**CHAPTER** **11:** **UNDERSTANDING** **THE** **SELF-ORGANIZING** **MAP**

* ¿Qué es un mapa autoorganizado?
* ¿Cómo se utiliza un mapa autoorganizado para clasificar patrones?
* Formación de un mapa autoorganizado
* Tratar con neuronas que no aprenden a clasificar

En el capítulo 5, aprendiste sobre la red neuronal de retro propagación feedforward. Aunque la arquitectura feedforward se utiliza comúnmente para redes neuronales, no es la única opción disponible. En este capítulo, examinaremos otra arquitectura comúnmente utilizada para redes neuronales, el mapa auto organizado (SOM).

El mapa autoorganizado, a veces llamado red neuronal de Kohonen, lleva el nombre de su creador, Tuvo Kohonen. El mapa autoorganizado se difiere de la red neuronal de retro propagación feedforward de varias maneras importantes. En este capítulo, examinaremos el mapa autoorganizado y veremos cómo se implementa. El capítulo 12 se contiene presentando una aplicación práctica del mapa autoorganizado, reconocimiento de caracteres.

# Presentación del mapa autoorganizado

El mapa autoorganizado difiere considerablemente de la red neuronal de retropropagación feedforward tanto en cómo se entrena como en cómo recuerda un patrón. El mapa autoorganizado no utiliza una función de activación o un valor de umbral.

Además, la salida del mapa autoorganizado no se compone de salida de varias neuronas; más bien, cuando se presenta un patrón en un mapa autoorganizado, una de las neuronas de salida se selecciona como el "ganador". Esta neurona "ganadora" proporciona la salida del mapa autoorganizado. A menudo, una neurona "ganadora" representa a un grupo en los datos que se presentan al mapa autoorganizado. Por ejemplo, en el capítulo 12 examinaremos un programa OCR que utiliza 26 neuronas de salida que mapean patrones de entrada a las 26 letras del alfabeto latino.

La diferencia más significativa entre el mapa autoorganizado y la red neuronal de retropropagación feedforward es que el mapa autoorganizado entrena en un modo sin supervisión. Esto significa que el mapa autoorganizado se presenta con datos, pero no se especifica la salida correcta que corresponde a los datos de entrada.

También es importante entender las limitaciones del mapa autoorganizados. Recordará de discusiones anteriores que las redes neuronales sin capas ocultas solo se pueden aplicar a ciertos problemas. Este es el caso del mapa autoorganizado. Los mapas autoorganización se utilizan porque son redes relativamente simples de construir y se pueden entrenar muy rápidamente.

## Cómo un mapa autoorganizado reconoce un patrón

Ahora examinaremos cómo el mapa autoorganizado reconoce un patrón. Vamos a iniciar por examinar la estructura del mapa autoorganizado. A continuación, se le indicará cómo entrenar el mapa autoorganizado para reconocer adecuadamente los patrones que desea.

## La estructura del mapa autoorganizado

El mapa autoorganizado funciona de manera diferente a la red neuronal feedforward que aprendimos en el capítulo 5, "Feedforward Backpropagation Neural Networks". El mapa autoorganizado solo contiene una capa de neuronas de entrada y una capa de neurona de salida. No hay ninguna capa oculta en un mapa autoorganizándose.

La entrada a un mapa autoorganizado se envía a la red neuronal a través de las neuronas de entrada. Las neuronas de entrada reciben números de punto flotante que componen el patrón de entrada a la red. Un mapa autoorganizado requiere que las entradas se normalicen para que caigan entre -1 y 1. Presentar un patrón de entrada a la red causará un reacción de las neuronas de salida.

La salida de un mapa autoorganizándose es muy diferente de la salida de una red neuronal feedforward. Recuerde del capítulo 5 que, si tenemos una red neuronal con cinco neuronas de salida, recibiremos una producción que consta de cinco valores. Como se señaló anteriormente, este no es el caso con el mapa autoorganizado. En un mapa autoorganizándose, solo una de las neuronas de salida produce realmente un valor. Además, este único valor es **true** o **false**. Por lo tanto, la salida del mapa autoorganizado suele ser el valor de la neurona que se disparó (por ejemplo, neuronas #5). La estructura de un mapa autoorganizado típico se muestra en la Figura 11.1.

### figura 11.1: A autoorganización mapa.

|  |  |  |
| --- | --- | --- |
|  | Entrada real |  |
|  | normalización |  |
|  |  |  |





I1

I2

O1

O2

Neuron ganadora

Normalización de entrada

Capa de entrada

Capa de salida

Salida final

Ahora que entiendes la estructura del mapa autoorganiz, examinaremos cómo la red procesa la información considerando un mapa autoorganizado muy simple. Esta red tendrá sólo dos neuronas de entrada y dos neuronas de salida. La entrada que se administrará a las dos neuronas de entrada se muestra en la Tabla 11.1.

### Tabla 11.1: Entradas de muestra a un mapa autoorganizado

|  |  |
| --- | --- |
| Neurona de entrada 1 (I1) | 0.5 |
| Neurona de entrada 2 (I2) | 0.75 |

También debemos conocer los pesos de conexión entre las neuronas. Los pesos de conexión se indican en el Cuadro 11.2.

### Tabla 11.2: Pesos de conexión en el mapa autoorganiz de muestra

|  |  |
| --- | --- |
| I1 -> O1 | 0.1 |
| I2 -> O1 | 0.2 |
| I1 -> O2 | 0.3 |
| I2 -> O2 | 0.4 |

Usando estos valores, ahora examinaremos qué neurona ganará y producirá out-put. Comenzaremos normalizando la entrada.

## Normalización de la entrada

El mapa autoorganizado requiere que su entrada se normalice. Por lo tanto, algunos textos se refieren a la normalización como una tercera capa. Sin embargo, en este libro, el mapa autoorganizado se considera una red de dos capas, porque sólo hay dos neuronas reales en el trabajo.

El mapa autoorganizado impone limitaciones estrictas a la entrada que recibe. La entrada al mapa autoorganizado debe estar entre los valores de –1 y 1. Además, cada una de las neuronas de entrada debe utilizar toda la gama. Si una o más de las neuronas de entrada sólo aceptaran los números entre 0 y 1, el rendimiento de la red neuronal sufriría.

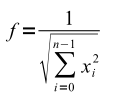
La entrada para un mapa autoorganizado generalmente se normaliza utilizando uno de los dos métodos comunes, la normalización multiplicativa y la normalización del eje z.

La normalización multiplicativa es la más simple de los dos métodos, sin embargo, la normalización del eje z a veces puede proporcionar un mejor factor de escala. Los algoritmos para estos dos métodos se discutirán en las próximas dos secciones. Comenzaremos con normalización multiplicativa.

## Normalización multiplicativa

Para realizar la normalización multiplicativa, primero debemos calcular la longitud vectorial de los datos de entrada o vector. Esto se hace sumando los cuadrados del vector de entrada y luego tomando la raíz cuadrada de este número, como se muestra en la ecuación 11.1.

