11

Implementación de una red neuronal artificial multicapa desde cero

Como ya sabrá, el aprendizaje profundo está recibiendo mucha atención por parte de la prensa y es, sin duda, el tema más candente en el campo del aprendizaje automático. El aprendizaje profundo puede entenderse como un subcampo del aprendizaje automático que se ocupa de entrenar **redes neuronales** artificiales (NN) con muchas capas de forma eficiente. En este capítulo, aprenderás los conceptos básicos de las redes neuronales artificiales para que estés bien equipado para los siguientes capítulos, en los que se presentarán bibliotecas avanzadas de aprendizaje profundo basadas en Python y arquitecturas de **redes neuronales profundas (DNN)** que son particularmente adecuadas para el análisis de imágenes y textos.

Los temas que trataremos en este capítulo son los siguientes:

- Comprensión conceptual de las NN multicapa
- Implementación desde cero del algoritmo fundamental de retropropagación para el entrenamiento de redes neuronales
- Entrenamiento de una NN multicapa básica para la clasificación de imágenes

Modelización de funciones complejas con redes neuronales artificiales

Al principio de este libro, comenzamos nuestro viaje por los algoritmos de aprendizaje automático con neuronas artificiales en *el capítulo 2*, *Entrenamiento de algoritmos sencillos de aprendizaje automático para la clasificación*. Las neuronas artificiales representan los bloques de construcción de las NN artificiales multicapa que discutiremos en este capítulo.

El concepto básico de las NN artificiales se construyó a partir de hipótesis y modelos sobre cómo funciona el cerebro humano para resolver tareas problemáticas complejas. Aunque las NN artificiales han ganado mucha popularidad en los últimos años, los primeros estudios sobre ellas se remontan a la década de 1940, cuando Warren McCulloch y Walter Pitts describieron por primera vez cómo podían funcionar las neuronas. (*Un cálculo lógico de las ideas inmanentes a la actividad nerviosa*, de *W. S. McCulloch* y *W. Pitts*, *The Bulletin of Mathematical Biophysics*, 5(4):115-133, 1943).

Sin embargo, en las décadas que siguieron a la primera implementación del modelo **neuronal McCulloch-Pitts** -el perceptrón de Rosenblatt en la década de 1950-, muchos investigadores y profesionales del aprendizaje automático empezaron a perder lentamente el interés por las NN, ya que nadie tenía una buena solución para entrenar una NN con múltiples capas. Finalmente, el interés por las NN se reavivó en 1986, cuando D*.E. Rumelhart*, *G.E. Hinton* y *R.J. Williams* participaron en el (re)descubrimiento y popularización del algoritmo de *retropropagación* para entrenar NN de forma más eficiente, del que hablaremos con más detalle más adelante en este capítulo (*Learning representations by backpropagating errors*, by *D.E. Rumelhart*, *G.E. Hinton*, and *R.J. Williams*, *Nature*, 323 (6088): 533-536, 1986). También se recomienda a los lectores interesados en la historia de la **inteligencia artificial** (**IA)**, el aprendizaje automático y las NN que lean el artículo de Wikipedia sobre los llamados *inviernos de la IA*, que son los periodos de tiempo en los que una gran parte de la comunidad investigadora perdió el interés por el estudio de las NN (https://en.wikipedia.org/wiki/AI_winter).

Sin embargo, las NN son hoy más populares que nunca gracias a los numerosos avances logrados en la década anterior, que han dado lugar a lo que ahora llamamos algoritmos y arquitecturas de aprendizaje profundo: NN compuestas por muchas capas. Las NN son un tema candente no solo en la investigación académica, sino también en las grandes empresas tecnológicas, como Facebook, Microsoft, Amazon, Uber, Google y muchas más que invierten mucho en NN artificiales y en la investigación del aprendizaje profundo.

