**CHAPTER** **5:** **FEEDFORWARD** **NEURAL** **NETWORKS**

* Presentación de la red neuronal de retropropagación feedforward
* Comprensión del algoritmo feedforward
* Implementación del algoritmo feedforward
* Comprensión del algoritmo de backpropagation
* Implementación del algoritmo de backpropagation

En este capítulo examinaremos una de las arquitecturas de redes neuronales más comunes, la red neuronal feedforword backpropagation. Esta arquitectura de red neuronal es muy popular, ya que se puede aplicar a muchas tareas diferentes. Para entender esta arquitectura de red neuronal, debemos examinar cómo se entrena y cómo procesa un patrón.

El primer término, "feedforward" describe cómo esta red neuronal procesa y vuelve a llamar a los patrones. En una red neuronal feedforward, las neuronas solo están conectadas. Cada capa de la red neuronal contiene conexiones a la siguiente capa (por ejemplo, desde la entrada hasta la capa oculta), pero no hay conexiones de vuelta. Esto difiere de la red neuronal hopfield que se examinó en el capítulo 3. La red neuronal hopfield estaba completamente conectada, y sus conexiones son tanto hacia adelante como hacia atrás. Exactamente cómo una red neuronal feedforward recuerda un patrón se explorará más adelante en este capítulo.

El término "backpropagation" describe cómo se entrena este tipo de red neuronal. La backpropagation es una forma de capacitación supervisada. Cuando se utiliza un método de entrenamiento supervisado, la red debe proporcionarse con entradas de muestra y salidas anticipadas. Las salidas previstas se comparan con las salidas reales para la entrada dada. Utilizando las salidas previstas, el algoritmo de entrenamiento de backpropagation entonces toma un error calculado y ajusta las ponderaciones de las diversas capas hacia atrás desde la capa de salida a la capa de entrada. El proceso exacto por el cual se produce la backpropagation también se discutirá más adelante en este capítulo.

Los algoritmos backpropagation y feedforward se utilizan a menudo juntos; sin embargo, esto no es de ninguna manera un requisito. Sería bastante permisible crear una red neuronal que utilice el algoritmo feedforward para determinar su salida y no utilice el algoritmo de entrenamiento de backpropagation. Del mismo modo, si decide crear un trabajo de red neuronal que utiliza métodos de entrenamiento de backpropagation, no se limita necesariamente a un algoritmo feedforward para determinar la salida de la red neuronal. Aunque estos casos son menos comunes que la red neuronal de backpropagation feedforward, se pueden encontrar ejemplos. En este libro, examinaremos sólo el caso en el que los algoritmos feedforward y backpropagation se utilizan juntos. Comenzaremos esta discusión examinando cómo funciona una red neuronal feedforward.

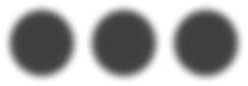
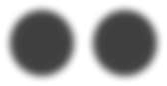
# Una red neuronal feedforward

Una red neuronal feedforward es similar a los tipos de redes neuronales que ya hemos examinado. Al igual que muchos otros tipos de redes neuronales, la red neuronal feedforward comienza con una capa de entrada. La capa de entrada puede estar conectada a una capa de guaria oculta o directamente a la capa de salida. Si está conectada a una capa oculta, la capa oculta se puede conectar a otra capa oculta o directamente a la capa de salida. Puede haber cualquier número de capas ocultas, siempre y cuando haya al menos una capa oculta o capa de salida proporcionada. En uso común, la mayoría de las redes neuronales tendrán una capa oculta, y es muy raro que una red neuronal tenga más de dos capas ocultas.

## La estructura de una red neuronal feedforward

La Figura 5.1 ilustra una red neuronal típica feedforward con una sola capa oculta.

### Figura 5.1: Una red neuronal de avance típica (una sola capa oculta).



entrada

1

entrada

2

escondido

1

escondido

2

escondido

3

salida

1

Las redes neuronales con más de dos capas ocultas son poco comunes.

## Elección de su estructura de red

Como vimos en la sección anterior, hay muchas maneras en que se pueden construir redes neuronales feedforward. Usted debe decidir cuántas neuronas estarán dentro de las capas de entrada y salida. También debe decidir cuántas capas ocultas va a tener y cuántas neuronas habrá en cada una de ellas.

Hay muchas técnicas para elegir estos parámetros. En esta sección cubriremos algunas de las "reglas generales" que puede utilizar para ayudarle en estas decisiones; sin embargo, estas reglas sólo le llevarán hasta cierto punto. En casi todos los casos, se requerirán algo de experiencia determinar la estructura óptima para su red neuronal feedforward. Hay muchos libros dedicados enteramente a este tema. Para una discusión exhaustiva sobre la estructuración de las redes neuronales feedforward, debe consultar el libro Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks (MIT Press, 1999).

## La capa de entrada

La capa de entrada es el conducto a través del cual el entorno externo presenta un patrón a la red neuronal. Una vez que se presenta un patrón a la capa de entrada, la capa de salida producirá otro patrón. En esencia, esto es todo lo que hace la red neuronal. La capa de entrada debe representar la condición para la que estamos entrenando la red neuronal. Cada neurona de entrada debe representar alguna variable independiente que tenga una influencia sobre la salida de la red neuronal.

Es importante recordar que las entradas a la red neuronal son números de punto flotante. Estos valores se expresan como el tipo de datos Java primitivo "double". Esto no quiere decir que solo se pueden procesar datos numéricos con la red neuronal; si desea procesar una forma de datos que no es numérica, debe desarrollar un proceso que normalice estos datos a una representación numérica. En el capítulo 12, "OCR and the Self-Organizing Map", te mostraré cómo comunicar información gráfica a una red neuronal.

## La capa de salida

La capa de salida de la red neuronal es lo que realmente presenta un patrón al entorno externo. El patrón presentado por la capa de salida se puede rastrear directamente hasta la capa de entrada. El número de neuronas de salida debe estar directamente relacionado con el tipo de trabajo que la red neuronal debe realizar.

Para determinar el número de neuronas que se usarán en la capa de salida, primero debe tener en cuenta el uso previsto de la red neuronal. Si la red neuronal se va a utilizar para clasificar elementos en grupos, entonces a menudo es preferible tener una neurona de salida para cada grupo en el que se van a asignar los elementos de entrada. Si la red neuronal va a realizar la reducción de ruido en una señal, entonces es probable que el número de neuronas de entrada coincida con el número de neuronas de salida. En este tipo de red neuronal, querrá que los patrones salgan de la red neuronal en el mismo formato que entraron.

