**CHAPTER** **4:** **HOW** **A** **MACHINE** LEARS

* Comprensión de capas
* Formación supervisada
* Entrenamiento sin supervisión
* Cálculo de errores
* Entender la regla de Hebb y la regla delta

Hay muchas maneras diferentes que una red neuronal puede aprender; sin embargo, cada algoritmo de aprendizaje implica la modificación de la matriz de peso, que contiene los pesos para las conexiones entre las neuronas. En este capítulo, examinaremos algunos de los métodos populares más utilizados para ajustar estos pesos. En el capítulo 5, "The Feedforward Backpropagation Neural Network", seguiremos esta discusión con una introducción al método de formación de backpropagation. La backpropagation es uno de los métodos de formación de red neural más comunes utilizados hoy en día.

# Métodos de aprendizaje

El entrenamiento es un proceso muy importante para una red neuronal. Hay dos formas de capacitación que pueden ser empleadas, supervisadas y sin supervisión. Entrenamiento supervisado en implica proporcionar a la red neuronal un conjunto de datos de entrenamiento y la salida prevista. En el entrenamiento sin supervisión, la red neuronal también se proporciona con conjuntos de entrenamiento, pero no con salidas anticipadas. En este libro, examinaremos tanto la formación supervisada como la no supervisada. Este capítulo proporcionará una breve introducción a cada enfoque. A continuación, estarán cubiertos con mucho más detalle en capítulos posteriores.

## Entrenamiento sin supervisión

¿Qué significa entrenar una red neuronal sin supervisión? Como se mencionó anteriormente, la red neuronal se proporciona con conjuntos de entrenamiento, que son colecciones de valores de entrada definidos. La red neuronal no supervisada no está provista de salidas anticipadas.

El entrenamiento sin supervisión se utiliza normalmente para entrenar redes neuronales de clasificación. Una red neuronal de clasificación recibe patrones de entrada, que se presentan a las neuronas de entrada. Estos patrones de entrada se procesan, causando una sola neurona en la capa de salida para disparar. Esta neurona de disparo proporciona la clasificación para el patrón e identifica a qué grupo pertenece el patrón.

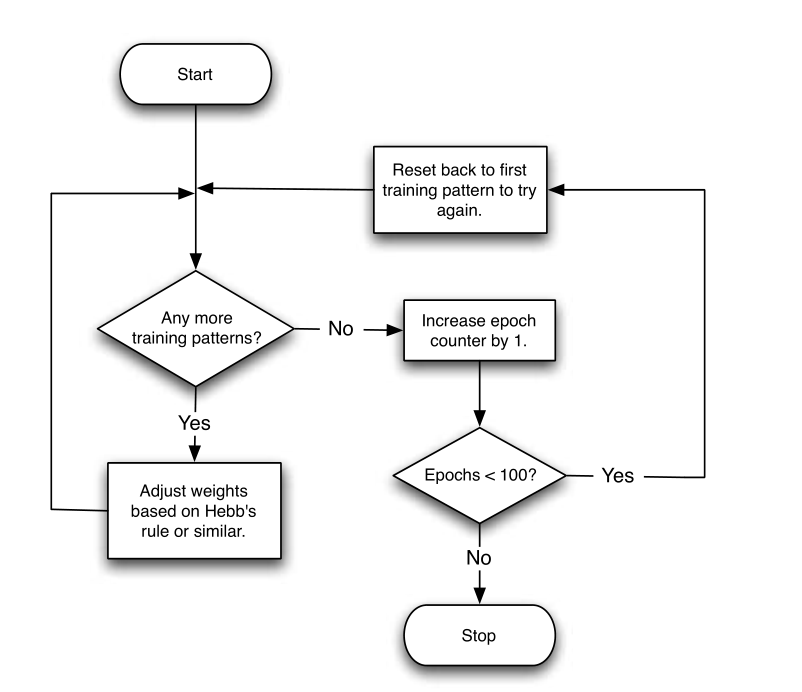
Otra aplicación común para la formación no supervisada es la minería de datos. En este caso, usted tiene una gran cantidad de datos para ser buscado, pero es posible que no sepa exactamente lo que está buscando. Desea que la red neuronal clasifique estos datos en varios grupos. No desea dictar a la red neuronal con anticipación qué patrón de entrada debe clasificarse en qué grupo. A medida que la red neuronal entrena, los patrones de entrada caen en grupos con otras entradas que tienen características similares. Esto le permite ver qué patrones de entrada comparten similitudes.

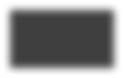
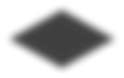
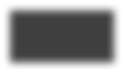
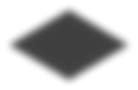
El entrenamiento sin supervisión es una técnica de entrenamiento muy común para self-organizing maps (SOM), también llamados redes neuronales de Kohonen. En el capítulo 11, discutiremos cómo construir un SOM e introduciremos el proceso general para capacitarlos sin supervisión.

En el capítulo 12, "OCR and the Self-Organizing Map", se le mostrará una aplicación prac- tical de un SOM. El programa de ejemplo presentado en el capítulo 12, que está diseñado para leer a mano, aprende a través del uso de un método de formación sin supervisión. Los patrones de entrada presentados al SOM son imágenes de puntos de caracteres escritos a mano y hay 26 neuronas de salida, que corresponden a las 26 letras del alfabeto inglés. A medida que se entrena el SOM, los pesos se ajustan para que los patrones de entrada se puedan clasificar en estos 26 grupos. Como se demostrará en el capítulo 12, esta técnica da como resultado un método relativamente eficaz para el reconocimiento de caracteres.