A día de hoy, las NN complejas impulsadas por algoritmos de aprendizaje profundo se consideran soluciones de vanguardia para la resolución de problemas complejos, como el reconocimiento de imágenes y de voz. Algunas de las aplicaciones más recientes son:

- Predicción de las necesidades de recursos de COVID-19 a partir de una serie de radiografías (https://arxiv.org/ abs/2101.04909)
- Modelización de las mutaciones de los virus (https://science.sciencemag.org/content/371/6526/284)
- Aprovechamiento de los datos de las plataformas de medios sociales para gestionar fenómenos meteorológicos extremos (https:// onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311)
- Mejorar las descripciones de las fotos para personas ciegas o con problemas de visión (https://tech. fb.com/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are- blind-or-visually-impaired/)

Recapitulación de la red neuronal monocapa

Este capítulo trata sobre las NN multicapa, su funcionamiento y cómo entrenarlas para resolver problemas complejos. Sin embargo, antes de profundizar en una arquitectura particular de NN multicapa, vamos a reiterar brevemente algunos de los conceptos de las NN monocapa que introdujimos en el *capítulo 2*, a saber, el algoritmo **ADAp- tive LInear NEuron** (**Adaline**), que se muestra en *la figura 11.1*:

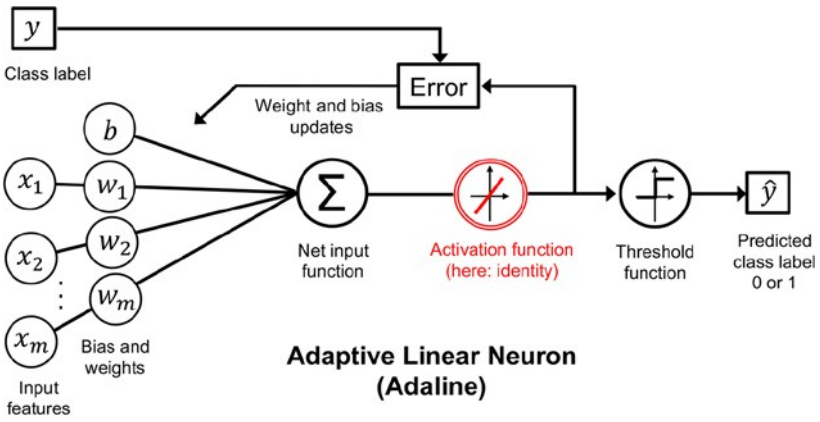

Figura 11.1: El algoritmo Adaline

En el *Capítulo 2*, implementamos el algoritmo Adaline para realizar la clasificación binaria, y utilizamos el algoritmo de optimización de descenso de gradiente para aprender los coeficientes de peso del modelo. En cada epoch (pasada sobre el conjunto de datos de entrenamiento), actualizamos el vector de pesos $\boldsymbol{w}$ y la unidad de sesgo b mediante la siguiente regla de actualización:

$$\boldsymbol{ww}\ \boldsymbol{w}\ \boldsymbol{ww} + \boldsymbol{wwww}\ b\ \boldsymbol{w}\ b + \boldsymbol{wb}$$

donde $\Delta \boldsymbol{wj} = -\eta \frac{\partial L}{\partial \partial \partial j}$ y $\Delta b\ b\ b\eta \frac{\partial L}{\partial b}$ para la unidad de sesgo y cada peso w_j en el vector de pesos $\boldsymbol{w}$.

En otras palabras, calculamos el gradiente basándonos en todo el conjunto de datos de entrenamiento y actualizamos los pesos del modelo dando un paso en la dirección opuesta al gradiente de pérdida $\nabla L(\boldsymbol{ww})$. (Para simplificar, nos centraremos en los pesos y omitiremos la unidad de sesgo en los párrafos siguientes; sin embargo, como recordará del *Capítulo 2*, se aplican los mismos conceptos). Para encontrar los pesos óptimos del modelo, optimizamos una función objetivo que definimos como la función de pérdida de la **media de errores al cuadrado** (**MSE**)
$L(\boldsymbol{w})$. Además, multiplicamos el gradiente por un factor, la **tasa de aprendizaje** η, que tuvimos que elegir cuidadosamente para equilibrar la velocidad de aprendizaje con el riesgo de sobrepasar el mínimo global de la función de pérdida.