### Ecuación 11.1: Normalización multiplicativa



*ƒ* = 1

√

*n*−1

∑ *x* 2

*Yo*

*i*=0

La ecuación anterior produce el factor de normalización por el que se multiplica cada entrada para escalarlas correctamente. Utilizando los datos de muestra proporcionados en las Tablas 11.1 y 11.2, el factor de normalización se calcula de la suiente manera:

1.0 / Math.sqrt( (0.5 \* 0.5) + (0.75 \* 0.75) )

Esto produce un factor de normalización de 1.1094.

## Normalización del eje Z

A diferencia del algoritmo multiplicativo para la normalización, la normalización del eje z al-gorithm no depende de los datos reales en sí; en su lugar, los datos sin procesar se multiplican por una constante. Para calcular el factor de normalización mediante la normalización del eje z, utilizamos la Ecuación 11.2.

### Ecuación 11.2: Normalización del eje Z

**

*ḥ* = 1

√ *n*

Como se puede ver en la ecuación anterior, el factor de normalización sólo depende del tamaño de la entrada, denotado por la variable **n**. Esto conserva la información absoluta de magnitudes. Sin embargo, no queremos ignorar completamente las entradas reales; por lo tanto, se crea una entrada sintética, basada en los valores de entrada. La entrada sintética se calcula utilizando la ecuación 11.3.

### Ecuación 11.3: Entrada sintética

*s* = *ḥ* √*n*−*l*2

La variable **n** representa el tamaño de entrada. La variable **f** es el factor de normalización. La variable **l** es la longitud Ddel vector. La entrada sintética se agregará al vector de entrada que se presentó a la red neuronal.

Es posible que se pregunte cuándo debe utilizar el algoritmo multiplicativo y cuándo debe utilizar el algoritmo del eje z. En general, querrá utilizar el algoritmo del eje z, ya que el algoritmo del eje z conserva la magnitud absoluta. Sin embargo, si la mayoría de los valores de training están cerca de cero, el algoritmo del eje z puede no ser la mejor opción. Esto se debe a que el componente sintético de la entrada dominará los otros valores cercanos a cero.

## Cálculo de la salida de cada neurona

Para calcular la salida, se deben considerar los pesos de conexión del vector de entrada y la neurona. En primer lugar, se debe calcular el producto punto de las neuronas de entrada y sus pesos de conexión. Para calcular el producto punto entre dos vectores, debe multi- ply cada uno de los elementos en los dos vectores como se muestra en la ecuación 11.4.

### Ecuación 11.4: Cálculo de la salida SOM

[ 0.5 0. 75 ]∗[ 0.1 0.2 ]=(0.5∗0.1)+(0.75∗0.2 )=0.395

Como se puede ver en el cálculo anterior, el producto punto es 0.395. Esta calculación- ción tendrá que realizarse para cada una de las neuronas de salida. En este ejemplo, sólo examinaremos los cálculos de la primera neurona de salida. Los cálculos necesarios para la segunda neurona de salida se llevan a cabo de la misma manera.

La salida ahora debe normalizarse multiplicándola por el factor de normalización que se determinó en el paso anterior. Debe multiplicar el producto punto de 0,395 por el factor de normalización de 1,1094. El resultado es una salida de 0,438213. Ahora que la salida se ha calculado y normalizado, debe asignarse a un número bipolar.

## Mapeo a un número a distancia bipolar

Como se recordará en el capítulo 2, un número bipolar es una forma alternativa de representar números binarios. En el sistema bipolar, el binario cero se asigna a –1 y el binario uno sigue siendo un 1. Debido a que la entrada a la red neuronal se ha normalizado a este rango, debemos realizar una normalización similar en la salida de las neuronas. Para hacer este mapeo, multiplicamos por dos y restamos uno. Para la salida de 0,438213, el resultado es una salida final de –0,123574.

El valor –0.123574 es la salida de la primera neurona. Este valor se comparará con las salidas de la otra neurona. Al comparar estos valores podemos determinar una neurona "ganadora".

## Elección del ganador

Hemos visto cómo calcular el valor de la primera neurona de salida. Si queremos determinar una neurona ganadora, también debemos calcular el valor de la segunda neurona de salida. Ahora revisaremos rápidamente el proceso para calcular la segunda neurona.

El mismo factor de normalización se utiliza para calcular la segunda neurona de salida que se utilizó para calcular la primera neurona de salida. Como se recuerda de la sección anterior, el factor de normalización es 1.1094. Si aplicamos el producto punto para los pesos de la segunda neurona de salida y el vector de entrada, obtenemos un valor de 0,45. Este valor se basa en el factor de normalización de 1,1094, lo que da como resultado un valor de 0,0465948. ahora puede calcular la salida final para la neurona 2 mediante la conversión de la salida de 0.0465948 a bipolar, que rinde –0.9068104.

Como se puede ver, ahora tenemos un valor de salida para cada una de las neuronas. La primera neurona tiene un valor de salida de –0.123574 y la segunda neurona tiene un valor de salida de –0.9068104. Para elegir la neurona ganadora, seleccionamos la neurona que produce el mayor valor de salida. En este caso, la neurona ganadora es la segunda neurona de salida con una salida de -0.9068104, que supera la salida de la primera neurona de -0.123574.

Ahora ha visto cómo se derivó la salida del mapa autoorganizado. Como se puede ver, los pesos entre las neuronas de entrada y salida determinan esta salida. En la siguiente sección veremos cómo se pueden ajustar estos pesos para producir una salida más adecuada para la tarea deseada. El proceso de entrenamiento modifica estos pesos y se describirá en la siguiente sección.

## Cómo aprende un mapa autoorganizado

En esta sección, aprenderá a entrenar a un mapa. Hay varios pasos involucrados en el proceso de entrenamiento. En general, el proceso para entrenar un autoorganiz- mapa de ing implica pasar por varias épocas hasta que el error del mapa autoorganiz está por debajo de un nivel aceptable. En esta sección, aprenderá a calcular la tasa de error para un mapa autoorganizado y cómo ajustar los pesos para cada época. También aprenderá a determinar cuándo no son necesarias épocas adicionales para entrenar aún más la red neuronal.

El proceso de formación para el mapa autoorganizado es competitivo. Para cada conjunto de entrenamiento, una neurona "ganará". Esta neurona ganadora tendrá su peso ajustado para que reaccione aún más fuertemente a la entrada la próxima vez que lo vea. A medida que diferentes neu-rons ganan por diferentes patrones, su capacidad para reconocer ese patt ern en particular aumentará.