Como puede ver, el entrenamiento no supervisado se puede aplicar a una serie de situaciones. Se tratará con mucho más detalle en los capítulos 11 y 12. La Figura 4.1 ilustra el flujo de información a través de un algoritmo de entrenamiento no supervisado.

**Figure** **4.1:** Entrenamiento sin **supervisión.**

****



Empezar

Más

patrón de entrenamiento?

No

Sí

¿Épocas < 100?

Sí

No

Aumente el contador contador de época en 1.

Ajuste los pesos en función de la regla de Hebb o similar.

Restablezca el primer patrón de entrenamiento para intentarlo de nuevo.

## Supervisado Entrenamiento



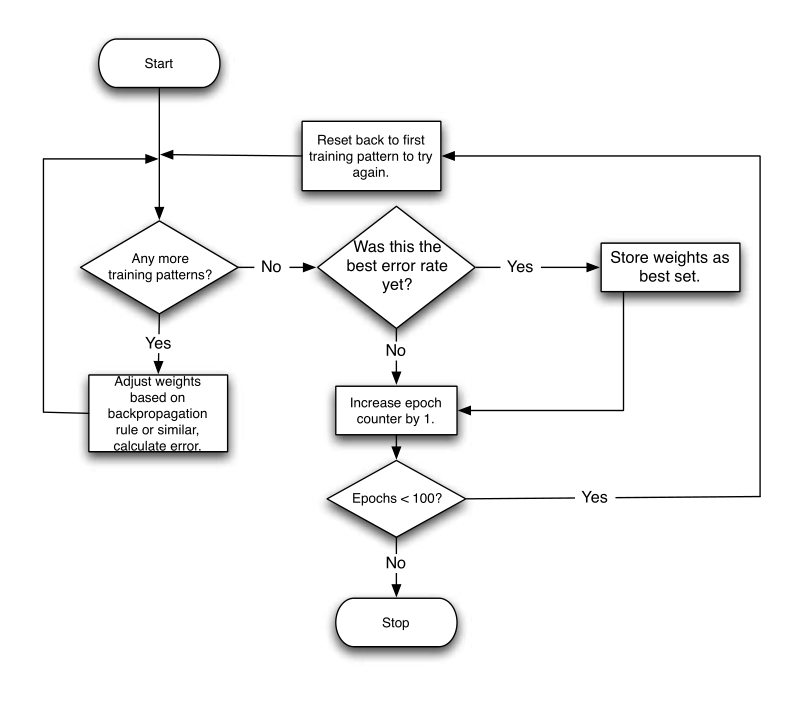
Parada

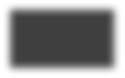
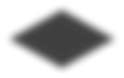
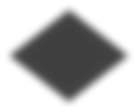
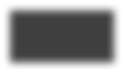
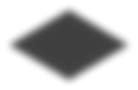
El método de capacitación supervisado es similar al método de capacitación no supervisado, en que se proporcionan conjuntos de entrenamiento. Al igual que con el entrenamiento no supervisado, estos conjuntos de entrenamiento especifican señales de entrada a la red neuronal. La principal diferencia entre la capacitación supervisada y no supervisada es que en la formación supervisada se proporcionan los outputs esperados. Esto permite a la red neuronal ajustar los valores de la matriz de peso en función de las diferencias entre la salida prevista y la salida real.

Hay varios algoritmos de entrenamiento supervisados populares. Uno de los más comunes es el algoritmo de backpropagation. La backpropagation se discutirá en el próximo capítulo. También es posible utilizar recocido simulado o algoritmos genéticos para implementar entrenamiento supervisado. El recocido simulado y los algoritmos genéticos serán desencantados en los capítulos 6, "Entrenamiento usando un algoritmo genético", y el capítulo 7, "Entrenamiento usando recocido simulado". Ahora discutiremos cómo se calculan los errores para los algoritmos de entrenamiento supervisados y no supervisados.

La Figura 4.2 ilustra el flujo de información a través de una formación supervisadaalgo- rithm.

**Figura 4.2: Formación** **supervisada.**





Empezar

¿Fue este el

mejor tasa de Error ¿Todavía?

Más

patrón de entrenamiento?

No

Sí

Sí

No

¿Épocas < 100?

Sí

No

Parada

Aumente el contador contador de época en 1.

Almacene los pesos perdidos como mejor conjunto.

Ajustar pesas

basado en la regla de backpropagation o similar, cálculo el error.

Restablezca el primer patrón de entrenamiento para intentarlo de nuevo.

# Cálculo de errores

El cálculo de errores es un aspecto importante de cualquier red neuronal. Ya sea que la red neuronal esté supervisada o sin supervisión, se debe calcular una tasa de error. El objetivo de prácticamente todos los algoritmos de entrenamiento es minimizar la tasa de error. En esta sección, examinaremos cómo se calcula la tasa de error para una red neuronal supervisada. También discutiremos cómo se determina la tasa de error para un algoritmo de entrenamiento no supervisado. Comenzaremos esta sección examinando dos pasos de cálculo de errores utilizados para la capacitación supervisada.

## Cálculo de errores y formación supervisada

Hay dos valores que deben tenerse en cuenta para determinar la tasa de error para la formación supervisada. En primer lugar, debemos calcular el error para cada elemento del conjunto de entrenamiento a medida que se procesa. En segundo lugar, debemos calcular el promedio de los errores para todos los elementos del conjunto de entrenamiento en cada muestra. Por ejemplo, considere el operador lógico XOR del capítulo 1 que solo tiene cuatro elementos en su conjunto de entrenamiento. Refiera al cuadro 1.3 para revisar el operador lógico XOR.