Para un ejemplo específico de cómo elegir el número de neuronas de entrada y el número de las neuronas de salida, considere un programa que se utiliza para el reconocimiento óptico de caracteres (OCR), como el programa presentado en el ejemplo en el capítulo 12, "OCR y el mapa autoorganizable." Para determinar el número de neuronas utilizadas para el OCR ex-amplio, primero consideraremos la capa de entrada. El número de neuronas de entrada que usaremos es el número de píxeles que podrían representar cualquier carácter dado. Los caracteres procesados por este programa se normalizan a un tamaño universal representado por una cuadrícula 5x7. Una cuadrícula 5x7 contiene un total de 35 píxeles. Por lo tanto, el programa OCR tiene 35 neuronas de entrada.

El número de neuronas de salida utilizadas por el programa OCR variará dependiendo de cuántos caracteres el programa ha sido entrenado para reconocer. El archivo de formación predeterminado que se proporciona con el programa OCR se utiliza para entrenarlo para reconocer 26 caracteres. Usando este archivo, la red neuronal tendrá 26 neuronas de salida. Presentar un patrón a las neuronas de entrada disparará la neurona de salida adecuada que corresponde a la letra que representa el patrón de entrada.

# Solución del problema XOR

A continuación, examinaremos una red neuronal simple que aprenderá el operador XOR. El operador XOR fue cubierto en el capítulo 1. Verá cómo utilizar varias clases de esta red neuronal. Estas clases se imparten en la descarga complementaria con la compra de este libro. El Apéndice A describe cómo obtener esta descarga. Más adelante en el capítulo, se le mostrará cómo se construyeron estas clases.

### Listado 5.1: El problema XOR (XOR.java)

paquete com.heatonresearch.book.introneuralnet.ch5.xor;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

FeedforwardLayer;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

FeedforwardNetwork;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

tren. Tren;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

train.backpropagation. Backpropagation;

/\*\*

* Capítulo 5: La red neuronal de retropropagación feedforward

\*

* XOR: Aprende el patrón XOR con una red neuronal feedforward
* que utiliza backpropagation.

\*

* @author Jeff Heaton
* @version 2,1

\*/

clase pública XOR {

doble estática pública XOR\_INPUT[][] = { { 0.0, 0.0 },

{ 1.0, 0.0 },

{ 0.0, 1.0 }, { 1.0, 1.0 } };

doble estática pública XOR\_IDEAL[][] = { { 0.0 }, { 1.0 },

{ 1.0 }, { 0.0 } };

vacío estático público main(final String args[]) { final FeedforwardNetwork network =

nuevo FeedforwardNetwork(); network.addLayer(nuevo FeedforwardLayer(2)); network.addLayer(nuevo FeedforwardLayer(3)); network.addLayer(nuevo FeedforwardLayer(1)); network.reset();

entrenar la red neuronal

tren final = nueva backpropagation(red, XOR\_INPUT, XOR\_IDEAL,

0.7, 0.9);

int epoch = 1;

hacer {

train.iteration();

System.out

. println("Epoch #" + epoch

+ " Error:" + train.getError()); epoch++;

} mientras que ((epoch < 5000) && (train.getError() > 0.001));

probar la red neuronal System.out.println("Resultados de la red neuronal:"); para (int i = 0; i < XOR\_IDEAL.length; i++) {

final doble real[] = red.

computeOutputs(XOR\_INPUT[i]);

System.out.println(XOR\_INPUT[i][0] + ","

+ XOR\_INPUT[i][1]

+ ", real=" + real[0] + ",ideal="

+ XOR\_IDEAL[i][0]);

}

}

}

Las últimas líneas de salida de este programa se muestran aquí.

Epoch #4997 Error:0.006073963240271441 Epoch #4998 Er- ror:0.006073315333403568 Epoch #4999 Error:0.006072676304029325

Resultados de la red neuronal: 0.0,0.0, real=0. 0025486129720869773,ide al=0.0 1.0,0.0, actual=0.9929280525379659,ideal=1.0 0.0,1.0, actu

al=0. 9944310234678858,ideal=1.0 1.0,1.0, real=0.007745179145434

604,ideal=0.0

Como se puede ver, la red atravesó casi 5.000 épocas de entrenamiento. Esto produce un error de poco más de medio por ciento y tomó sólo unos segundos. A continuación, se mostraron los resultados. La red neuronal produjo un número cercano a cero para la entrada de 0,0 y 1,1. También produjo un número cercano a 1 para las entradas de 1,0 y 0,1.

Este programa es muy fácil de construir. En primer lugar, se crea una matriz **doble** bidimensional que contiene la entrada para la red neuronal. Estos son los conjuntos de entrenamiento para la red neuronal.

doble XOR\_ público[ ][] = {

|  |  |  |  |
| --- | --- | --- | --- |
| { | 0.0, | 0.0 | }, |
| { | 1.0, | 0.0 | }, |
| { | 0.0, | 1.0 | }, |
| { | 1.0, | 1.0 | } }; |

A continuación, se crea una matriz **doble** bidimensional que contiene la salida ideal para cada uno de los conjuntos de entrenamiento anteriores.

doble estática pública XOR\_IDEAL[][] = {

{ 0.0 },

{ 1.0 },

{ 1.0 },

{ 0.0 } };

Puede parecer que una **matriz doble** unidimensional sería suficiente para esta tarea. Sin embargo, las redes neuronales pueden tener más de una neurona de salida, lo que produciría más de un **doble** valor. Esta red neuronal tiene sólo una neurona de salida.