Primero examinaremos el proceso general de formación del mapa autoorganizado. El mapa autoorganizado se entrena repitiendo épocas hasta que ocurre una de dos cosas: si el error calculado está por debajo de un nivel aceptable, el proceso de entrenamiento está completo. Si, por otro lado, la tasa de error ha cambiado en una cantidad muy pequeña, este individu- al ciclo será abortado sin que se produzcan épocas adicionales. Si se determina que el ciclo debe ser abortado, los pesos se inicializarán con valores aleatorios y comenzará un nuevo ciclo de entrenamiento.

## Tasa de aprendizaje

La tasa de aprendizaje es una variable que es utilizada por el algoritmo de aprendizaje para ajustar los pesos de las neuronas. La tasa de aprendizaje debe ser un número positivo inferior a 1 y suele ser de 0,4 o 0,5. En las secciones siguientes, el símbolo alfa especificará la tasa de aprendizaje.

Por lo general, establecer la tasa de aprendizaje en un valor más alto utilizará el entrenamiento para prog- ress más rápidamente. Sin embargo, la red puede no converger si la tasa de aprendizaje se establece en un número demasiado alto. Esto se debe a que las oscilaciones de los vectores de peso serán demasiado grandes para que los patrones de clasificación surjan.

Otra técnica es comenzar con una tasa de aprendizaje relativamente alta y disminuir esta tasa a medida que avanza la formación. Esto permite un entrenamiento inicial rápido de la red neuronal que luego está "afinada" a medida que avanza el entrenamiento.

## Ajuste de pesos

La memoria e del mapa autoorganizado se almacena dentro de las conexiones ponderadas entre la capa de entrada y la capa de salida. Los pesos se ajustan en cada época. Una época ocurre cuando los datos de entrenamiento se presentan en el mapa autoorganizado y los pesos se ajustan en función de los resultados de estos datos. Los ajustes en los pesos deben producir una red que produzca resultados más favorables la próxima vez que se presenten los mismos datos de entrenamiento. Las épocas continúan a medida que se presentan más y más datos a la red y se ajustan los pesos.

Eventualmente, el retorno de estos ajustes de peso disminuirá hasta el punto de que ya no es valioso continuar con este conjunto particular de pesos. Cuando esta hap- plumas, toda la matriz de peso se restablece a nuevos valores aleatorios; comenzando así un nuevo ciclo. La matriz de peso final que se utilizará será la mejor matriz de peso de cada uno de los ciclos. Ahora examinaremos los pesos how se transforman.

El método original para calcular los cambios en los pesos, propuesto por Kohonen,a menudo se denomina método aditivo. Este método utiliza la siguiente equiparación- ción:

### Ecuación 11.5: Ajuste de los pesos SOM (aditivo)

*wt*+ 1

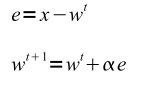
*wt*+ *x*

= *longitud* (*wt*+ *x* )

La variable **x** es el vector de entrenamiento que se presentó a la red. Variable **wt** es el peso de la neurona ganadora, y el resultado de la ecuación es el nuevo peso. Las barras verticales dobles representan la longitud vectorial. Esta ecuación será implemented como método en el ejemplo de mapa autoorganizado presentado más adelante en este capítulo.

Este método aditivo generalmente funciona bien para la mayoría de los mapas autoorganización; howev- er, en los casos para los que el método aditivo muestra una inestabilidad excesiva y no se con- verge, se puede utilizar un método alternativo. Este método se denomina método sustractivo. El método sustractivo utiliza las siguientes ecuaciones:

### Ecuación 11.6: Ajuste del peso SOM (sustractivo)

**

*e*= *x* −*wt* *wt*+1=*wt*+ *e*

Estas dos ecuaciones describen la transformación básica que se producirá en los pesos de la red. En la siguiente sección, verá cómo estas ecuaciones son imple- mented como un programa Java, y su uso se demostrará.

## Cálculo del error

Before podemos entender cómo calcular el error para la red neuronal, primero debemos definir el "error". Un mapa autoorganizado está entrenado de una manera no supervisada, por lo que la definición de error es algo diferente de la definición con la que estamos familiarizados.

En el capítulo anterior, la capacitación supervisada implicaba calcular el error. El error fue la diferencia entre la salida prevista de la red neuronal y la salida real de la red neuronal. En este capítulo, estamos examinando la formación indeseable. En el entrenamiento sin supervisión, no hay una salida anticipada; por lo tanto, usted puede estar preguntándose exactamente cómo podemos calcular un error. La respuesta es que el error que estamos calculando no es un error verdadero, o al menos no un error en el sentido normal de la palabra.

El propósito del mapa autoorganizado es clasificar la entrada en varios conjuntos. El error para el mapa autoorganizado, por lo tanto, proporciona una medida de qué tan bien la red el trabajo consiste en clasificar la entrada en grupos de salida. El error en sí no se utiliza para modificar los pesos, como es el caso en el algoritmo de retropropagación. No hay una forma oficial de calcular el error para un mapa autoorganizado, por lo que examinaremos dos métodos diferentes en la siguiente sección mientras exploramos cómo implementar un método de entrenamiento de Java.

# Implementación del mapa autoorganizado

Ahora que tiene una comprensión de cómo funciona el mapa autoorganizándose, implementaremos uno usando Java. En esta sección, veremos cómo se pueden usar varias clases juntas para crear un mapa autoorganizado. Después de esta sección, se le mostrará un ejemplo de cómo utilizar las clases de mapa autoorganización para crear un simplemapaauto-orga- nizing. Por último, en el capítulo 12, se le mostrará cómo construir una aplicación más compleja, basada en el mapa autoorganizado, que puede reconocer la escritura a mano.

Primero, debe entender la estructura de las clases de mapa autoorganización que estamos construyendo. Las clases utilizadas para implementar el mapa autoorganizado se suman a la tabla 11.3.

### Tabla 11.3: Clases utilizadas para implementar el mapa autoorganizado

|  |  |
| --- | --- |
| **clase** | **propósito** |
| NormalizarInput | Normaliza la entrada para el mapa autoorganizado. Esta clase implementa el método de normalización descrito anteriormente en este capítulo. |
| AutoorganizaciónMap | Esta es la clase principal que implementa el mapa autoorganizado. |
| TrainSelfOrganizingMap | Se utiliza para entrenar el mapa autoorganizándose. |

Ahora que está familiarizado con la estructura general de las clases de mapa autoorganización, examinaremos cada clase individual. Verá cómo estas clases funcionan juntas para proporcionar funcionalidad de mapa autoorganizándose. Comenzaremos examinando cómo se construye el conjunto de entrenamiento mediante la clase **NormalizeInput.**

**La clase** de **normalización** **SOM**

La clase **NormalizeInput** recibe toda la información que necesitará de su constructor. La firma para el constructor se muestra aquí:

public NormalizeInput( entrada doble final[], tipo normalizationtype final)

El constructor comienza almacenando el tipo de normalización solicitado. Esto puede ser la normalización multiplicativa o del eje z.

this.type = tipo;