En la optimización por descenso de gradiente, actualizamos todos los pesos simultáneamente después de cada época, y definimos la derivada parcial para cada peso w_j en el vector de pesos, $\boldsymbol{w}$, como sigue:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i \left(y^{(i)} - a^{(i)}\right)^2 = -\frac{2}{n} \sum_i \left(y^{(i)} - a^{(i)}\right) x_j^{(i)}$$

Aquí, $y^{(i)}$ es la etiqueta de clase objetivo de una muestra particular $x^{(i)}$, y $a^{(i)}$ es la activación de la neurona, que es una función lineal en el caso especial de Adaline.

Además, definimos la función de activación $\sigma(\cdot)$ como sigue:

$$\sigma(\cdot) = z = a$$

Aquí, la entrada neta, z, es una combinación lineal de los pesos que conectan la capa de entrada con la capa de salida:

$$z = \sum_j w_j x_j + b = \boldsymbol{w}^T \boldsymbol{x} + b$$

Aunque utilizamos la activación $\sigma(\cdot)$ para calcular la actualización del gradiente, implementamos una función de umbral para convertir la salida de valor continuo en etiquetas de clase binarias para la predicción:

$$y = \begin{cases} 1 & \text{si } z \geq \theta \\ 0 & \text{de lo contrario} \end{cases}$$

Convención de nomenclatura monocapa

Obsérvese que, aunque Adaline consta de dos capas, una de entrada y otra de salida, se denomina red monocapa por su único enlace entre las capas de entrada y salida.

Además, aprendimos un *truco* para acelerar el aprendizaje del modelo, la llamada optimización **estocástica de descenso gra- diente** (**SGD**). La SGD aproxima la pérdida de una sola muestra de entrenamiento (aprendizaje en línea) o de un pequeño subconjunto de ejemplos de entrenamiento (aprendizaje por mini lotes). Utilizaremos este concepto más adelante en este capítulo cuando implementemos y entrenemos un **perceptrón multicapa** (**MLP**). Aparte de un aprendizaje más rápido -debido a las actualizaciones de pesos más frecuentes en comparación con el descenso de gradiente-, su naturaleza ruidosa también se considera beneficiosa cuando se entrenan NN multicapa con funciones de activación no lineales, que no tienen una función de pérdida convexa. En este caso, el ruido añadido puede ayudar a escapar de los mínimos de pérdida locales, pero trataremos este tema con más detalle más adelante en este capítulo.

Introducción a la arquitectura de redes neuronales multicapa

En esta sección, aprenderá cómo conectar múltiples neuronas simples a una NN multicapa feedforward; este tipo especial de red *totalmente conectada* también se denomina **MLP**.

La figura 11.2 ilustra el concepto de un MLP formado por dos capas:

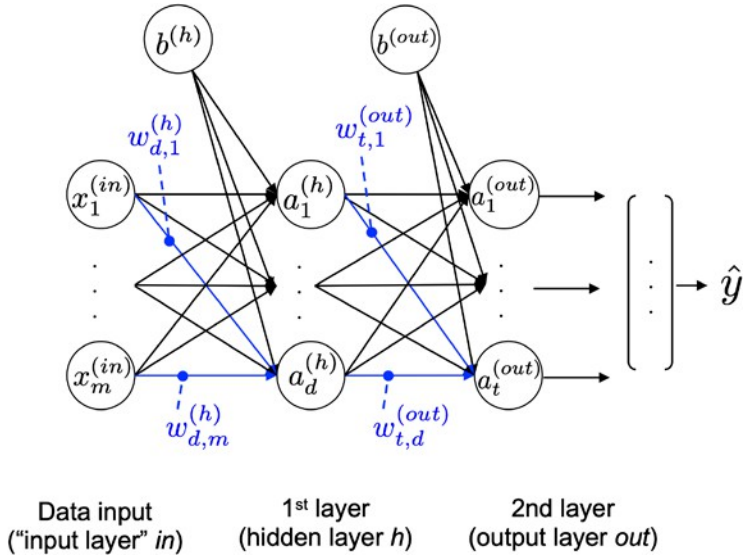

Figura 11.2: Un MLP de dos capas

Junto a la entrada de datos, el MLP representado en *la Figura 11.2* tiene una capa oculta y una capa de salida. Las unidades de la capa oculta están totalmente conectadas a las características de entrada, y la capa de salida está totalmente conectada a la capa oculta. Si una red de este tipo tiene más de una capa oculta, también la llamaremos **NN profunda**. (Tenga en cuenta que, en algunos contextos, las entradas también se consideran una capa. Sin embargo, en este caso, convertiría el modelo Adaline, que es una red neuronal de una capa, en una red neuronal de dos capas, lo que puede resultar contraintuitivo).

