

Redes Neuronales

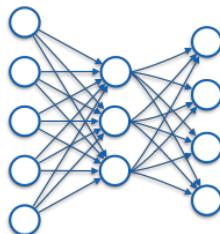
Índice

- Introducción
- Neuronas artificiales
- Interpretación geométrica
- Arquitectura de la red
- Diseño de la red
- Entrenamiento de la red
- *Backpropagation*
- Aprendizaje estocástico
- Ajuste de la red

Introducción

Sistema computacional inspirado en redes neurales biológicas

Conjunto de **neuronas artificiales** conectadas entre si



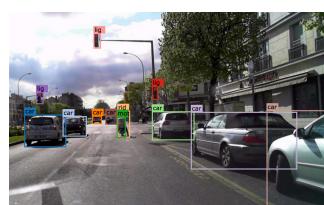
Son capaces de **aprender** presentándole una serie de ejemplos

Los ejemplos deben estar *etiquetados* con la salida esperada

Se van “*ajustando*” las conexiones entre neuronas

Áreas de aplicación

Reconocimiento de imagen



Reconocimiento del habla

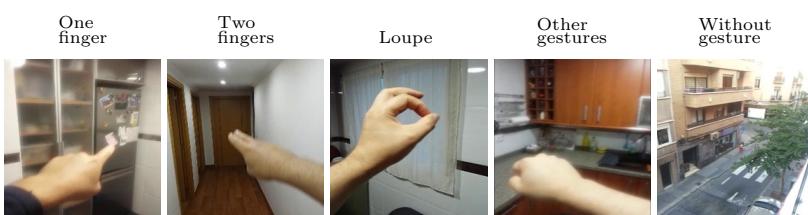
Procesamiento del lenguaje natural

Conducción autónoma

Diagnóstico médica

...

Ejemplo: Reconocimiento de gestos

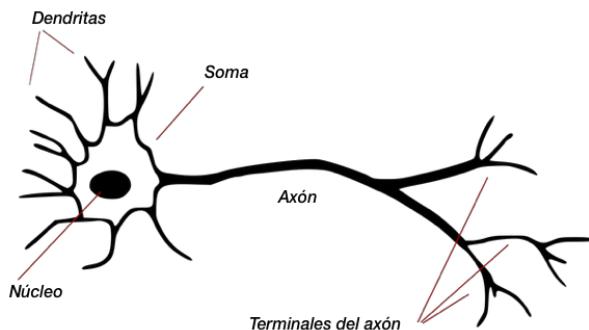


Neuronas biológicas

Dendritas: Reciben entrada (potencial eléctrico) de otras neuronas

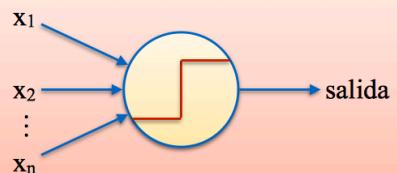
Soma: Integra las entradas. Es un dispositivo “*todo o nada*”, se activa si recibe suficiente potencial de entrada.

 **Axón:** Transporta la salida a otras neuronas. La comunicación entre el axón de una neurona y las *dendritas* de otra se denomina *sinapsis*.

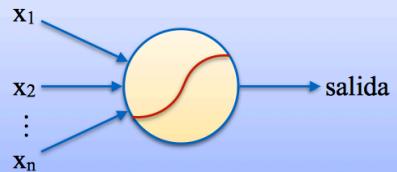


Neuronas artificiales

Perceptrón

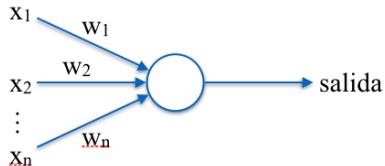


Neurona sigmoidea



Perceptrón

Neurona artificial que toma una serie de **entradas** x y produce una *salida*



El perceptrón **toma una decisión** (*salida*) ponderando (w) una serie de factores (x)

$$\text{salida} = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq \text{umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{umbral} \end{cases}$$

Perceptrón: Notación matricial

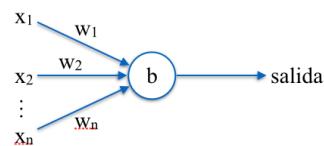
Podemos representar las entradas y pesos **como tuplas**

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

El **bias** (b) nos indica lo fácil que es que el perceptrón “se dispare” $b = -\text{umbral}$