En el capítulo 1, seleccionamos intuitivamente valores para la matriz de peso. Esto está bien para una red neuronal simple, como la que implementa el operador XOR. Sin embargo, esto no es práctico para redes neuronales más complejas. Normalmente, el proceso implica crear una matriz de peso aleatoria y probar cada fila del conjunto de entrenamiento. A continuación, se calcula un error de salida para cada elemento del conjunto de entrenamiento. Por último, una vez procesados todos los elementos del conjunto de entrenamiento, se determina el error del cuadrado medio raíz (RMS) para todos ellos.

## Error de salida

El error de salida es simplemente un cálculo de error que se realiza para determinar cuán diferente es la salida de una red neuronal de la salida ideal. Este valor rara vez se utiliza para ningún propósito que no sea como un paso a paso en el cálculo del error cuadrado medio raíz (RMS) para todo el conjunto de entrenamiento. Una vez que todos los elementos de un conjunto de entrenamiento se han ejecutado a través de la red, se puede calcular el error RMS. Este error actúa como la tasa global de error para toda la red neuronal.

Crearemos una clase genérica de cálculo de errores que se utilizará en todas las redes neuronales de este libro. Esta clase se denomina **ErrorCalculation**. Esta clase funciona calculando el error de salida para cada miembro del set de entrenamiento. Este error es al-lowed para crecer hasta que se han presentado todos los elementos del conjunto de entrenamiento. A continuación, se calcula el error RMS. El cálculo del error RMS se cubre en la siguiente sección. La clase **ErrorCalculation** se muestra en el listado 4.1.

### Listado 4.1: La clase ErrorCalculation (ErrorCalculation.java)

package com.heatonresearch.book.introneuralnet.neural.util;

/\*\*

* ErrorCalculation: An implementation of root mean square (RMS)
* error calculation. This class is used by nearly every neural
* network in this book to calculate error.

\*

* @author Jeff Heaton
* @version 2.1

\*/

public class ErrorCalculation { private double globalError; private int setSize;

/\*\*

* + Returns the root mean square error for a complete
  + training set.

\*

* + @param len
  + The length of a complete training set.
  + @return The current error for the neural network.

\*/

public double calculateRMS() {

final double err = Math.sqrt(this.globalError / (this.setSize));

return err;

}

/\*\*

* + Reset the error accumulation to zero.

\*/

public void reset() { this.globalError = 0;

this.setSize = 0;

}

/\*\*

* + Called to update for each number that should be checked.
  + @param actual The actual number.
  + @param ideal The ideal number.

\*/

public void updateError(final double actual[], final double ideal[]) {

for (int i = 0; i < actual.length; i++) {

final double delta = ideal[i] - actual[i]; this.globalError += delta \* delta; this.setSize += ideal.length;

}

}

}En primer lugar, veremos cómo utilizar la clase **ErrorCalculation.** Cuando se produce un error

para calcular, normalmente habrá dos matrices, la matriz **ideal** y la matriz **real.** La matriz **ideal** contiene los valores que esperamos que produzca la red neuronal. Una matriz **ideal** se muestra aquí:

double ideal[][] = {

{1,2,3,4},

{5,6,7,8},

{9,10,11,12},

{13,14,15,16} };

Esta matriz **ideal** contiene cuatro conjuntos, que son las filas de la matriz. Cada fila contiene cuatro números. La red neuronal que esta matriz se utilizaría para entrenar tendría, por lo tanto, cuatro neuronas de salida. Las columnas corresponden a las neuronas de salida. Las filas son las elementos individuales del conjunto de entrenamiento.

La salida real de la red neuronal se almacena en una matriz **real.** Si estuviéramos entrenando una red neuronal, tendríamos una matriz de valores de entrada que darían como resultado los valores de salida reales. Aquí se proporciona una matriz **real** hipotética para su uso con la matriz **ideal** anterior.

double real[][] = {

{1,2,3,5},

{5,6,7,8},

{9,10,11,12},

{13,14,15,16} };

Como puede ver, la matriz **real** está bastante cerca de la matriz **ideal.** Un

objeto tipo **ErrorCalulation** ahora se crea una instancia de **error** con nombre

ErrorCalculation error = nuevo ErrorCalculation();

La salida para cada elemento del conjunto de entrenamiento ahora debe compararse con la salida ideal. Recorremos las cuatro filas de las matrices.

for(int i=0;i<ideal.length;i++) { error.updateError(real[i], ideal[i]);

}

Se presentan las filas correspondientes de la matriz **real** y la matriz **ideal**

alobjeto de error y se actualiza el **objeto de error.**

Por último, se calcula e imprime el error RMS.

System.out.println( error.calculateRMS(ideal.length, ideal[0]. longitud) );

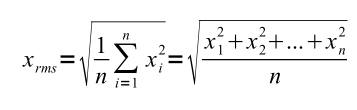
El objeto **de error** se puede reutilizar para otro cálculo de error, si es necesario. Simplemente llame al método de **restablecimiento** y el objeto de error está listo para ser utilizado de nuevo.

Esta clase **CalculateError** se utilizará con frecuencia en este libro. Cada vez que se necesite el error RMS para una red neuronal, se utilizará esta clase. En la siguiente sección se describirá cómo se calcula el error RMS.