Añadir capas ocultas adicionales

Podemos añadir cualquier número de capas ocultas al MLP para crear arquitecturas de red más profundas. En la práctica, podemos pensar en el número de capas y unidades de una NN como hiperparámetros adicionales que queremos optimizar para una tarea problemática determinada utilizando la técnica de validación cruzada, que ya tratamos en el *Capítulo 6*, *Mejores prácticas de aprendizaje para la evaluación de modelos y el ajuste de hiperparámetros*.

Sin embargo, los gradientes de pérdida para actualizar los parámetros de la red, que calcularemos más adelante mediante retropropagación, serán cada vez más pequeños a medida que se añadan más capas a una red. Este problema de gradiente evanescente dificulta el aprendizaje de modelos. Por lo tanto, se han desarrollado algoritmos especiales para ayudar a entrenar tales estructuras DNN; esto se conoce como **aprendizaje profundo**, que discutiremos con más detalle en los siguientes capítulos.

Como se muestra en *la Figura 11.2*, denotamos la i-ésima unidad de activación en la l-ésima capa como $a_i^{(l)}$. Para que las implementaciones matemáticas y de código sean un poco más intuitivas, no utilizaremos índices numéricos para referirnos a las capas, sino que utilizaremos el superíndice *in* para las características de entrada, el superíndice *h* para la capa oculta y el superíndice *out* superíndice para la capa de salida. Por ejemplo, $x_i^{(in)}$ se refiere al i-ésimo valor de la característica de entrada, $a_i^{(h)}$ se refiere a la unidad ith en la capa oculta, y $a_i^{(out)}$ se refiere a la unidad ith en la capa de salida. Observe que las $\boldsymbol{b}$ de la *Figura 11.2* denotan las unidades de sesgo. De hecho, $\boldsymbol{b}^{(h)}$ y $\boldsymbol{b}^{(out)}$ son vectores cuyo número de elementos es igual al número de nodos de la capa a la que corresponden. Por ejemplo, $\boldsymbol{b}^{(h)}$ almacena *d* unidades de polarización,
donde *d* es el número de nodos de la capa oculta. Si esto le parece confuso, no se preocupe. Si miras la implementación del código más adelante, donde inicializamos las matrices de pesos y los vectores unitarios de sesgo, te ayudará a aclarar estos conceptos.

Cada nodo de la capa *l* está conectado a todos los nodos de la capa $l + 1$ mediante un coeficiente de peso. Por ejemplo, la conexión entre la unidad k de la capa *l* y la unidad j de la capa $l + 1$ se escribirá como $ww_{j,k}^{(l)}$. Volviendo a la *Figura 11.2*, denotamos la matriz de pesos que conecta la entrada con la capa oculta como $\boldsymbol{W}^{(h)}$, y escribimos la matriz que conecta la capa oculta con la capa de salida como $\boldsymbol{W}^{(out)}$.

Mientras que una unidad en la capa de salida sería suficiente para una tarea de clasificación binaria, en la figura anterior vimos una forma más general de NN, que nos permite realizar una clasificación multiclase mediante una generalización de la técnica de **uno contra todos** (**OvA**). Para entender mejor cómo funciona esto, recuerde la representación de un solo **punto** de variables categóricas que introdujimos en el *Capítulo 4*, *Creación de buenos conjuntos de datos de entrenamiento - Preprocesamiento de datos*.