A continuación, se calculan el factor de normalización y los valores de entrada sintéticos. El

**calculateFactors** método se explica en la siguiente sección.

calculateFactors(entrada);

Por último, se crea la matriz de entrada. Esto incluye la entrada sintética. El

**createInputMatrix** método se describe en una sección a seguir. this.inputMatrix = this.createInputMatrix(entrada, this.synth); **Cálculo de** **los** **factores**

El método **calculateFactors** calcula tanto el factor de normalización como el valor de entrada sintético. La firma del método **calculateFactors** es

se muestra aquí:

void calculateFactorsprotegido( entrada doble final[]) {

En primer lugar, la matriz de entrada se convierte en una matriz de columnas.

entrada final matrixMatrix = Matrix.createColumnMatrix(entrada);

A continuación, se calcula la longitud del vector para la variable **inputMatrix.**

doble len = MatrixMath.vectorLength(inputMatrix);

Se evalúa la longitud del vector, para asegurarse de que no se ha vuelto demasiado pequeño.

len = Matemáticas.max(len, SelfOrganizingMap.VERYSMALL);

Se determina el número de entradas y este valor se almacena en **numInputs**

variable.

final int numInputs = input.length;

A continuación, se determina el tipo de normalización que se va a realizar.

if (this.type == NormalizationType.MULTIPLICATIVE) {

Si el tipo de normalización es multiplicativo, se utiliza el recíproco de la longitud vectorial como factor de normalización.

this.normfac = 1.0 / len;

Dado que el método de normalización es aditivo, no se necesita ninguna entrada sintética. Simplemente establecemos la variable de entrada sintética en cero, para que no tenga ninguna influencia.

this.synth = 0.0;

} else {

Si se utiliza la normalización del eje z, el factor de normalización se calcula como el recíproco de la raíz cuadrada del número de entradas.

this.normfac = 1.0 / Math.sqrt(numInputs);

Ahora debemos determinar la entrada sintética.

final doble d = numInputs – Math.pow(len,2);

Si la entrada sintética se calcula para ser mayor que cero, entonces la multiplicamos por el factor de normalización.

if (d > 0.0) {

this.synth = Math.sqrt(d) \* this.normfac;

} else { este.synth = 0;

}

}

Si la entrada sintética es inferior a cero, entonces establecemos la variable de entrada sintética en

cero.

## Creación de la matriz de entrada

Ahora que se ha determinado el factor de normalización y la entrada sintética, se puede crear la matriz in-put. La matriz de entrada se crea mediante el método **createInputMatrix.** La firma para el método **createInputMatrix** se muestra aquí:

matriz protegida createInputMatrix( doble patrón final[],

doble extra final)

En primer lugar, se crea una matriz que tiene una fila y columnas iguales a una más que la longitud del patrón. La columna adicional mantendrá la entrada sintética.

resultado final de Matrix = nueva Matriz (1, pattern.length + 1);

A continuación, se insertan todos los valores del patrón.

para (int i = 0; i < pattern.length; i++) { result.set(0, i, patrón[i]);

}

Por último, se agrega la entrada sintética y se devuelve la variable **de resultado.**

result.set(0, pattern.length, extra);

resultado de retorno;

La matriz de entrada ya está lista para su uso.

# La clase de implementación som

El mapa autoorganizado se implementa en la clase **TrainSelfOrganizingMap**. Se presenta un patrón en el SOM utilizando un método denominado **ganador.** La firma para este método se muestra aquí:

público int ganador ( doble entrada final[]) {

Este método hace poco más que crear una matriz normalizada y enviarla a una versión más avanzada del método **ganador** que acepta una matriz normalizada.

final NormalizeInput normalizedInput = new NormalizeInput(input, this.normalizationType);

La neurona winning se devuelve de este método **ganador** más avanzado.

ganador delretorno (normalizedInput);

Este método **ganador** acepta un objeto **NormizedInput.** La firma para este método se muestra aquí:

público int ganador (última entrada NormalizedInput)

En primer lugar, se crea una variable local para contener la neurona ganadora. Esta variable se denomina

**ganar**.

int win = 0;

A medida que avanzamos, realizamos un seguimiento de la neurona de salida con la mayor producción. Una de las neuronas de salida ganará. Establecemos la variable **más grande** en un número muy pequeño antes de empezar.

doble más grande = valor Double.MIN\_;

Luego pasamos por encima de las neuronas de salida.

para (int i = 0; i < this.outputNeuronCount; i++) {

Obtenemos la fila de la matriz de peso que corresponde a esta salida neu- ron.

optr final de Matrix = this.outputWeights.getRow(i);

A continuación, obtenemos el producto punto entre esta fila y el patrón de entrada. Esta es la salida para esta neurona.

this.output[i] = MatrixMath

. dotProduct(input.getInputMatrix(), optr)

\* input.getNormfac();

La salida de la neurona se asigna a un número entre –1 y 1.

this.output[i] = (this.output[i]+1.0)/2.0;

El número se evalúa para ver si esta es la mayor salida hasta ahora.

if (this.output[i] > más grande) { biggest = this.output[i]; ganar = i;

}

Si la salida está por encima de una o por debajo de cero, se ajusta según sea necesario.

if( este.output[i] <0 ) { this.output[i]=0;

}

if( este.output[i]>1 ) { this.output[i]=1;

}

}

La neurona ganadora es devuelta.

volver a ganar;

Como puede ver, la salida se calcula de forma muy diferente a la salida de las redes feedforward vistas anteriormente en este libro. El mapa autoorganizado es una red neuronal competi- tive; por lo tanto, la salida de esta red neuronal es el neu- ronganador.

# La clase de formación SOM

El mapa autoorganizado está entrenado utilizando técnicas diferentes a las utilizadas con las redes neuronales feedforward demostradas hasta ahora. El entrenamiento se realiza mediante una clase denominada **TrainSelfOrganizingMap**. Esta clase se implementa como los otros métodos de entrenamiento; pasa por una serie de iteraciones hasta que el error es sufi- científicamente pequeño. La iteración de entrenamiento se discute en el próximo section.