Ahora se crea un objeto **FeedforwardNetwork.** Este es el objeto principal de la red neuronal. Las capas se agregarán a este objeto.

red final feedforwardNetwork = nueva feedforwardNetwork();

La primera capa que se agregará será la capa de entrada. Se crea un objeto **FeedforwardLaye**. El valor de dos especifica que habrá dos neuronas de entrada.

network.addLayer(nuevo FeedforwardLayer(2));

La segunda capa a añadir será una capa oculta. Si no se agregan capas adicionales más allá de esta capa, entonces será la capa de salida. La primera capa agregada es al-ways la capa de entrada, la última capa agregada es siempre la capa de salida. Las capas añadidas entre esas dos capas son las capas ocultas. Se crea un objeto **FeedforwardLayer** para que sirva como capa oculta. El valor de tres especifica que habrá tres neuronas en la capa oculta.

network.addLayer(nuevo FeedforwardLayer(3));

La capa final a añadir será la capa de salida. Un **FeedforwardLayer**

objeto se crea. El valor de uno especifica que habrá una sola neurona de salida.

network.addLayer(nuevo FeedforwardLayer(1));

Por último, se restablece la red neuronal. Esto aleatoriza los valores de peso y umbral. Esta red neuronal aleatoria ahora tendrá que ser entrenada.

network.reset();

El método de backpropagation se utilizará para entrenar la red neuronal. Para ello, se crea un objeto **Backpropagation.**

tren final = nueva backpropagation(red, XOR\_INPUT, XOR\_IDEAL, 0,7, 0,9);

El objeto de entrenamiento requiere que se pasen varios argumentos a su constructor. El primer argumento es la red a entrenar. El segundo argumento es la **XOR\_INPUT** y las variables XOR\_IDEAL, que proporcionan los conjuntos de entrenamiento y los resultados esperados. Por último, se especifica la tasa de aprendizaje y el impulso.

La tasa de aprendizaje especifica la rapidez con la que aprenderá la red neuronal. Esto suele ser un valor alrededor de uno, ya que es un porcentaje. El impulso especifica cuánto efecto tendrá la iteración de entrenamiento anterior en la iteración actual. El impulso también es un porcentaje, y por lo general es un valor cercano a uno. Para no utilizar ningún impulso en el algoritmo de backpropagation, especificará un valor de cero. La tasa de aprendizaje y los valores de impulso se discutirán más adelante en este capítulo.

Ahora que el objeto de entrenamiento está configurado, el programa recorrerá las borrados de entrenamiento hasta que la tasa de error sea pequeña, o realice 5.000 épocas, o iteraciones.

int epoch = 1; hacer {

train.iteration();

System.out . println("Epoch #"

+ época

+ " Error:"

+ train.getError()); epoch++;

} mientras que ((epoch < 5000) && (train.getError() > 0.001));

Para realizar una iteración de entrenamiento, simplemente llame al método **de iteración** en el objeto train-ing. El bucle continuará hasta que el error sea menor a una décima parte del porcentaje, o el programa haya realizado 5.000 iteraciones de entrenamiento.

Por último, el programa mostrará los resultados producidos por la red neuronal.

System.out.println ("Resultados de la red neuronal:" );

para (int i = 0; i < XOR\_IDEAL.length; i++) {

A medida que el programa recorre cada uno de los conjuntos de entrenamiento, ese conjunto de entrenamiento se presenta a la red neuronal. Para presentar un patrón a la red neuronal, se utiliza el método **computeOutputs.** Este método acepta una **doble** matriz de valores de entrada. Esta matriz debe tener el mismo tamaño que el número de neuronas de entrada o un concepto- se lanzará. Este método devuelve una matriz de valores **dobles** del mismo tamaño que el número de neuronas de salida.

final doble real[] = network.computeOutputs(XOR\_INPUT[i]);

Se muestra la salida de la red neuronal.

System.out.println(XOR\_INPUT[i][0] + ","

+ XOR\_INPUT[i][1]

+ ", real="

+ real[0] + ",ideal=" + XOR\_IDEAL[i][0]);

}

}

Esta es una red neuronal muy simple. Utilizó el funcde activación sigmoide por defecto -

ción. Como verá en la siguiente sección, se pueden especificar otras funciones de activación.

# Funciones de activación

La mayoría de las redes neuronales pasan la salida de sus capas a través de funciones de activación. Estas funciones de activación escalan la salida de la red neuronal en rangos adecuados. El programa de red neuronal en la última sección utilizó la función de activación sigmoide. La función de activación sigmoid es la opción predeterminada para la clase **FeedforwardLayer.** Es posible utilizar otros. Por ejemplo, para utilizar la función de activación tangente hiperbólica, se usarían las siguientes líneas de código para crear las capas.

network.addLayer(nuevo FeedforwardLayer(nueva ActivationTANH(),2));

network.addLayer(nuevo FeedforwardLayer(nueva ActivationTANH(),3)); network.addLayer(nuevo FeedforwardLayer(nueva ActivationTANH(),1));

Como se puede ver en el código anterior, una nueva instancia de **ActivationTANH** es creada y pasado a cada capa de la red. Esto especifica que se debe utilizar la tangente hiperbólica, en lugar de la función sigmoide.

Usted puede notar que sería posible utilizar una función de activación diferente para cada capa de la red neuronal. Aunque técnicamente no hay nada que te impida hacer esto, tal práctica sería inusual.

Se proporcionan un total de tres funciones de activación:

* + Tangente hiperbólica
  + sigmoide
  + lineal

También es posible crear su propia función de activación. Hay una interfaz denominada **ActivationFunction**. Anyclassthatimplementsla interfaz **ActivationFunction** puede servir como función de activación. Las tres funciones de activación proporcionadas se discutirán en las secciones siguientes.

## Uso de una función de activación sigmoide

Una función de activación sigmoide utiliza la función sigmoide para determinar su activación- ción. La función sigmoide se define de la siguiente manera:

### Ecuación 5.1: La función sigmoide

*ḥ* ( *x* )= 1

1+*e*−*x*

El término sigmoide significa curvado en dos direcciones, como la letra "S". Puede ver la función sigmoide en la Figura 5.2.

### Figura 5.2: La función Sigmoid.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  | -4 |  |  |  | -3 |  |  |  | -2 |  |  |  | -1 |  |  |  | 0 |  |  |  | 1 |  |  |  | 2 |  |  |  | 3 |  |  |  | 4 |  |  |  | 5 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Una cosa importante a tener en cuenta acerca de la función de activación sigmoid es que sólo vuelve a girar valores positivos. Si necesita que la red neuronal devuelva números negativos, la función sigmoide no será adecuada. La función de activación sigmoid se implementa en la clase **ActivationFunctionSigmoid.** Esta clase se muestra en el listado 5.2.

### Listado 5.2: La clase de función de activación sigmoide (ActivationSigmoid.java)

paquete com.heatonresearch.book.introneuralnet.neural.activation;

/\*\*

* ActivationSigmoid: La función de activación sigmoide asume un
* forma sigmoidal. Solo se generan números positivos. No
* utilizar esta función de activación si la salida de números negativos es
* deseado.