## Root Mean Square (RMS) Error

El método RMS se utiliza para calcular la tasa de error de un conjunto de entrenamiento basado en resultados ideales predefinidos. El método RMS es eficaz para calcular la tasa de error independientemente de si los resultados reales están por encima o por debajo de los resultados ideales. Par Calcular el RMS para una serie de **n** valores de **x**, considere la ecuación 4.1.

### Ecuación 4.1: Error cuadrado medio raíz (RMS)

*x* =√1

1 2 *n*

*N*

∑ x2=

√*x*2+*x*2+.. . +*x*2

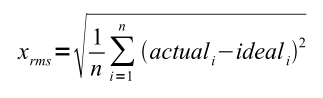
*Rms*

*n i*=1 *i n*

Los valores de **x** se cuadran y su suma se divide por **n**. Cuadrar los valores elimina el problema asociado a que algunos valores están por encima de los valores ideales y otros por encima, ya que la computación del cuadrado de un valor siempre resulta en un número positivo.

Para aplicar RMS a la salida de una red neuronal, considere la Ecuación 4.2.

### Ecuación 4.2: RMS para una red neuronal



*x* =√1 ∑

*N*

(*real* −*ideal* ) 2

*rms n* *i*=1 *i i*

Para calcular el RMS para las matrices de la sección anterior, calcularía la diferencia entre los resultados reales y los resultados ideales, como se muestra en la ecuación anterior. A continuación, se calcularía el cuadrado de cada uno de ellos y se sumarían los resultados. La suma se dividiría entonces por el número de elementos, y la raíz cuadrada del resultado de ese cálculo proporcionaría la tasa de error.

Para implementar esto en Java, se llama al método **updateError** para comparar la salida producida para cada elemento del conjunto de entrenamiento con los valores de salida ideales para la red neural. La firma para el método **updateError** se muestra aquí:

public void updateError(doble real[],double ideal[]) {

En primer lugar, recorremos todos los elementos de la matriz **real.**

para (int i = 0; i < real.length; i++) {

Determinamos la diferencia, o **delta,**entre los valores reales y los ideales. No importa si se trata de un número negativo; que se manejará en el siguiente paso.

delta doble = ideal[i] – real[i];

A continuación, agregamos el cuadrado de cada **delta** a la variable **globalError.** El

**setSize** variable se utiliza para realizar un seguimiento de cuántos elementos se han procesado.

globalError += delta \* delta; setSize += ideal.length;

}

}

Por último, una vez que todos los elementos en el conjunto de entrenamiento se han completado un ciclo a través del metodo updateError, se llama al método **calculateRMS** para calcular el error RMS. La firma para el método **calculateRMS** se muestra aquí.

cálculo doble públicoRMS() {

Calculamos el error como la raíz cuadrada del **globalError** dividido por el

**setSize** y devuelva el error.

doble err = Math.sqrt(globalError / ((double)setSize)); volver err;

}

Una vez que el objeto **CalculateError** se ha utilizado para calcular la tasa de error, debe restablecerse antes de que se pueda procesar otro conjunto de entrenamiento. De lo contrario, la variable **globalError** seguiría creciendo, en lugar de partir de cero. Para restablecer la clase **CalculateError,** se debe llamar al método **reset.**

## Cálculo de errores y entrenamiento sin supervisión

Hemos discutido cómo se calculan los errores para la capacitación supervisada, ahora debemos discutir cómo se calculan para la capacitación no supervisada. Esto puede no ser bastante obvio. ¿Cómo se puede calcular un error cuando no se proporcionan salidas ideales? El procedimiento exacto por el cual se hace esto se tratará en el capítulo 11, " Using a SelfOrganizing Map." Por ahora, simplemente destacaremos los detalles más importantes del proceso.

La mayoría de las redes neuronales no supervisadas están diseñadas para clasificar los datos de entrada. Los datos de entrada se clasifican en función de una de las neuronas de salida. El grado en que cada neurona de salida dispara para los datos de entrada se estudia con el fin de producir un error para el entrenamiento sin supervisión. Idealmente, nos gustaría que una sola neurona disparara a un alto nivel para cada miembro del conjunto de entrenamiento. Si este no es el caso, ajustamos los pesos a la neurona con el mayor número de disparos, es decir, la neurona ganadora consolida su victoria. Este método de entrenamiento hace que cada vez más neuronas disparen para los diferentes elementos en el conjunto de entrenamiento.

# Algoritmos de entrenamiento

El entrenamiento ocurre a medida que los pesos de conexión de las neuronas se modifican para producir resultados más deseables. Hay varias maneras en que la capacitación puede tener lugar. En las siguientes secciones discutiremos dos métodos simples para entrenar los pesos de conexión de una red neuronal. En el capítulo 5, examinaremos la contrapropagación, que es un algoritmo de entrenamiento mucho más complejo.

Los pesos de conexión de las neuronas no se modifican en un solo paso. El proceso por el cual se modifican los pesos de las neuronas ocurre sobre múltiples iteraciones. La red neuronal se presenta con datos de entrenamiento y luego se observan los resultados. El aprendizaje de la red neuronal se produce cuando estos resultados cambian el peso de la conexión. El proceso exacto por el que esto sucede viene determinado por el algoritmo de aprendizaje utilizado.