## Iteración de entrenamiento

Para realizar una **iteración** del entrenamiento, se llama al método de **iteración** de la clase **TrainSelfOrganizingMap.** La firma para el método **de iteración** se muestra aquí:

iteración del vacío público() produce RuntimeException

En primer lugar, se llama a **evaluateErrors** para determinar el nivel de error actual. La variable **totalError,** que se acaba de calcular, se guarda en la variable **globalError.**

evaluarErrors();

this.totalError = this.globalError;

El error actual se evalúa para ver si es mejor que el mejor error encontrado hasta ahora. Si es así, los pesos se copian sobre la matriz de mejor peso anterior.

if (this.totalError < this.bestError) { this.bestError = this.totalError; copyWeights(this.som, this.bestnet);

}

El número de neuronas que han ganado, desde la última vez que los errores fueron calcu30/05/2021lated, está determinado.

ganadores int = 0;

para (int i = 0; i < this.won.length; i++) { si (this.won[i] != 0) {

ganadores++;

}

}

Si ha habido muy pocos ganadores, uno se ve obligado.

if ((ganadores < this.outputNeuronCount) && (ganadores < this.train. longitud)) {

forceWin(); retorno;

}

Los pesos se ajustan en función del entrenamiento de esta iteración.

ajustar Pesos();

La tasa de aprendizaje se reduce gradualmente a 0,01.

if (this.learnRate > 0.01) { this.learnRate \*= this.reduction;

}

A continuación, los pesos se copian de la mejor red.

copyWeights(this.som, this.bestnet);

Por último, estos pesos se normalizan.

para (int i = 0; i < this.outputNeuronCount; i++) { normalizeWeight(this.som.getOutputWeights(), i);

}

Esto completa una iteración de entrenamiento.

## Evaluación de errores

La evaluación de errores es una parte importante del proceso de capacitación. Implica disuadir la cantidad de neuronas que ganan por un patrón de entrenamiento determinado. La neurona que tiene la mayor respuesta a un patrón de entrenamiento dado se ajusta para fortalecer esa victoria. Los errores se calculan mediante el método **evaluateErrors.** La firma para este método se muestra aquí:

void evaluateErrors() produce RuntimeException {

En primer lugar, se borra la matriz de corrección. La matriz de corrección mantendrá las correcciones de entrenamiento. Estas correcciones se aplicarán a la red neuronal real más adelante cuando se llame al método **adjustWeights.** El método **adjustWeights** se cubre en la siguiente sección.

this.correc.clear();

para (int i = 0; i < this.won.length; i++) { this.won[i] = 0;

}

this.globalError = 0.0;

A continuación, recorremos todos los conjuntos de entrenamiento.

para (int tset = 0; tset < this.train.length; tset++) {

La entrada se normaliza y se presenta en la red neuronal.

final NormalizeInput input = new NormalizeInput(this.train[tset], this.som.getNormalizationType());

La variable **mejor** mantendrá la neurona ganadora.

final int best = this.som.winner(entrada);

Se registra el número de veces que gana cada neurona.

this.won[best]++;

A continuación, se obtienen los pesos de la neurona ganadora.

matriz final wptr = this.som.getOutputWeights(). getRow(mejor);

La longitud se calculará y colocará en la variable **de longitud.**

doble longitud = 0,0; doble diferencia;

Ahora repasamos todas las neuronas de entrada.

para (int i = 0; i < this.inputNeuronCount; i++) {

Se calcula la diferencia entre el conjunto de entrenamiento y la entrada de matriz de peso correspondiente.

diff = this.train[tset][i] \* input.getNormfac()

- wptr.get(0, i);

La longitud se calcula cuadrando la diferencia. Una longitud es la raíz cuadrada de la suma de los cuadrados.

longitud += diff \* diff;

Lo que se haga a continuación depende del método de aprendizaje.

if (this.learnMethod == LearningMethod.SUBTRACTIVE) {

Para el método sustractivo, la diferencia se añade a la neurona ganadora.

this.correc.add(best, i, diff);

} else {

Para el método aditivo, se utiliza una matriz de trabajo. La matriz de trabajo es una matriz temporal utilizada para contener los valores que se agregarán. La matriz de trabajo es una sola columna con un número de filas iguales a las neuronas de entrada. El conjunto de entrenamiento multiplicado por el cor- valor de peso de respuesta se agrega y se escala por la tasa de aprendizaje.

this.work.set(0, i, this.learnRate \* this.train[tset][i]

\* input.getNormfac() + wptr.get(0, i));

}

}

Por último, se aborda la entrada sintética. Se determina la diferencia entre la entrada sintética y el valor de matriz de peso correspondiente.

diff = input.getSynth() - wptr.get(0, this.inputNeuronCount);

Esta diferencia se agrega a la longitud.

longitud += diff \* diff;

Si el método de aprendizaje es aditivo, entonces la diferencia sintética simplemente se agrega.

if (this.learnMethod ==LearningMethod.SUBTRACTIVE){ this.correc.add(best, this.inputNeuronCount, diff);

} else {

El valor de matriz de trabajo para esta neurona de entrada se establece en la entrada sintética agregada al valor de matriz de peso correspondiente escalado por la tasa de aprendizaje.

this.work

.set(0, this.inputNeuronCount, this.learnRate

\* input.getSynth()

+ wptr.get(0, this.inputNeuronCount));

}

La longitud calculada de las diferencias es el error. A continuación, determinamos si esto supera el error actual.

si (longitud > this.globalError) { this.globalError = longitud;

}

Hasta ahora, el método aditivo no ha estado modificando la matriz de corrección; más bien ha estado modificando la matriz de trabajo. La matriz de corrección ahora debe actualizarse con los cambios que se reflejarán cuando se llame al método **adjustWeights.**

if (this.learnMethod == LearningMethod.ADDITIVE){ normalizeWeight(this.work, 0);

para (int i = 0; i <= this.inputNeuronCount; i++) { this.correc.add(mejor, i, this.work.get(0, i)

- wptr.get(0, i));

}

}

}

El cálculo del error ya está completo.

this.globalError = Math.sqrt(this.globalError);

}

Por último, la longitud se determina realizando la raíz cuadrada del error, que es la suma de las diferencias al cuadrado.

## Forzar un ganador

Hay momentos en que ciertas neuronas no ganarán por ningún patrón de entrenamiento. Estas neuronas son de peso muerto y deben ajustarse para ganar por algunos patrones. El método **forceWin** se utiliza para ajustar estas neuronas. Este método se llama cuando muy pocas neuronas han estado ganando. La firma para el **forceWin** método se muestra aquí.

void forceWin() produce RuntimeException

La variable **mejor** mantendrá la neurona ganadora para cada iteración de conjunto deformación de trenes.