\*

* @author Jeff Heaton
* @version 2,1

\*/

la clase pública ActivationSigmoid implementa ActivationFunction {

/\*\*

\* Id. de serie para esta clase.

\*/

serie larga final estática privadaVersionUID = 5622349801036468572L;

/\*\*

* Una función de umbral para una red neuronal.
* @param La entrada a la función.
* @return La salida de la función.

\*/

activación doble públicaFunction( doble final d) { retorno 1.0 / (1 + Math.exp(-1.0 \* d));

}

/\*\*

* Algunos métodos de entrenamiento requieren la derivada.
* @param la entrada.
* @return La salida.

\*/

doble derivada públicaFunción(doble d) { retorno d\*(1.0-d);

}

}

Como puede ver, la función sigmoide se define dentro de la **activaciónFunction**

método. Este método fue definido por la interfaz **ActivationFunction.** Si desea crear su propia función de activación, es tan simple como crear una clase que implementa la interfaz **ActivationFunction** y proporcionar un método **activationFunction.**

La interfaz **ActivationFunction** también define un método denominado **derivativeFunction** que implementa la derivada de la función de activación principal. Ciertos métodos de entrenamiento requieren la derivada de la función de activación. **La backpropagation** es uno de estos métodos. **La backpropagation** no se puede utilizar en una red neuronal que utiliza una función de activación que no tiene un derivado. Sin embargo, todavía se podría utilizar un algoritmo genético o recocido simulado. Estas dos técnicas se tratarán en los próximos dos capítulos.

## Uso de una función de activación tangente hiperbólica

Como se mencionó anteriormente, la función de activación sigmoid no devuelve valores inferiores a cero. Sin embargo, es posible "mover" la función sigmoide a una región del gráfico para que proporcione números negativos. Esto se hace utilizando la función hiperbólica tang- gent. La ecuación para la función de activación hiperbólica se muestra en la Ecuación 5.2.

### Ecuación 5.2: La función TANH

*e*2x 1

−

*ḥ* ( *x* )=

*e*2x+1

Aunque esto se ve considerablemente más complejo que la función sigmoide, se puede pensar con seguridad en ella como una versión positiva y negativa compatible de la función sigmoide. El gráfico de la función tangente hiperbólica se proporciona en la Figura 5.3.

### Figura 5.3: La función tangente hiperbólica.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  | -4 |  |  |  | -3 |  |  |  | -2 |  |  |  | -1 |  |  |  | 0 |  |  |  | 1 |  |  |  | 2 |  |  |  | 3 |  |  |  | 4 |  |  |  | 5 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Una cosa importante a tener en cuenta acerca de la función de activación tangente hiperbólica es que devuelve valores positivos y negativos. Si necesita que la red neuronal devuelva números negativos, esta es la función de activación que se va a utilizar. La función de activación tangente hiperbólica se implementa en la clase **ActivationFunctionTanH.** Esta clase se muestra en el listado 5.3.

### Listado 5.3: La clase de función tangente hiperbólica (ActivationTANH.java)

paquete com.heatonresearch.book.introneuralnet.neural.activation;

/\*\*

* ActivationTANH: La función de activación tangente hiperbólica
* toma la forma curva de la tangente hiperbólica. Esto
* función de activación produce tanto positivo como negativo
* salida. Utilice esta función de activación si
* se desea una salida positiva.

\*

* @author Jeff Heaton
* @version 2,1

\*/

Activación de clase públicaTANH implementa ActivationFunction {

/\*\*

\* Id. de serie para esta clase.

\*/

serie larga final estática privadaVersionUID = 9121998892720207643L;

/\*\*

* Una función de umbral para una red neuronal.
* @param La entrada a la función.
* @return La salida de la función.

\*/

activación doble públicaFunction(doble d) {

resultado doble final = (Math.exp(d\*2.0)-1.0)/ (Math.exp(d\*2. 0)+1.0);

resultado de retorno;

}

/\*\*

* Algunos métodos de entrenamiento requieren la derivada.
* @param la entrada.
* @return La salida.

\*/

doble derivada públicaFunction(doble d) {

retorno( 1.0-Math.pow(activationFunction(d), 2.0) );

}

}

Como puede ver, la función tangente hiperbólica se define dentro de la

activationFunction método. Este método fue definido por la interfaz **ActivationFunction.** El **derivadoFunction** también se define para devolver el resultado del derivado de la función tangente hiperbólica.

## Uso de una función de activación lineal

La función de activación lineal no es esencialmente ninguna función de activación en absoluto. Es prob- ably el menos utilizado de las funciones de activación. El funcde activación lineal- ción no modifica un patrón antes de generarlo. La función para la capa lineal se da en la ecuación 5.3.

### Ecuación 5.3: Una función lineal

*ḥ* ( *x* )= *x*

La función de activación lineal puede ser útil en situaciones en las que necesita el rango de números de los neumáticos para ser salida. Por lo general, querrás pensar en tus neuronas como activas o no activas. Dado que las funciones de activación de tangente hiperbólica y sigmoide han establecido límites superiores e inferiores, tienden a utilizarse más para operaciones de tipo booleano (activado o desactivado). La función de activación lineal es útil para presentar un rango. Se proporciona un gráfico de la función de activación lineal en la Figura 5.4.

### Figura 5.4: La función de activación lineal.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  | -4 |  |  |  | -3 |  |  |  | -2 |  |  |  | -1 |  |  |  | 0 |  |  |  | 1 |  |  |  | 2 |  |  |  | 3 |  |  |  | 4 |  |  |  | 5 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | -2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

La implementación de la función de activación lineal es bastante simple. Se muestra en el Listado 5.4.

### Listado 5.4: La función de activación lineal (ActivationLinear.java)

paquete com.heatonresearch.book.introneuralnet.neural.activation;

importación com.heatonresearch.book.introneuralnet.neural.exception.

NeuralNetworkError;

/\*

* ActivationLinear: La capa Lineal realmente no es un
* función de activación en absoluto. La entrada simplemente se transmite,
* sin modificar, a la salida. Esta función de activación es
* principalmente teórico y de poco uso real. Por lo general, un
* función de activación que escala entre 0 y 1 o
* -1 y 1 deben utilizarse.

\*

* @author Jeff Heaton
* @version 2,1

\*/

activationlinear de clase pública implementa ActivationFunction {

/\*\*

\* Id. de serie para esta clase.