Los algoritmos de aprendizaje, que comúnmente se denominan reglas de aprendizaje, casi siempre se expresan como funciones. Una función de aprendizaje proporciona orientación sobre cómo se debe cambiar un peso entre dos neuronas. Considere una matriz de peso que contiene los pesos para las conexiones entre cuatro neuronas, como vimos en el capítulo 3, "Uso de una red neuronal hopfield." Esto se expresa como una matriz de dobles.

pesos dobles[][] = nuevo doble[4][4];

Esta matriz se utiliza para almacenar los pesos entre cuatro neuronas. Dado que los índices de matriz Java comienzan con cero, nos referiremos a estas neuronas como neuronas de cero a tres. Usando la matriz anterior, el peso entre la neurona dos y la neurona tres estaría contenido en los ubicación **weights[2] [3]**. Por lo tanto, nos gustaría una función de aprendizaje que devuelva el nuevo peso entre las neuronas "i" y "j", de tal manera que

pesos[i][j] += learningRule(...)

El método hipotético **learningRule** calcula el cambio (delta) que debe ocurrir entre las dos neuronas para que el aprendizaje se lleve a cabo. Nunca descartamos el valor de peso anterior por completo; más bien, calculamos un valor delta que se utiliza para modificar el peso original. Se necesita más de una sola modificación para que la red neural aprenda. Una vez que los pesos de la red neuronal han sido modificados, la red se presenta de nuevo con los datos de entrenamiento y el proceso continúa. Estas iteraciones continúan hasta que la tasa de error de la red neuronal ha caído a un nivel aceptable.

Otra entrada común a la regla de aprendizaje es el error. El error es el grado en que la salida real de la red neuronal difiere de la salida prevista. Si se proporciona un error de este tipo a la función de entrenamiento, el método se denomina entrenamiento supervisado. En el entrenamiento supervisado, la red neuronal está ajustando constantemente los pesos para intentar alinear mejor los resultados reales con las salidas previstas que se proporcionaron.

Por el contrario, si no se proporcionó ningún error a la función de entrenamiento, entonces estamos usando un algoritmo de entrenamiento no supervisado. Recuerde, en el entrenamiento sin supervisión, el trabajo de la red neuronal no se dice cuál es la salida "correcta". El entrenamiento sin supervisión deja la red neuronal para determinar esto por sí mismo. A menudo, el entrenamiento no supervisado se utiliza para permitir que la red neuronal agrupe los datos de entrada. El programador no sabe con anticipación exactamente cuáles serán las agrupaciones.

Ahora examinaremos dos algoritmos de entrenamiento comunes. La primera, la regla de Hebb, se utiliza para el entrenamiento sin supervisión y no tiene en cuenta el error de red. El segundo, la regla delta, se utiliza con entrenamiento supervisado y ajusta los pesos para que la entrada a la red neuronal produzca con mayor precisión la salida prevista. Comenzaremos con la Regla de Hebb.

## Regla de Hebb

Uno de los algoritmos de aprendizaje más comunes se llama Hebb's Rule. Esta regla fue desarrollada por Donald Hebb para ayudar con el entrenamiento no supervisado. Anteriormente examinamos una hipotética regla de aprendizaje definida por la siguiente expresión:

pesos[i][j] += learningRule(...)

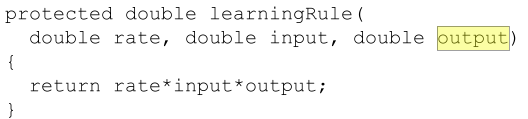
Las reglas para entrenar redes neuronales casi siempre se representan como fórmulas algebraicas. La regla de Hebb se expresa en la Ecuación 4.3.

### Ecuación 4.3: Regla de Hebb



Δ *wij*=y *ai* *ji*

La ecuación anterior calcula el cambio necesario (delta) en el peso para la conexión de la neurona "i" a la neurona "j." La letra griega mu (μ) representa la tasa de aprendizaje. La activación de cada neurona se administra como **ai** y **aj.** Esta ecuación se puede traducir fácilmente al siguiente método Java.

doble aprendizaje protegidoRule(

doble velocidad, doble entrada, doble salida)

{

tasa de retorno\*entrada\*salida;

}

Ahora examinaremos cómo funciona realmente este algoritmo de entrenamiento. Para ello, considerará una red neuronal simple con sólo dos neuronas. En esta red neuronal, estas dos neuronas componen la capa de entrada y salida. No hay ninguna capa oculta. El Cuadro 4.1 resume algunos de los posibles escenarios utilizando la capacitación de Hebbian. Supongamos que la tasa de aprendizaje es 1.

### Tabla 4.1: Uso de la regla de Hebb

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caso** | **Neuron** **i** value | **Neuron** **j** output | **Regla** de Hebb | **Peso Delta** |
| **Caso 1** | +1 | -1 | 1\*1\*-1 | -1 |
| **Caso 2** | -1 | +1 | 1\*-1\*1 | -1 |
| **Caso 3** | +1 | +1 | 1\*1\*1 | +1 |

Como se puede ver en la tabla anterior, si la activación de la neurona "i" fue +1 y la activación de la neurona j fue -1, el peso de conexión de la neurona entre la neurona "i"y la neurona "j" se reduciría en uno.

La regla de Hebb no está supervisada, por lo que no estamos entrenando la red neuronal para una salida ideal. Más bien, la regla de Hebb funciona reforzando lo que la red neuronal ya lo sabe. Esto a veces se resume con la pegadiza frase: "Neuronas que disparan juntas, alambre juntos." Es decir, si las dos neuronas tienen activaciones similares, su peso aumenta. Si dos neuronas tienen activaciones diferentes, su peso se desenroscó.

Un ejemplo de la regla de Hebb se muestra en el listado 4.2.