La variable **que** mantendrá el conjunto de entrenamiento que tiene la neurona ganadora más baja.

int mejor, que = 0;

En primer lugar, se obtienen los pesos de salida y la última salida del mapa autoorganizado que se va a entrenar.

resultado final de MatrixWeights = this.som.getOutputWeights();

A continuación, recorremos todos los conjuntos de entrenamiento y vemos qué neurona de salida tiene la respuesta más pequeña. Initialize **dist** a un gran valor, y seguimos bajando a medida que encontramos valores de salida in-creasingly pequeños.

doble dist = Double.MAX\_VALUE;

A continuación, recorremos todos los conjuntos de entrenamiento.

para (int tset = 0; tset < this.train.length; tset++) {

Se obtiene la neurona ganadora de cada conjunto de entrenamiento.

mejor = this.som.winner(this.train[tset]); salida doble final[] = this.som.getOutput();

La neurona se evalúa para ver si tiene una salida más baja que cualquier neurona anterior encontrada. Si es así, esta es la nueva neurona más baja.

si (salida[mejor] < dist) { dist = salida[best]; que = tset;

}

}

Luego reprocesamos la neurona con la salida más baja. El conjunto de entrenamiento es normal para que pueda presentarse en la red neuronal.

final NormalizeInput input = new NormalizeInput(this.train[which], this.som.getNormalizationType());

mejor = this.som.winner(entrada);

salida doble final[] = this.som.getOutput();

A continuación, se identifica la neurona con mayor producción para el conjunto de entrenamiento que produjo el ganador más bajo.

dist = valor Double.MIN\_;

int i = this.outputNeuronCount; mientras que ((i--) > 0) {

if (this.won[i] ! = 0) { continuar;

}

Si esta neurona es más baja, entonces se elige.

if (salida[i] > dist) { dist = salida[i]; que = i;

}

}

Los pesos de la neurona más baja se ajustan para que la neurona responda a

este patrón de entrenamiento.

para (int j = 0; j < input.getInputMatrix(). getCols(); j++) { outputWeights.set(que, j,

input.getInputMatrix().get(0,j));

}

normalizarWeight(outputWeights, que);

Por último, los pesos se normalizan.

## Ajustar pesos

Una vez calculados los errores, se llama al método **adjustWeights.** El método **adjustWeights** aplica la matriz de corrección a la matriz real. La firma para el método **adjustWeights** se muestra aquí.

doble ajuste protegidoWeights()

Establezca el **resultado** en cero. El resultado es el valor de error ajustado.

doble resultado = 0,0;

Repasamos todas las neuronas de salida.

para (int i = 0; i < this.outputNeuronCount; i++) {

Si esta neurona de salida nunca ha ganado, entonces continúe; no hay nada que hacer.

El método **forceWin** probablemente ayudará a esta neurona más tarde.

if (this.won[i] == 0) { continuar;

}

Se calcula el recíproco del número de veces que esta neurona de salida ha ganado.

doble f = 1.0 / this.won[i];

Si se utiliza el método sustractivo, esta reciprocidad se escala por la tasa de aprendizaje.

if (this.learnMethod == LearningMethod.SUBTRACTIVE) { f \*= this.learnRate;

}

Se calcula la longitud vectorial de los pesos de entrada multiplicados por sus valores de matriz de corrección.

doble longitud = 0,0;

for (int j = 0; j <= this.inputNeuronCount; j++) { final double corr = f \* this.correc.get(i,j); this.somLayer.getMatrix().add(i,j, corr); longitud += corr \* corr;

}

Se registra la longitud más larga.

si (longitud > resultado) { resultado = longitud;

}

}

Con la tasa de aprendizaje, la corrección se escala.

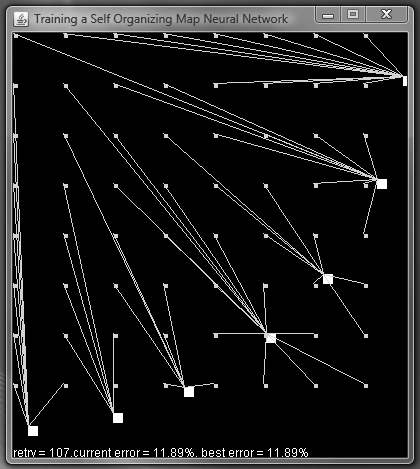
resultado = Math.sqrt(resultado) / this.learnRate; resultado de retorno;

Por último, se devuelve el valor de error.

# Uso del mapa autoorganizado

Ahora examinaremos un programa simple que entrena un mapa autoorganizado. A medida que la red está entrenada, se le mostrará una visualización gráfica de los pesos. La salida de este programa se muestra en la Figura 11.2.

### Figura 11.2: Formación de un mapa autoorganizándose.



Este programa contiene dos neuronas de entrada y siete neuronas de salida. Cada una de las siete neuronas de salida se trazan como cuadrados blancos. La dimensión x muestra los pesos entre ellos y la primera neurona de entrada y la dimensión y muestra los pesos que se preadolescentes y la segunda neurona de entrada. Verás que las cajas se mueven a medida que avanza el entrenamiento.

También verá líneas de puntos seleccionados en la cuadrícula dibujados en cada uno de los cuadrados. Estos identifican qué neurona de salida está ganando para las coordenadas x e y de ese punto. Puntos con coordenadas x e y similares se muestran como reconocidos por la misma neurona de salida.

Ahora examinaremos el programa, como se muestra en el Listado 11.1.

### Listado 11.1: El ejemplo de entrenamiento SOM (TestSOM.java)

paquete com.heatonresearch.book.introneuralnet.ch11.som;

importar java.awt.Color; importar java.awt.Dimension; importar java.awt.Graphics; importar java.awt.Image; importar java.awt.Toolkit;

importar java. texto. NumberFormat;

importar javax.swing. JFrame;

importar javax.swing. WindowConstants;

importar com.heatonresearch.book.introneuralnet.neural.matrix.

Matriz;

importar com.heatonresearch.book.introneuralnet.neural.som.

SelfOrganizingMap;

importación com.heatonresearch.book.introneuralnet.neural. som.

TrainSelfOrganizingMap;

importar com.heatonresearch.book.introneuralnet.neural.som.

NormalizarInput. NormalizaciónTipo;

importar com.heatonresearch.book.introneuralnet.neural.som.

TrainSelfOrganizingMap. LearningMethod;

/\*\*

* Capítulo 11: Uso de un mapa autoorganizado

\*

* TestSOM: Ejemplo muy sencillo para probar el SOM y mostrar cómo
* Obras.

\*

* @author Jeff Heaton
* @version 2,1

\*/

clase pública TestSOM extiende JFrame implementa Runnable {

/\*\*

\* Id. de serie para esta clase.

\*/

serial largo final estático privadoVersionUID = 2772365196327194581L;

/\*\*

\* Cuántas neuronas de entrada utilizar.

\*/

int final estático público INPUT\_COUNT = 2;

/\*\*

\* Cuántas neuronas de salida utilizar.

\*/

int final estático público OUTPUT\_COUNT = 7;

/\*\*

\* Cuántas muestras aleatorias generar.

\*/

int final estático público SAMPLE\_COUNT = 100;

/\*\*

* Inicia el programa.

\*

* args @param
* No se usa.

\*/

vacío estático público principal (final String args[]) { aplicación testsom final = nuevo TestSOM(); app.setVisible(true);

último Thread t = new Thread(app); t.setPriority(Thread.MIN\_PRIORITY); t.start();

}

/\*\*

* La longitud de la unidad en píxeles, que es el máximo de la
* altura y anchura de
* la ventana.

\*/

unidad int protegidaLength;

/\*\*

* Cuántos reintentos hasta ahora.

\*/

int retry protegido = 1;

/\*\*

* El porcentaje de error actual.