\*/

serie larga final estática privadaVersionUID =

-5356580554235104944L;

/\*\*

* Una función de umbral para una red neuronal.
* @param La entrada a la función.
* @return La salida de la función.

\*/

activación doble públicaFunction( doble final d) { devolver d;

}

/\*\*

* Algunos métodos de entrenamiento requieren la derivada.
* @param la entrada.
* @return La salida.

\*/

doble derivada públicaFunction(doble d) {

lanzar nuevo NeuralNetworkError("No se puede utilizar el lineal"

+" función de activación donde se requiere un derivado." );

}

}

Como puede ver, la función de activación lineal no hace más que devolver lo que es

dado. El derivado de la función lineal es 1. Esto no es útil para el entrenamiento, por lo tanto, la **derivativeFunction** para la función de activación lineal produce una excepción. No puede utilizar la backpropagation para entrenar una red neuronal que haga uso de la función lineal.

# El número de capas ocultas

Realmente hay dos decisiones que se deben tomar con respecto a las capas ocultas: cuántas capas ocultas realmente tener en la red neuronal y cuántas neuronas habrá en cada una de estas capas. Primero examinaremos cómo determinar el número de capas ocultas para usar con la red neuronal.

Los problemas que requieren dos capas ocultas rara vez se encuentran. Sin embargo, las redes neuronales con dos capas ocultas pueden representar funciones con cualquier tipo de forma. Actualmente no hay ninguna razón teórica para utilizar redes neuronales con más de dos capas ocultas. De hecho, para muchos problemas prácticos, no hay razón para usar más de una capa oculta. La Tabla 5.1 resume las capacidades de las arquitecturas de red neuronal con varias capas ocultas.

### Tabla 5.1: Determinación del número de capas ocultas

|  |  |
| --- | --- |
| **Número de** **capas ocultas** | **resultado** |
| ninguno | Sólo capaz de representar funciones o decisiones separables lineales. |
| 1 | Puede aproximar cualquier función que contenga una asignación continua de un espacio finito a otro. |
| 2 | Puede representar un límite de decisión arbitrario a la ac-curación arbitraria con funciones de activación racionales y puede aproximar cualquier asignación suave a cualquier precisión. |

Decidir el número de capas de neuronas ocultas es sólo una pequeña parte del problema. También debe determinar cuántas neuronas estarán en cada una de estas capas ocultas. Este proceso se cubre en la siguiente sección.

**El número** **de** **neuronas** **en** **las** capas **ocultas**

Decidir el número de neuronas en las capas ocultas es una parte muy importante para decidir su arquitectura general de la red neuronal. Aunque estas capas no interactúan directamente con el entorno externo, tienen una enorme influencia en la salida final. Tanto el número de capas ocultas como el número de neuronas en cada una de estas capas ocultas deben considerarse cuidadosamente.

El uso de muy pocas neuronas en las capas ocultas dará lugar a algo llamado sub-ajuste. El ajuste deficiente ocurre cuando hay muy pocas neuronas en las capas ocultas para detectar adecuadamente las señales en un conjunto de datos complicado.

El uso de demasiadas neuronas en las capas ocultas puede resultar en varios problemas. En primer lugar, demasiadas neuronas en las capas ocultas pueden resultar en un exceso de ajuste. El sobreajuste ocurre cuando la red neuronal tiene tanta capacidad de procesamiento de información que la cantidad limitada de información contenida en el conjunto de entrenamiento no es suficiente para entrenar todas las neuronas en las capas ocultas. Un segundo problema puede ocurrir incluso cuando los datos de entrenamiento son suficientes. Un número excesivamente grande de neuronas en las capas ocultas puede in-

reducir el tiempo que se tarda en entrenar la red. La cantidad de tiempo de entrenamiento puede aumentar hasta el punto de que es imposible entrenar adecuadamente la red neuronal. Obviamente, se debe llegar a algún compromiso entre demasiadas y muy pocas neuronas en las capas de guarida oculta.

Existen muchos métodos de regla general para determinar el número correcto de neu-rons que se utilizarán en las capas ocultas, como las siguientes:

* + El número de neuronas ocultas debe estar entre el tamaño de la capa de entrada y el tamaño de la capa de salida.
  + El número de neuronas ocultas debe ser 2/3 del tamaño de la capa de entrada, más el tamaño de la capa de salida.
  + El número de neuronas ocultas debe ser menos del doble del tamaño de la capa de entrada.

Estas tres reglas proporcionan un punto de partida para que usted pueda considerar. En última instancia, el selec- ción de una arquitectura para su red neuronal se reducirá a prueba y error. Pero, ¿qué se entiende exactamente por prueba y error? Usted no quiere comenzar a lanzar números aleatorios de capas y neuronas en su red. Para ello sería muy tiempo consumida- ing. Capítulo 8, "Poda de una red neuronal" explorará varias maneras de determinar una estructura óptima para una red neuronal.

# Examen del proceso feedforward

Más temprano en este capítulo vio cómo presentar datos a una red neuronal y entrenar esa red neuronal. El resto de este capítulo se centrará en cómo se realizaron estas operaciones. Las clases de red neuronal presentadas anteriormente no son demasiado complejas. Comenzaremos explorando cómo calculan la salida de la red neuronal. Esto se denomina proceso feedforward.

## Cálculo de la salida matemáticamente

La ecuación 5.4 describe cómo se puede calcular la salida de una sola neurona.

### Ecuación 5.4: Cálculos de avance

*n* −1

*salida*=(∑ *xi* *wi*)+*wn*

*i*=0

La ecuación anterior toma los valores de entradadenominados **x**y los multiplica por el peso **w**. Como recordará del capítulo 2, el último valor de la matriz de peso es el umbral. Este umbral está **wn**.

Para realizar la operación anterior con matemáticas de matriz, la entrada se utiliza para rellenar una matriz y se agrega una fila, los elementos de los cuales son todos. Este valor se multiplicará con el valor umbral. Por ejemplo, si la entrada fuera 0,1, 0,2 y 0,3, se produciría la siguiente matriz de entrada.

**Ecuación 5.5:** **Una** matriz **de** **entrada**

[0.1 0.2 0.3 1]

A continuación, el producto punto se tomaría entre la matriz de entrada y el peso ma- trix. Este número se alimentaría entonces a la función de activación para producir la salida de la neurona.