Por ejemplo, podemos codificar las tres etiquetas de clase en el conocido conjunto de datos Iris (*0=Setosa*, 1=Versicol- *o*, *2=Virginica*) de la siguiente manera:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Esta representación vectorial de un solo punto nos permite abordar tareas de clasificación con un número arbitrario de etiquetas de clase únicas presentes en el conjunto de datos de entrenamiento.

Si eres nuevo en las representaciones NN, la notación de indexación (subíndices y superíndices) puede parecer un poco confusa al principio. Lo que puede parecer demasiado complicado al principio tendrá mucho más sentido en secciones posteriores, cuando vectoricemos la representación NN. Como se ha introducido anteriormente, resumimos los pesos que conectan las capas de entrada y oculta mediante una matriz dimensional $d \times m$ $\boldsymbol{W}^{(h)}$, donde *d* es el número de unidades ocultas y *m* es el número de unidades de entrada.

Activación de una red neuronal mediante propagación hacia delante

En esta sección, describiremos el proceso de **propagación hacia delante** para calcular la salida de un modelo MLP. Para entender cómo encaja en el contexto del aprendizaje de un modelo MLP, vamos a resumir el procedimiento de aprendizaje MLP en tres sencillos pasos:

1. Comenzando en la capa de entrada, propagamos hacia delante los patrones de los datos de entrenamiento a través de la red para generar una salida.
2. Basándonos en la salida de la red, calculamos la pérdida que queremos minimizar utilizando una función de pérdida que describiremos más adelante.

3. Retropropagamos la pérdida, hallamos su derivada con respecto a cada unidad de peso y sesgo de la red y actualizamos el modelo.

Por último, después de repetir estos tres pasos durante varias épocas y aprender los parámetros de peso y sesgo de la MLP, utilizamos la propagación hacia delante para calcular la salida de la red y aplicamos una función de umbral para obtener las etiquetas de clase predichas en la representación de un disparo, que describimos en la sección anterior.

Ahora, vamos a recorrer los pasos individuales de la propagación hacia delante para generar una salida a partir de los patrones de los datos de entrenamiento. Dado que cada unidad de la capa oculta está conectada a todas las unidades de las capas de entrada, primero calculamos la unidad de activación de la capa oculta $a^{(h)}$ de la siguiente manera:

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma(z_1^{(h)})$$

Aquí, $z^{(h)}$ es la entrada de la red y $\sigma(\cdot)$ es la función de activación, que tiene que ser diferenciable para aprender los pesos que conectan las neuronas utilizando un enfoque basado en el gradiente. Para poder resolver problemas complejos como la clasificación de imágenes, necesitamos funciones de activación no lineales en nuestro modelo MLP, por

Por ejemplo, la función de activación sigmoidea (logística) que recordamos de la sección sobre regresión logística del *capítulo 3*, *Recorrido por los clasificadores de aprendizaje automático con Scikit-Learn*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Como recordará, la función sigmoidea es una curva *en forma de* S que mapea la entrada neta z en una distribución logística en el rango de 0 a 1, que corta el eje y en $z = 0$, como se muestra en *la Figura 11.3*:

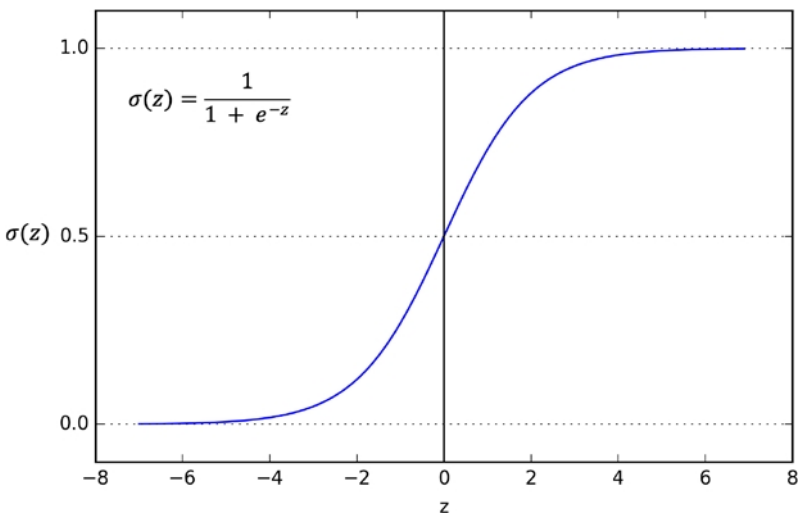

Figura 11.3: Función de activación sigmoidea

El MLP es un ejemplo típico de NN artificial feedforward. El término **feedforward** se refiere al hecho de que cada capa sirve de entrada a la capa siguiente sin bucles, en contraste con las NN recurrentes, una arquitectura que discutiremos más adelante en este capítulo y con más detalle en *el capítulo 15*, *Modelado de datos secuenciales mediante redes neuronales recurrentes*. El término *perceptrón multicapa* puede sonar un poco confuso, ya que las neuronas artificiales de esta arquitectura de red suelen ser unidades sigmoidales, no perceptrones. Podemos pensar en las neuronas del MLP como unidades de regresión logística que devuelven valores en el rango continuo entre 0 y 1.

A efectos de eficiencia y legibilidad del código, ahora escribiremos la activación de una forma más compacta utilizando los conceptos del álgebra lineal básica, lo que nos permitirá vectorizar nuestra implementación del código a través de NumPy en lugar de escribir múltiples bucles `for` de Python anidados y costosos computacionalmente:

$$z^{(h)} = \mathbf{x}^{(in)}\mathbf{W}^{(h)T} + b^{(h)}$$

$$a^{(h)} = \sigma(z^{(h)})$$

Aquí, $\mathbf{z}^{(h)}$ es nuestro vector de características $1 \times m$ dimensional. $\mathbf{W}^{(h)}$ es una matriz de pesos $d \times m$ dimensional donde d es el número de unidades en la capa oculta; por consiguiente, la matriz transpuesta $\mathbf{W}^{(h)T}$ es $m \times d$ *dimensional*. El vector de sesgo $\mathbf{b}^{(h)}$ consta de d unidades de sesgo (una unidad de sesgo por nodo oculto).

Tras la multiplicación matricial-vectorial, obtenemos el vector de entrada de red $1 \times d$ dimensional $\mathbf{z}^{(h)}$ para calcular la activación $\mathbf{a}^{(h)}$ (donde $a^{(h)} \in \mathbb{R}^{1 \times d}$).

Además, podemos generalizar este cálculo a todos los n ejemplos del conjunto de datos de entrenamiento:

$$\mathbf{Z}^{(h)} = \mathbf{X}^{(in)}\mathbf{W}^{(h)T} + \mathbf{b}^{(h)}$$

Aquí, $\mathbf{X}^{(in)}$ es ahora una matriz n×m, y la multiplicación matricial dará como resultado una matriz de entrada neta de dimensión n×d, $\mathbf{Z}^{(h)}$. Finalmente, aplicamos la función de activación $\sigma(\cdot)$ a cada valor de la matriz de entrada neta para obtener la matriz de activación n×d en la siguiente capa (aquí, la capa de salida):

$$A^{(h)} = \sigma(Z^{(h)})$$

Del mismo modo, podemos escribir la activación de la capa de salida en forma vectorizada para múltiples ejemplos:

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)}\mathbf{W}^{(out)T} + \mathbf{b}^{(out)}$$

Aquí, multiplicamos la transposición de la matriz $t \times d$ $\mathbf{W}^{(out)}$ (t es el número de unidades de salida) por la matriz de dimensión n×d, $\mathbf{A}^{(h)}$, y añadimos el vector de sesgo de dimensión t $\mathbf{b}^{(out)}$ para obtener la matriz de dimensión $n \times t$, $\mathbf{Z}^{(out)}$. (Las columnas de esta matriz representan las salidas de cada muestra).

Por último, aplicamos la función de activación sigmoidea para obtener la salida de valor continuo de nuestra red:

$$A^{(out)} = \sigma(Z^{(out)})$$

De forma similar a $\mathbf{Z}^{(out)}$, $\mathbf{A}^{(out)}$ es una matriz de dimensión $n \times t$.