### Listado 4.2: Uso de la regla de Hebb (Hebb.java)

paquete com.heatonresearch.book.introneuralnet.ch4. hebb;

/\*\*

* Capítulo 4: Aprendizaje automático

\*

* Hebb: Aprende, usando la regla de Hebb.

\*

* @author Jeff Heaton
* @version 2,1

\*/

clase pública Hebb {

/\*\*

* + El método principal solo es un objeto delta y llama
  + Ejecutar.

\*

* + args @param
  + No se utiliza

\*/

vacío estático público principal ( args de cadena final []) { último Hebb delta = nuevo Hebb(); delta.run();

}

/\*\*

* + Peso para la neurona 1

\*/

doble w1;

/\*\*

* + Peso para la neurona 2

\*/

doble w2;

/\*\*

* + Tasa de aprendizaje

\*/

doble velocidad = 1,0;

/\*\*

* + Época actual #

\*/

int epoch = 1;

Hebb público() {

this.w1 = 1;

this.w2 = -1;

}

/\*\*

* + Procesa una época. Aquí aprendemos de los tres
  + muestras de entrenamiento y luego
  + actualizar las ponderaciones en función del error.

\*/

época del vacío protegido() { System.out.println("\*\*\*Beginning Epoch #"

+ this.epoch + "\*\*\*"); presentPattern(-1, -1);

presentPattern(-1, 1);

presentPattern(1, -1);

presentPattern(1, 1); this.epoch++;

}

/\*\*

* + Presenta un patrón y aprende de él.

\*

* + @param i1
  + Entrada a la neurona 1
  + @param i2
  + Entrada a la neurona 2
  + @param i3
  + Entrada a la neurona 3
  + @param previsto
  + La producción prevista

\*/

vacío protegido presentePattern(doble final i1, doble final i2) {

doble resultado; delta doble;

ejecutar la red como está en los datos de entrenamiento

y obtener el error

System.out.print("Presentado [" + i1 + "," + i2 + "]"); resultado = reconocer (i1, i2);

System.out.print(" result=" + result);

ajustar el peso 1

delta = trainingFunction(this.rate, i1, result); this.w1 += delta;

System.out.print(",delta w1=" + delta);

ajustar el peso 2

delta = trainingFunction(this.rate, i2, result); this.w2 += delta;

System.out.println(",delta w2=" + delta);

}

/\*\*

* + @param i1
  + Entrada a la neurona 1
  + @param i2
  + Entrada a la neurona 2
  + @param i3
  + Entrada a la neurona 3
  + @return la salida de la red neuronal

\*/

doble reconocimientoprotegido ( doble final i1, doble final i2) { doble final a = (this.w1 \* i1) + (this.w2 \* i2); retorno (a \* .5);

}

/\*\*

* + Este método recorre 10 épocas.

\*/

ejecución pública nula () {

para (int i = 0; i < 5; i++) { epoch();

}

}

/\*\*

* + El learningFunction implementa la regla de Hebb. Esto
  + método volverá
  + el ajuste de peso para la neurona de entrada especificada.

\*

* + tasa de @param
  + La tasa de aprendizaje
  + @param entrada
  + La neurona de entrada que estamos procesando
  + @param error
  + El error entre la salida real y
  + salida anticipada.
  + @return La cantidad para ajustar el peso.

\*/

entrenamiento doble protegidoFunction(doblevelocidad final, doble entrada final,

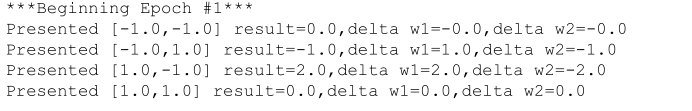
salida doble final) { tasa de retorno \* entrada \* salida;

}

}

El ejemplo de regla del Hebb utiliza dos neuronas de entrada y una neurona de salida. Como resultado, hay un total de dos pesos, un peso para cada una de las conexiones entre la neuron de entrada y la neurona de salida. El primer peso, que es el peso entre la neurona uno y la neurona de salida se inicializa a uno. El segundo peso, que es el peso entre la neurona dos y la neurona de salida, se inicializa a dos.

Considere la salida de la primera época.

Comenzando epoch #1\*\*\*

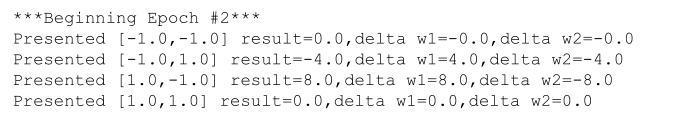
Presentado [-1. 0,-resultado1.0] =0.0,delta w1=-0.0,delta w2=-0.0 Presentado [-1.0,1.0] resultado=-1.0,delta w1=1.0,delta w2=-1.0 Presentado [Resultado 1.0,-1.0] =2.0,delta w1=2.0,delta w2=-2.0 Presentado [1.0,1.0] result=0.0,delta w1=0.0,delta w2=0.0

La salida anterior muestra cómo la red de tres neuronas respondió a cuatro patrones de entrada diferentes. Los dos patrones de entrada medios devolvieron los resultados más fuertes. El segundo patrón era fuerte en la dirección negativa, el tercer patrón era fuerte en la dirección positiva. La regla de Hebb tiende a fortalecer la producción en la dirección hacia la que ya tiene una tendencia.

La salida anterior también muestra las deltas calculadas para el peso uno (w1) y el peso dos (w2). El primero y el cuarto patrones produjeron salidas de cero, por lo que ninguno producirá un delta para el peso, aparte de cero. Sin embargo, la salida negativa del patron dos producirá un delta de peso de –1, y el resultado positivo del patrón tres producirá un delta de peso de 2. Estos pesos delta se aplicarán y fortalecerán las tendencias negativas o positivas de las respectivas neuronas.