## Cálculo de la salida para una red neuronal feedforward

Para obtener la salida de la red neuronal, se debe llamar a la función **computeOutputs.** Esta función llamará a las capas de la red neuronal y determinará la salida. La firma de la función **computeOutputs** se muestra aquí:

public double[] computeOutputs (entrada doble final[])

Lo primero que hace la función **computeOutputs** es comprobar y ver si la matriz de **entrada** tiene el tamaño correcto. El tamaño de la matriz de **entrada** debe coincidir con el número de neuronas de entrada en la capa de entrada.

if (input.length != this.inputLayer.getNeuronCount()) { throw new NeuralNetworkError(

"Discrepancia de tamaño: No se pueden calcular las salidas para el tamaño de entrada="

+ input.length

+ " para el tamaño de la capa de entrada="

+ this.inputLayer.getNeuronCount());

}

A continuación, cada uno de los **feedforwardLayer** objetos se recorrerán en bucle.

para (capa final feedforwardLayer : this.layers) {

La capa de entrada recibirá la entrada que se proporcionó.

if (layer.isInput()) { layer.computeOutputs(entrada);

Cada una de las capas ocultas recibirá la salida de la capa anterior como en- put.

} si (layer.isHidden()) { layer.computeOutputs(null);

}

}

Por último, se devuelve el resultado de la capa de salida.

return (this.outputLayer.getFire());

El método **computeOutputs** para la red neuronal obtiene su salida calculando la salida de cada capa. En la siguiente sección se explicará cómo se calcula la salida de una capa.

## Cálculo de la salida para una capa

La clase **FeedforwardLayer** también contiene un método **computeOutputs.** Este método se utiliza para calcular la salida de cada una de las capas. La firma para este método se muestra aquí:

public double[] computeOutputs (patrón doble final[])

En primer lugar, el método **computeOutputs** comprueba si se le presentó o no un patrón.

if (patrón ! = null) {

Si se presentó un **patrón,** el patrón debe copiarse en la variable **fire** in-stance.

para (int i = 0; i < getNeuronCount(); i++) { setFire(i, patrón[i]);

}

}

Si no se presentó un **patrón,** la variable de instancia de **incendio** tendrá al-

listo para ser establecido por la capa anterior.

entrada final matrixMatrix = createInputMatrix(this.fire);

A continuación, se debe calcular un valor de salida para convertirse en el valor de entrada para cada una de las neuronas de la siguiente capa.

para (i = 0; i < this.next.getNeuronCount(); i++) {

Para ello, obtenemos una matriz de columnas para cada una de las neuronas de la siguiente capa.

final Matrix col = this.matrix.getCol(i);

A continuación, se determina el producto de punto entre la columna de matriz de peso y la entrada.

Este será el valor que se presentará a la siguiente capa.

suma doble final = MatrixMath.dotProduct(col, inputMatrix);

La salida se pasa a la **activationFunction** y, a continuación, se almacena en la variable fire instance de la siguiente capa.

this.next.setFire(i,

this.activationFunction.activationFunction(suma));

}

La variable **fire** instance se devuelve como salida para esta capa.

devolver this.fire;

Este proceso continúa con cada capa. La salida de la capa de salida es la salida de la red neuronal.

# Examen del proceso de backpropagation

Ahora ha visto cómo calcular la salida para una red neuronal feedforward. Ha visto tanto las ecuaciones matemáticas como la implementación de Java. Al examinar cómo calcular los valores finales de la red, usamos los pesos de conexión y los valores de umbral para determinar el resultado final. Es posible que se pregunte cómo se determinaron estos valores.

Los valores contenidos en la matriz de peso y umbral se determinaron mediante el algoritmo de backpropagation. Este es un algoritmo muy útil para entrenar redes neuronales. El algoritmo de backpropagation funciona ejecutando la red neuronal tal como lo hicimos en nuestro ejemplo de reconocimiento, como se muestra en la sección anterior. El principal diferencia en el algoritmo de backpropagation es que presentamos la red neuronal con datos de entrenamiento. A medida que cada elemento de datos de entrenamiento se presenta a la red neuronal, el er- ror se calcula entre la salida real de la red neuronal y la salida que se esperaba (y se especifica en el conjunto de entrenamiento). A continuación, se modifican los pesos y el umbral, por lo que existe una mayor probabilidad de que la red devuelva el resultado correcto cuando la red se presenta a continuación con la misma entrada.

La backpropagation es un método muy común para entrenar redes feedforward multicapa. La backpropagation se puede utilizar con cualquier red feedforward que utilice una función de activación que sea diferente. Es esta función derivada la que usaremos durante el entrenamiento. No es necesario que entienda el cálculo o cómo tomar la derivada de una ecuación para trabajar con el material en este capítulo. Si está utilizando una de las funciones de activación comunes, simplemente puede obtener la función de activación derivada de un gráfico.

Para entrenar la red neuronal, se debe determinar un método para calcular el error. A medida que se entrena la red neuronal, la red se presenta con muestras del conjunto de entrenamiento. El resultado obtenido de la red neuronal se compara con el resultado previsto que forma parte del conjunto de entrenamiento. El grado en que la salida de la red neuronal difiere de esta salida esperada es el error.

Para entrenar la red neuronal, debemos intentar minimizar este error. Para minimizar el error, se deben modificar los pesos y umbrales de conexión de las neuronas. Debemos definir una función que calcule la tasa de error de la red neuronal. Este error func- tion debe ser matemáticamente diferente. Porque la red utiliza una

función de activación, las activaciones de las neuronas de salida se pueden considerar como funciones diferentes- envidiables de la entrada, pesos, y umbrales. Si la función de error también es una función diferente, como la suma de la función de error cuadrado, la función de error en sí es una función diferente de estos pesos. Esto nos permite evaluar la des-rivative del error utilizando las ponderaciones. A continuación, utilizando estos derivados, encontramos pesos y umbrales que minimizarán la función de error.

Hay varias maneras de encontrar pesos que minimizarán la función de error. El enfoque más popular es utilizar el método de descenso de gradiente. El algoritmo que evalúa la derivada de la función de error se conoce como backpropagation, porque propaga los errores hacia atrás a través de la red.

## La interfaz del tren

Este libro cubrirá tres métodos de entrenamiento diferentes que se pueden utilizar para redes neuronales feedforward. Este capítulo presenta el método backpropagation. El capítulo 6 discutirá el uso de un algoritmo genético para el entrenamiento. El capítulo 7 discutirá el uso de recocido simulado para una red neuronal. Los tres métodos de entrenamiento implementan la interfaz Train. La interfaz **del tren** se muestra en el listado 5.5.

