**

**Ecuación 2.19:**  **Bipolar**  **a**  **booleana**

*ḥ* (  *x* )=(  *x*+1)

2

Las redes neuronales que funcionan en números **booleanos** normalmente requerirán que estos valores **booleanos** se expresen como números bipolares. Para ayudar con esta conversión, se proporciona la clase **BiPolarUtil.** En el Cuadro 2.2 se resumen las funciones proporcionadas por la clase **BiPolarUtil.**

El código siguiente utiliza la clase  **BiPolarUtil** para construir una matriz bipolar.

boolean booleanoData2[][] = {{true,false},{false,true}}; Matriz2 = matriz nueva( BiPolarUtil.

bipolar2double(booleanData2));

Este código producirá la matriz e descrita en la ecuación 2.20.

**Ecuación 2.20:**  **Una**  **matriz**  **bipolar**

### 1 −1

[ ]

−1 1

Esto será particularmente útil en el siguiente capítulo en el que se presenta un trabajo de red neuronal hopfield. Una red neuronal hopfield hace uso de matrices bipolares.

# Resumen del capítulo

Las matemáticas de matriz son muy importantes para las redes neuronales. Java no incluye soporte integrado para matrices; por lo tanto, aquí se han presentado tres clases de matriz. Estas clases de matriz se utilizan para crear matrices y realizar varias operaciones mathemati- cal en ellas.

La notación bipolar es una forma especial de representar valores binarios. En la notación bipolar, el valor binario de  **true** se representa con uno y el valor binario de  **false** se representa con uno negativo. La mayoría de las redes neuronales que funcionan con valores binarios harán uso de la notación bipolar. La red neuronal hopfield, que se introducirá en el capítulo 3, hace uso de números bipolares.

Este libro hará uso de las clases matriciales presentadas en este capítulo para todas las redes neu-ral que se introducirán. Esto le permitirá ver rápidamente el mathemati- cal underpinnings de cada red neuronal.

# Vocabulario

Bipolar booleana columna matriz punto matriz de identidad del producto Matrix

Vector escalar de matriz de fila

Matriz de peso

# Preguntas para revisión

1. ¿Cuál es el propósito de usar números bipolares, en lugar de num- bers booleanos?
2. ¿La multiplicación de matrices es conmutativa?
3. ¿Cuáles son las dimensiones de una matriz de peso utilizada para conectar una capa de dos neuronas a una capa de tres neuronas?
4. Realice la siguiente multiplicación.

### [3 2 ]∗ 1 2 3 =*�*

1 3

1 4

[4 5 6]

1. Realice el siguiente producto de puntos.

[6 ]

=

[6 2 7 4 ] *•*

6

7

9