También es importante tener en cuenta que este ejemplo siempre está aplicando las deltas a medida que se presentan los patrones. Esta es la razón por la que el tercer patrón siempre tendrá un out-put más fuerte que el segundo patrón; el delta para el segundo patrón ya ha sido applied para cuando se presenta el tercer patrón.

La segunda época continúa de la misma manera que la primera.

Comenzando epoch #2\*\*\*

Presentado [-1. 0,-Resultado 1.0] =0.0,delta w1=-0.0,delta w2=-0.0 Presentado [-1.0,1.0] resultado=-4.0,delta w1=4.0,delta w2=-4.0 Presentado [1.0,-1.0] resultado= 8.0,delta w1=8.0,delta w2=-8.0 Presentado [1.0,1.0] resultado=0.0,delta w1=0.0,delta w2=0.0

Sin embargo, ya se han aplicado los deltas de la época anterior. Se calculan nuevos deltas de peso que mejoran aún más las tendencias positivas o negativas de las neuronas.

## Regla Delta

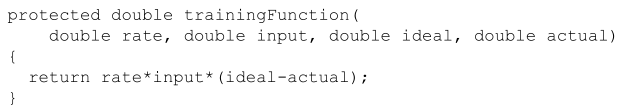
La regla delta también se conoce como la regla de error cuadrado (LMS) menos media. Con esta regla, la salida real de una red neuronal se compara con la salida prevista. Dado que se especifica la salida prevista, el uso de la regla delta se considera entrenamiento supervisado. Algebraicamente, la regla delta está escrita de la siguiente manera en la Ecuación 4.4.

### Ecuación 4.4: La Regla Delta

**

Δ *wij*=2 y *xi*(*ideal*−*real* )*j*

La ecuación anterior calcula el cambio necesario (delta) en pesos para la conexión de la neurona "i" a la neurona "j." La letra griega mu (μ) representa la tasa de aprendizaje. El **ideal** variable representa la salida deseada de la neurona "j". La variable **real** representa la salida real de la neurona "j". Como resultado, **(ideal-real)** es el error. Esta ecuación se puede traducir fácilmente en el método java fol- lowing.

formación doble protegidaFunction(

doble velocidad, doble entrada, doble ideal, doble real)

{

tasa de retorno\*entrada\*(ideal-real);

}

Ahora examinaremos cómo funciona realmente el algoritmo de entrenamiento delta. Para ver esto, veremos el programa de ejemplo que se muestra en el Listado 4.3.

### Listado 4.3: Uso de la regla Delta (Delta.java)

paquete com.heatonresearch.book.introneuralnet.ch4. delta;

/\*\*

* Capítulo 4: Aprendizaje automático

\*

* Delta: Aprenda, utilizando la regla delta.

\*

* @author Jeff Heaton
* @version 2,1

\*/

clase pública Delta {

/\*\*

* + El método principal sólo es un objeto delta y
  + las llamadas se ejecutan.

\*

* + args @param
  + No se utiliza

\*/

vacío estático público principal (final String args[]) { delta delta final = nuevo Delta(); delta.run();

}

/\*\*

* + Peso para la neurona 1

\*/

doble w1;

/\*\*

* + Peso para la neurona 2

\*/

doble w2;

/\*\*

* + Peso para la neurona 3

\*/

doble w3;

/\*\*

* + Tasa de aprendizaje

\*/

doble velocidad = 0,5;

/\*\*

* + Época actual #

\*/

int epoch = 1;

/\*\*

* + Procesa una época. Aquí aprendemos de los tres entrenamientos
  + muestras y luego
  + actualizar las ponderaciones en función del error.

\*/

época del vacío protegido() { System.out.println("\*\*\*Beginning Epoch #"

+ this.epoch + "\*\*\*");

|  |  |  |  |
| --- | --- | --- | --- |
| presentPattern(0, | 0, | 1, | 0); |
| presentPattern(0, | 1, | 1, | 0); |
| presentPattern(1, | 0, | 1, | 0); |

presentPattern(1, 1, 1, 1); this.epoch++;

}

/\*\*

* + Este método calculará el error entre el
  + producción anticipada y
  + la salida real.

\*

* + @param real
  + La salida real de la red neuronal.
  + @param previsto
  + La producción de neuronas prevista.
  + @return El error.

\*/

doble getErrorprotegido(doble real final, doble esperado final) {

retorno (anticipado - real);

}

/\*\*

* + Presenta un patrón y aprende de él.

\*

* + @param i1
  + Entrada a la neurona 1
  + @param i2
  + Entrada a la neurona 2
  + @param i3
  + Entrada a la neurona 3
  + @param previsto
  + La producción prevista

\*/

vacío protegido presentePattern(doble final i1, doble final i2,

final doble i3, doble doble previsto final) { doble error;

doble real; delta doble;

ejecutar la red como está en los datos de entrenamiento

y obtener el error

System.out.print("Presentado [" + i1 + "," + i2 + ","

+ i3 + "]");

real = reconocer (i1, i2, i3);

error = getError(real, anticipado); System.out.print(" anticipated=" + anticipado);

System.out.print(" real=" + real); System.out.println(" error=" + error);

ajustar el peso 1

delta = trainingFunction(this.rate, i1, error); this.w1 += delta;

ajustar el peso 2

delta = trainingFunction(this.rate, i2, error); this.w2 += delta;

ajustar el peso 3

delta = trainingFunction(this.rate, i3, error); this.w3 += delta;

}

/\*\*

* + @param i1
  + Entrada a la neurona 1
  + @param i2
  + Entrada a la neurona 2
  + @param i3
  + Entrada a la neurona 3
  + @return la salida de la red neuronal

\*/

doble reconocimiento protegido (doblefinal i1, doble final i2, doble final i3) {

final doble a = (this.w1 \* i1) + (this.w2 \* i2)

+ (this.w3 \* i3); retorno (a \* .5);

}

/\*\*

* + Este método recorre 100 épocas.