### Listado 5.5: La interfaz del tren (tren.java)

paquete com.heatonresearch.book.introneuralnet.neural.feedforward. tren;

importar com.heatonresearch.book.introneuralnet.neural.feedforward.

FeedforwardNetwork;

/\*\*

* Tren: Interfaz para todo el entrenamiento de la red neuronal feedforward
* métodos. Actualmente hay tres métodos de entrenamiento definidos:

\*

* Backpropagation
* Algoritmos genéticos
* Recocido simulado

\*

* @author Jeff Heaton
* @version 2,1

\*/

interfaz pública Tren {

/\*\*

* Obtenga el porcentaje de error actual del entrenamiento.
* @return El error actual.

\*/

público doble getError();

/\*\*

* Obtenga la mejor red actual de la capacitación.
* @return La mejor red.

\*/

público FeedforwardNetwork getNetwork();

/\*\*

* Realizar una iteración de entrenamiento.

\*/

iteración pública del vacío( );

}

Hay tres métodos diferentes que deben implementarse para usar el **Tren**

interfaz. El método **getError** devolverá el nivel de error actual, calculado mediante el cuadrado medio raíz (RMS) para la red neuronal actual. El método **getNetwork** devuelve una red neuronal entrenada que alcanza el nivel de error notificado por el método **getError.** Por último, se llama al método **de iteración** para realizar una iteración de entrenamiento. Por lo general, se llama repetidamente al método **de iteración** hasta que el método **getError** devuelve un error aceptable.

## Las clases Java de backpropagation

El algoritmo de backpropagation se implementa en dos clases. La clase **Backpropagation,** que implementa la interfaz **Train,** es la clase de entrenamiento principal utilizada. Internamente, la clase **Backpropagation** utiliza la clase **BackpropagationLayer** para contener información que la backpropagation al- gorithm necesita para cada capa de la red neuronal.

Los dos métodos principales a los que llamarán los métodos mediante la clase **Backpropagation** son las funciones **iteración** y **getError.** La función **de iteración** tiene la siguiente firma:

iteración del vacío público() {

La función de iteración comienza por recorrer en bucle todos los conjuntos de entrenamiento que se proporcionaron.

para (int j = 0; j < this.input.length; j++) {

Cada conjunto de entrenamiento se presenta a la red neuronal y las salidas son calcu- lated.

this.network.computeOutputs(this.input[j]);

Una vez calculadas las salidas, el entrenamiento puede comenzar. Este es un proceso de dos pasos. En primer lugar, el error se calcula comparando la salida con los valores ideales.

calcError(this.ideal[j]);

Una vez procesados todos los conjuntos, la red aprende de estos errores.

aprender();

Por último, el nuevo error global, es decir, el error en todos los conjuntos de entrenamiento, es calcu- lated.

this.error = this.network.calculateError(this.input, this.ideal);

Es este error el que se devolverá cuando se realiza una llamada a **getError**.

## Cálculo del error de backpropagation

El primer paso en el cálculo de errores de backpropagation es llamar al método **calcError** del objeto **BackPropagation.** La firma para el método **calcError** se muestra aquí:

calcError del vacío público(doble ideal final[])

En primer lugar, verificamos que el tamaño de la matriz **ideal** corresponde al número de neuronas de salida. Dado que la matriz **ideal** especifica los valores ideales para las neuronas de salida, debe coincidir con el tamaño de la capa de salida.

if (ideal.length != this.network.getOutputLayer(). getNeuronCount()) {

lanzar nuevo NeuralNetworkError(

"Discrepancia de tamaño: No se puede calcError para el tamaño de entrada ideal="

+ ideal.length + " para el tamaño de la capa de salida="

+ this.network.getOutputLayer(). getNeuronCount());

}

El objeto **BackPropagation** contiene un **BackpropagationLayer** ob-

para cada una de las capas de la red neuronal. Estos objetos deben ser borrados. El código siguiente pone a cero los errores anteriores de la **BackpropagationLayer** objetos.

para (capa final feedforwardLayer: este.network.getLayers()) { getBackpropagationLayer(capa). clearError();

}

Como su nombre indica, la retropropagación se propaga hacia atrás a través de la neu-

red ral.

para (int i = this.network.getLayers().size() - 1; i >= 0; i--) {

Obtenga cada capa de la red neuronal.

capa final feedforwardLayer = this.network.getLayers().get(i);

Ahora llame a cualquiera de las dos versiones sobrecargadas del método **calcError.** Si es la capa de salida, pase el **ideal** para la comparación. Si no es la capa de salida, entonces los valores **ideales** no son necesarios.

if (layer.isOutput()) { getBackpropagationLayer(capa). calcError(ideal);

} else { getBackpropagationLayer(capa). calcError();

}

}

La clase **BackpropagationLayer** tiene dos versiones del método **calcError.** La firma de la versión que opera en la capa de salida se muestra a continuación:

calcError del vacío público( doble ideal final[]) {

En primer lugar, calcule el porcentaje de errores para cada neurona. Bucle a través de todos los neu- rons de salida.

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

A continuación, establezca el error para esta neurona. El error es simplemente la diferencia entre la salida ideal y la salida real.

setError(i, ideal[i] – this.layer.getFire(i));

Calcula el delta para esta neurona. El delta es el error multiplicado por la deriva- tive de la función de activación. Enlazar el número, para que no se vuelva extremadamente pequeño o grande.

setErrorDelta(i, BoundNumbers.bound(calculateDelta(i)));

}

Estas deltas de error se utilizarán durante el proceso de aprendizaje que se cubre en el

siguiente sección.

Todas las demás capas de la red neuronal tendrán sus errores calculados por el

calcError método que no requiere una matriz **ideal.**

vacío público calcError() {

En primer lugar, obtenga la siguiente capa. Ya que nos estamos propagando hacia atrás, esta será la capa que acaba de ser procesada.