\*/

protegido double totalError = 0;

/\*\*

* El mejor porcentaje de error.

\*/

protegido double bestError = 0;

/\*\*

* La red neuronal.

\*/

red SelfOrganizingMap protegida; doble entrada protegida[][];

/\*\*

* La imagen fuera de pantalla. Se utiliza para evitar parpadeos.

\*/

imagen protegida fuera de la pantalla;

/\*\*

* El constructor configura la posición y el tamaño de la
* ventana.

\*/ TestSOM() {

setTitle(

"Formación de una red neuronal de mapas autoorganizándose"); setSize(400, 450);

kit de herramientas final = Toolkit.getDefaultToolkit(); dimensión final d = toolkit.getScreenSize(); setLocation((int) (d.width - this.getSize(). getWidth()) / 2,

(int) (d.height - this.getSize(). getHeight()) / 2); setDefaultCloseOperation( WindowConstants.DISPOSE\_ON\_CLOSE); setResizable(false);

}

/\*\*

* Mostrar el progreso de la red neuronal.

\*

* @param g
* Un objeto gráfico .

\*/ @Override

pintura vacía pública (Gráficos g) { si (this.net == null) {

retorno;

}

if (this.offScreen == null) {

this.offScreen = this.createImage((int)

getBounds(). getWidth(),

(int) getBounds(). getHeight());

}

g = this.offScreen.getGraphics();

último ancho int = getContentPane(). getWidth(); altura final int = getContentPane(). getHeight(); this.unitLength = Math.min(ancho, alto); g.setColor(Color.black);

g.fillRect(0, 0, anchura, altura);

trazar los pesos de las neuronas de salida g.setColor(Color.white);

resultado final de matrixWeights = this.net.getOutputWeights();

para (int y = 0; y < outputWeights.getRows(); y++) {

g.fillRect((int) (outputWeights.get(y, 0) \* this.unitLength),

(int) (outputWeights.get(y, 1)

\* this.unitLength), 10, 10);

}

trazar una cuadrícula de muestras para probar la red con g.setColor(Color.green);

para (int y = 0; y < this.unitLength; y += 50) {

para (int x = 0; x < this.unitLength; x += 50) { g.fillOval(x, y, 5, 5);

doble d final [] = nuevo doble[2]; d[0] = x;

d[1] = y;

final int c = this.net.winner(d); final int x2 = (int) (

outputWeights.get(c, 0) \* this.unitLength); final int y2 = (int) (

outputWeights.get(c, 1) \* this.unitLength);

g.drawLine(x, y, x2, y2);

}

}

mostrar la información de estado

g.setColor(Color.white);

final NumberFormat nf = NumberFormat.getInstance(); nf.setMaximumFractionDigits(2); nf.setMinimumFractionDigits(2); g.drawString("reintento = " + this.retry

+ ",error actual = "

+ nf.format(this.totalError \* 100)

+ "%, mejor error = "

+ nf.format(this.bestError \* 100)

+ "%", 0,

(int) getContentPane(). getBounds(). getHeight());

getContentPane(). getGraphics(). drawImage( this.offScreen, 0, 0, esto);

}

/\*\*

* Se llama para ejecutar el subproceso en segundo plano. El
* hilo de fondo configura el
* red neuronal y datos de entrenamiento y comienza la formación
* la red.

\*/

corrida pública sin efecto( ) {

construir el conjunto de entrenamiento

this.input = nuevo double[SAMPLE\_COUNT][INPUT\_COUNT];

para (int i = 0; i < SAMPLE\_COUNT; i++) {

para (int j = 0; j < INPUT\_COUNT; j++) { este.input[i][j] = Math.random();

}

}

construir y entrenar la red neuronal this.net = nuevo SelfOrganizingMap(INPUT\_COUNT,

OUTPUT\_COUNT,

NormalizaciónType.MULTIPLICATIVE); tren trainSelfOrganizingMap final =

nuevo TrainSelfOrganizingMap(

this.net, this.input,

LearningMethod.SUBTRACTIVE,0.5); train.initialize();

double lastError = Double.MAX\_VALUE; int errorCount = 0;

while (errorCount < 10) {

train.iteration(); this.retry++;

this.totalError = train.getTotalError(); this.bestError = train.getBestError(); pintura(getGraphics());

if (this.bestError < lastError) { lastError = this.bestError; errorCount = 0;

} else {

errorCount++;

}

}

}

}

Hay varias constantes que rigen la forma en que funciona el ejemplo de formación SOM.

Estas constantes se resumen en el Cuadro 11.4.

### Tabla 11.4: Constantes TestSOM

|  |  |  |
| --- | --- | --- |
| **constante** | **valor** | **propósito** |
| INPUT\_COUNT | 2 | Cuántas neuronas de entrada utilizar. |
| OUTPUT\_COUNT | 7 | Cuántas neuronas de salida usar. |
| SAMPLE\_COUNT | 100 | Cuántas muestras aleatorias generar. |

Hay dos componentes principales para este programa. El primero es el método **run,** que implementa el subproceso en segundo plano. El hilo de fondo procesa el tren del SOM. El segundo es el método de **pintura,** que muestra gráficamente la progress que está haciendo el proceso de entrenamiento. Estos dos métodos se discutirán en las dos secciones siguientes.

**Hilo de** fondo

Se utiliza un subproceso en segundo plano para procesar el SOM mientras se ejecuta la aplicación. Esto le permite ver el progreso del entrenamiento gráficamente. El método **run** controla el subproceso en segundo plano. La firma para el método de **ejecución** se muestra aquí:

carrera nula ()

En primer lugar, el conjunto de entrenamiento se crea utilizando números aleatorios. Estos son puntos aleatorios en una cuadrícula a la que el programa entrenará.

construir el conjunto de entrenamiento

this.input = nuevo double[SAMPLE\_COUNT][INPUT\_COUNT];

para (int i = 0; i < SAMPLE\_COUNT; i++) { para (int j = 0; j < INPUT\_COUNT; j++) {

this.input[i][j] = Math.random();

}

}

A continuación, se crea la red neuronal.

construir y entrenar la red neuronal

this.net = nuevo SelfOrganizingMap(INPUT\_COUNT, OUTPUT\_ COUNT,NormalizationType.MULTIPLICATIVE);

A continuación, se crea una clase de formación para entrenar el SOM.

tren TrainSelfOrganizingMap final = nuevo TrainSelfOrganizingMap( this.net, this.input);

El entrenador está inicializado.

train.initialize();

La **última variableError** se inicializa en un valor muy alto y el **errorCount** se establece en cero.

double lastError = Double.MAX\_VALUE; int errorCount = 0;

A continuación, bucle hasta que el error no ha mejorado para diez iteraciones.

while (errorCount < 10) {

Se procesa una iteración de entrenamiento y se mantiene el mejor error.

train.iteration(); this.retry++;

this.totalError = train.getTotalError(); this.bestError = train.getBestError();

La ventana se actualiza con la cuadrícula actual utilizando el método de **pintura** descrito en la siguiente sección.

pintura (getGraphics());

El mejor error se evalúa para determinar si ha habido una mejora. Si no hubo ninguna mejora, errorCount se incrementa en uno.

if (this.bestError < lastError) { lastError = this.bestError;

errorCount = 0;

} else { errorCount++;

}

}

El bucle continúa hasta que no ha habido ninguna mejora en el nivel de error para

diez iteraciones.