\*/

ejecución pública nula () {

para (int i = 0; i < 100; i++) { epoch();

}

}

/\*\*

* + LearningFunction implementa la regla delta. Esto
  + método volverá
  + el ajuste de peso para la neurona de entrada especificada.

\*

* + tasa de @param
  + La tasa de aprendizaje
  + @param entrada
  + La neurona de entrada que estamos procesando
  + @param error
  + El error entre la salida real y
  + salida anticipada.
  + @return La cantidad para ajustar el peso.

\*/

entrenamiento doble protegidoFunction(doblevelocidad final, doble entrada final,

error doble final) { tasa de retorno \* entrada \* error;

}

}

Este programa se formará para 100 iteraciones. Está diseñado para enseñar la red-

trabajar para reconocer tres patrones. Estos patrones se resumen de la siguiente manera:

|  |  |  |  |
| --- | --- | --- | --- |
| Para | 001 | Salida | 0 |
| Para | 011 | Salida | 0 |
| Para | 101 | Salida | 0 |
| Para | 111 | Salida | 1 |

Para cada época, se le mostrarán los resultados reales y anticipados. Para la época 100, la red será entrenada. La salida de la época 100 se muestra aquí:

Comenzando epoch #100\*\*\*

Presentado [0.0,0.0,1.0] anticipado=0.0 actu- al=-0.333333131711973 error=0.3333333131711973

Presentado [0.0,1.0,1.0] anticipado=0.0 real=0.333333333558949 error=-0.33333333558949

Presentado [1.0,0.0,1.0] anticipado=0.0 real=0.33333333370649876 erro=-0.3333333370649876

Presentado [1.0,1.0,1.0] anticipado=1.0 real=0.6666666655103011 error=0.3333333448969893

Como se puede ver en la pantalla anterior, sólo hay dos salidas posibles 0.333 y 0.666. La salida de 0,333 corresponde a 0 y la salida de 0,666 corresponde a

1. Una red neuronal nunca producirá la salida exacta deseada, pero a través del redondeo se acerca bastante. Si bien la regla delta es eficiente en el ajuste de pesos, no es la más utilizada.

En el siguiente capítulo examinaremos la red de backpropagation feedforward, que es una de las redes neuronales más utilizadas. Backpropagation es una forma más avanzada de la regla delta.

# Resumen del capítulo

La tasa de error de una red neuronal es una estadística muy importante, que se utiliza como parte del proceso de entrenamiento. En este capítulo se muestra cómo calcular el ror de salida er para un elemento de conjunto de entrenamiento individual, así como cómo calcular el error RMS para todo el conjunto de entrenamiento.

El entrenamiento se produce cuando se modifican los pesos de la sinapsis para producir una salida más adecuada. El entrenamiento no supervisado se produce cuando la red neuronal se deja para desactivar las respuestas correctas. El entrenamiento supervisado se produce cuando la red neuronal se proporcionad con datos de entrenamiento y salidasanticipadas. La regla de Hebb se puede usar para el entrenamiento sin supervisión. La regla delta se utiliza para el entrenamiento supervisado.

En este capítulo aprendimos cómo una máquina aprende a través de la modificación de los pesos asociados con las conexiones entre las neuronas. Este capítulo introdujo los conceptos básicos de cómo aprende una máquina. La backpropagation es una forma más avanzada de la regla delta, que se introdujo en este capítulo. En el siguiente capítulo ex- plore backpropagation y ver cómo la clase de red neuronal lo implementa.

# Vocabulario

Clasificación Delta Rule Epoch

Iteración de reglas de Hebb

Tasa de aprendizaje de la red neuronal de Kohonen

Mapa autoorganizado (SOM) de Root Mean Square (RMS) entrenamiento supervisado sin supervisión

# Preguntas para revisión

* 1. Explicar la diferencia entre el entrenamiento supervisado y el entrenamiento no supervisado.
  2. ¿Cuál es la principal diferencia entre la regla delta y la regla de Hebb?
  3. Considere los siguientes cuatro resultados; calcular el error RMS.

Conjunto de entrenamiento #1, Esperado = 5, Real = 5

Conjunto de entrenamiento #2, Esperado = 2, Real = 3

Conjunto de entrenamiento #3, Esperado = 6, Real = 5

Conjunto de entrenamiento #4, Esperado = 8, Real = 4

Conjunto de entrenamiento #5, Esperado = 1, Real = 2

* 1. Usa la regla de Hebb para calcular el ajuste de peso.

Dos neuronas: N1 y N2 N1 a N2 Peso: 3

Activación N1: 2

Activación N2: 6

Calcule el delta del peso.

* 1. Utilice la regla delta para calcular el ajuste de peso.

Dos neuronas: N1 y N2 N1 a N2 Peso: 3

Activación N1: 2

Activación N2: 6, Se espera: 5

Calcule el delta del peso.