Final BackpropagationLayer siguiente = this.backpropagation.getBackpropagationLayer( this.layer.getNext());

Recose cada valor de matriz para las conexiones entre esta capa y la siguiente.

para (int i = 0; i < this.layer.getNext(). getNeuronCount(); i++) { para (int j = 0; j < this.layer.getNeuronCount(); j++) {

Los métodos de cálculo de errores se piden para cada conjunto de entrenamiento, por lo tanto, es nec- essary para acumular las deltas de matriz antes de que los errores se borran. Determine la contribución de esta capa al error mirando el delta de la siguiente capa y compárela con las salidas de esta capa. Dado que nos estamos propagando hacia atrás, la siguiente capa es en realidad la capa que acabamos de procesar.

accumulateMatrixDelta(j, i, next.getErrorDelta(i)

* this.layer.getFire(j));

Calcule y agregue al error de esta capa multiplicando el valor de matriz por su delta.

setError(j, getError(j) + this.layer.getMatrix().get(j, i)

* next.getErrorDelta(i));

}

Además, acumula deltas que afectan al umbral.

accumulateBiasDelta(i, next.getErrorDelta(i));

}

Para las capas ocultas, calcule el delta utilizando el derivado de la activación

función.

if (this.layer.isHidden()) {

deltas de capa oculta

para (int i = 0; i < this.layer.getNeuronCount(); i++) { setErrorDelta(i, BoundNumbers.bound(calculateDelta(i)));

}

}

Una vez calculados todos los errores, el método **learn** se puede utilizar para ap-

ply las deltas a la matriz de peso y enseñar a la red neuronal a reconocer mejor el patrón de entrada.

## Aprendizaje de backpropagation

El proceso de aprendizaje es relativamente simple. Todos los cambios deseados ya se calcularon durante el cálculo del error. Ahora se trata simplemente de aplicar estos cambios. Los valores de la tasa de aprendizaje y los parámetros de impulso afectarán a la forma en que se aplican estos cambios.

Se llama al método **learn** en el objeto **BackPropagation** para iniciar el proceso de aprendizaje.

público void learn()

Recorre todas las capas. La orden no es importante. Durante el cálculo del error, los resultados de una capa dependían de otra. Como resultado, era muy importante asegurarse de que el error se propagara hacia atrás. Sin embargo, durante el proceso de aprendizaje, los valores simplemente se aplican a las capas de red neuronal de una en una.

para (capa final feedforwardLayer : this.network.getLayers()) {

Llamar al método **learn** de cada uno de los objetos **BackpropagationLayer** hace que se apliquen los cambios calculados.

getBackpropagationLayer(capa).learn(this.learnRate, this.momentum);

}

Se utiliza el método **learn** proporcionado en la clase **BackpropagationLayer**

para realizar las modificaciones reales. La firma para el método de **aprendizaje** se muestra aquí:

aprendizaje del vacío público (doble aprendizaje finalRate , doble impulso final)

La capa **de aprendizaje** se asegura de que haya una matriz. Si no hay matriz, entonces no hay nada que entrenar.

if (this.layer.hasMatrix()) {

A continuación, se realiza una matriz que contiene los valores delta de matriz acumulados que son escalados por el **learnRate**. La tasa de aprendizaje se puede considerar como un porcentaje. Un valor de uno significa que las deltas se aplicarán sin escala.

matriz final m1 = MatrixMath.multiply(this.accMatrixDelta,

learnRate);

Las deltas anteriores se almacenan en la variable **matrixDelta.** El aprendizaje de la iteración anterior se aplica a la iteración actual escalada por la variable de **impulso.** Algunas variantes de **Backpropagation** no utilizan ningún impulso. Para no especificar ningún impulso, utilice un valor de **impulso** de cero.

matriz final m2 = MatrixMath.multiply(this.matrixDelta, momentum);

Agregue los dos juntos y almacene el resultado en la variable **matrixDelta.** Esto se utilizará con el impulso para la próxima iteración de entrenamiento.

this.matrixDelta = MatrixMath.add(m1, m2);

Agregue los valores calculados a la matriz actual. Esto modifica la matriz y causa aprendizaje.

this.layer.setMatrix(MatrixMath.add(this.layer.getMatrix(), this.matrixDelta));

Borre los errores para la siguiente iteración de aprendizaje.

this.accMatrixDelta.clear();

}

El método **learn** de la clase **BackpropagationLayer** se llama una vez por

capa.

# Resumen del capítulo

En este capítulo, aprendió cómo funcionauna red neuronal feedforward backpropagation. La red neuronal de retropropagación feedforward se compone en realidad de dos algoritmos de red neuronal. No es necesario utilizar siempre feedforward y backpropagation juntos, pero este es a menudo el caso. En los próximos capítulos se introducirán otros métodos de formación. El término "feedforward" se refiere a un método por el cual una red neuronal reconoce un patrón. El término "backpropagation" describe un proceso por el cual se entrenará la red neuronal.

Una red neuronal feedforward es una red en la que las neuronas solo están conectadas a la siguiente capa. No hay conexiones entre las neuronas en capas anteriores o entre las neuronas y ellas mismas. Además, las neuronas no están conectadas a las neuronas más allá de la siguiente capa. A medida que un patrón es procesado por un diseño feedforward, se aplicarán las antigüedades y los pesos de conexión.

Las redes neuronales se pueden entrenar mediante backpropagation. La backpropagation es una forma de capacitación supervisada. La red neuronal se presenta con los datos de entrenamiento, y los resultados de la red neuronal se comparan con los resultados esperados. La diferencia entre los resultados reales y los resultados esperados es el error. La backpropagation es un método mediante el cual los pesos y el umbral de entrada de la red neuronal se alteran de una manera que hace que este error se reduzca.

La backpropagation no es la única manera de entrenar una red neuronal feedforward. El recocido y los algoritmos genéticos son otros dos métodos comunes. El siguiente chap- ter demostrará cómo se puede utilizar un algoritmo genético para entrenar una red neuronal.

# vocabulario

Función de activación Derivado de la retropropagación

Feedforward

Tasa de aprendizaje de la función de activación tangente hiperbólica

Impulso de la función de activación lineal

Sobreajuste

Subadaptación de la función de activación sigmoide

# Preguntas para revisión

1. ¿Qué es una función de activación? Explique cuándo puede utilizar la función de activación tangente hiperbólica sobre la función de activación sigmoide.
2. ¿Cómo puede determinar si una función de activación es compatible con el método de entrenamiento de backpropagation?
3. Considere una red neuronal con una neurona de salida y tres neuronas de entrada. Los pesos entre las tres neuronas de entrada y la neurona de una sola salida son 0.1, 0.2 y 0.3. El umbral es 0,5. ¿Cuál sería el valor de salida para una entrada de 0,4, 0,5 y 0,6?
4. Explicar el papel de la tasa de aprendizaje en la backpropagation.
5. Explicar el papel del impulso en la backpropagation.