## Mostrando el progreso

El estado actual de la matriz de peso y el entrenamiento de la red neuronal se muestra llamando al método **de pintura.** La firma para el método **de pintura** se muestra aquí:

pintura del vacío público (Gráficos g)

Si no hay ninguna red definida, entonces no hay nada que dibujar, así que regresamos.

if (this.net == null) { return;

}

Para evitar parpadeo de pantalla, este programa utiliza una imagen fuera de pantalla para dibujar la cuadrícula.

Una vez dibujada la cuadrícula, la imagen de fondo se copia en la ventana. A continuación, el out-put se muestra en una sola pasada.

Las siguientes líneas de código comprueban si la imagen fuera de pantalla se ha creado todavía.

Si esta imagen no se ha creado, ahora se crea una.

if (this.offScreen == null) { this.offScreen = this.createImage((int) getBounds(). getWidth(),

(int) getBounds(). getHeight());

}

Se obtiene un objeto gráfico con el que se dibujará la imagen fuera de pantalla.

g = this.offScreen.getGraphics();

Se determinan las dimensiones de la ventana.

último ancho int = getContentPane(). getWidth(); altura final int = getContentPane(). getHeight();

El mínimo de la altura y anchura de la ventana se utiliza como tamaño para una sola unidad.

Toda la ventana está establecida en negro.

this.unitLength = Math.min(ancho, alto); g.setColor(Color.black);

g.fillRect(0, 0, anchura, altura);

Se obtienen los pesos de salida.

g.setColor(Color.white);

resultado final de matrixWeights = this.net.getOutputWeights();

Luego recorremos y mostramos las neuronas de salida. Estos corresponderán a los datos de entrenamiento aleatorios generados en la sección anterior.

para (int y = 0; y < outputWeights.getRows(); y++) {

Se dibujan rectángulos rellenos que corresponden a todas las neuronas de salida.

g.fillRect((int) (outputWeights.get(y, 0)

* + this.unitLength),

(int) (outputWeights.get(y, 1)

* + this.unitLength), 10, 10);

}

A continuación, se traza una cuadrícula de las muestras con las que probar la red. A continuación, determinamos

en cuál de las neuronas de salida se agrupa cada punto de entrenamiento.

g.setColor(Color.green);

para (int y = 0; y < this.unitLength; y += 50) { para (int x = 0; x < this.unitLength; x += 50) {

Se dibuja un óvalo en cada uno de los puntos de entrenamiento.

g.fillOval(x, y, 5, 5);

doble d final [] = nuevo doble[2];

El punto de formación se presenta al SOM.

d[0] = x;

d[1] = y;

A continuación, se obtiene la neurona ganadora.

final int c = this.net.winner(d);

Se determinan los pesos para la neurona ganadora.

final int x2 = (int) (outputWeights.get(c, 0)

* this.unitLength);

final int y2 = (int) (outputWeights.get(c, 1)

* this.unitLength);

A continuación, se traza una línea desde la muestra hasta la neurona ganadora.

g.drawLine(x, y, x2, y2);

}

}

A medida que avanza la capacitación, se obtiene información de estado y los números de salida

correctamente formateado.

mostrar la información de estado g.setColor(Color.white);

final NumberFormat nf = NumberFormat.getInstance(); nf.setMaximumFractionDigits(2); nf.setMinimumFractionDigits(2);

Se muestra una cadena de texto.

g.drawString("reintento = "

+ this.retry

+ ",error actual = "

+ nf.format(this.totalError \* 100)

+ "%, mejor error = "

+ nf.format(this.bestError \* 100)

+ "%", 0,

(int) getContentPane(). getBounds(). getHeight());

Ahora que la imagen fuera de la pantalla está lista, se muestra en la ventana.

getContentPane(). getGraphics(). drawImage(this.offScreen, 0, 0, esto);

El progreso de la formación ahora es visible para el usuario.

# Resumen del capítulo

En este capítulo aprendimos sobre el mapa autoorganizado. El mapa autoorganizador difiere de la red de backpropagation feedforward de varias maneras. El mapa auto-orga- nizing utiliza entrenamiento sin supervisión. Esto significa que recibe datos de entrada, pero no datos de salida anticipados. A continuación, asigna las muestras de entrenamiento a cada una de sus neuronas de salida.

Un mapa autoorganizado contiene solo dos capas. La red se presenta con un patrón de entrada que se pasa a la capa de entrada. Este patrón de entrada debe normalizarse en números entre –1 y 1. La salida de esta red neuronal será una sola neurona de salida ganadora. Las neuronas de salida se pueden considerar como grupos que el mapa autoorganizado ha clasificado.

Para entrenar el mapa autoorganizado, lo presentamos con los elementos de entrenamiento y vemos qué neurona de salida "gana". Los pesos de esta neurona ganadora se modifican para que responda aún más fuertemente al patrón que hizo que ganara la próxima vez que se encuentre el patrón.

También puede haber un caso en el que una o más neuronas no ganan nunca. Tales neuronas son peso muerto en la red neuronal. Debemos identificar estas neuronas y ajustarlas para que reconozcan patrones que ya son reconocidos por otras neuronas más "sobrecargadas de trabajo". Esto permitirá que la carga del reconocimiento caiga más uniformemente sobre las neuronas de salida.

Este capítulo presentó sólo un ejemplo simple del mapa autoorganizándose. En el siguiente capítulo aplicaremos el mapa autoorganizado a una aplicación del mundo real. Veremos cómo utilizar el mapa autoorganizándose para reconocer la escritura a mano.

# vocabulario

Aprendizaje competitivo de ajuste de peso aditivo

Normalización de entrada Normalización multiplicativa Autoorganización mapa ajuste de peso sustractivo Z-Axis Normalización

# Preguntas para revisión

1. ¿Cuántas capas ocultas se utilizan normalmente con un mapa autoorganizado? ¿Cuáles son sus papeles?
2. ¿Para qué tipos de problemas se utilizan normalmente los mapas autoorganización?
3. ¿En qué se diferencia el aprendizaje competitivo del aprendizaje que se presentó anteriormente en este libro, como la retropropagación o los algoritmos genéticos?
4. ¿Qué salida produce un mapa autoorganizándose y qué representa esta salida?
5. ¿Cuál es la principal ventaja de la normalización del eje z sobre la multiplicación ni la malización? ¿Cuándo podría ser más útil la normalización multiplicativa?