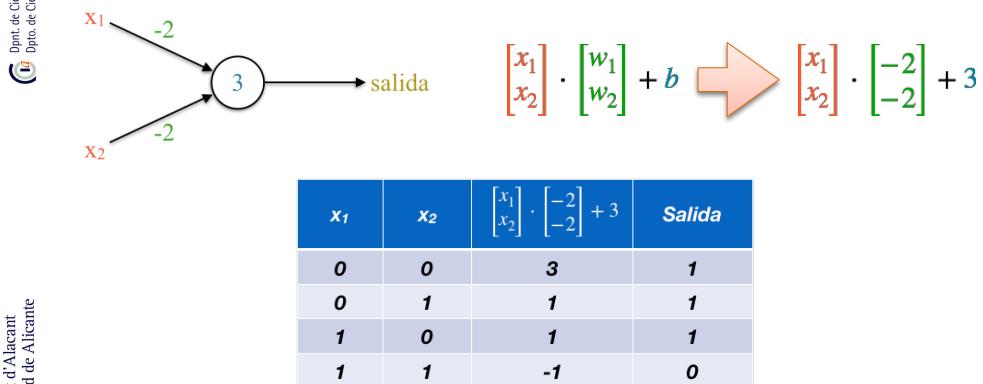
Podemos reescribir la **salida** como

$$\text{salida} = \begin{cases} 0 & \text{si } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{si } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$



Ejemplo

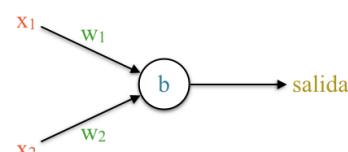
Operación NAND



- Podemos implementar cualquier operación lógica

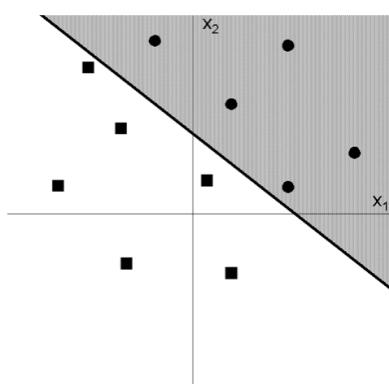
Interpretación geométrica

- Podemos ver un **perceptrón de dos entradas**, como un clasificador de los puntos que quedan a cada lado de una recta



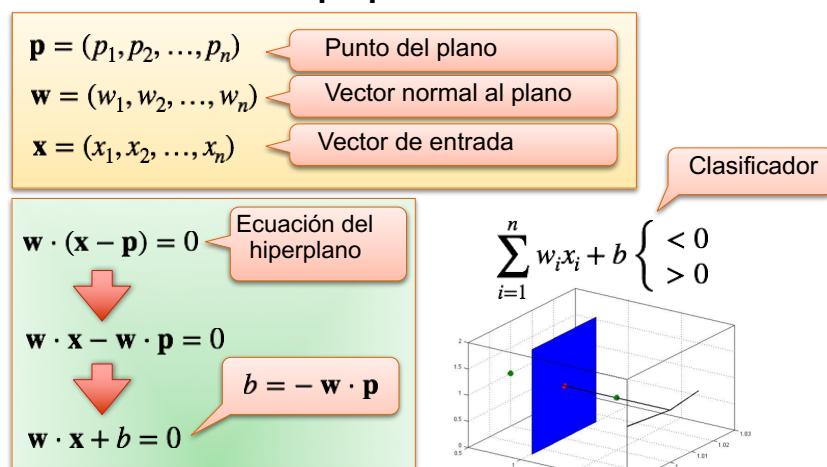
$$\text{salida} = \begin{cases} 0 & \text{si } w_1x_1 + w_2x_2 + b \leq 0 \\ 1 & \text{si } w_1x_1 + w_2x_2 + b > 0 \end{cases}$$

$w_1x_1 + w_2x_2 + b = 0$ Ecuación de la recta



Interpretación geométrica

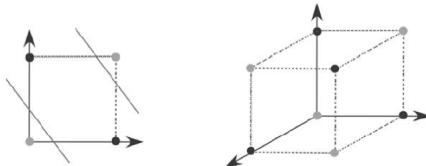
En el caso general de un perceptrón de n entradas, los datos se clasifican mediante un **hiperplano** de n dimensiones



No-separabilidad lineal

Existen situaciones donde **un único hiperplano** no puede separar los datos

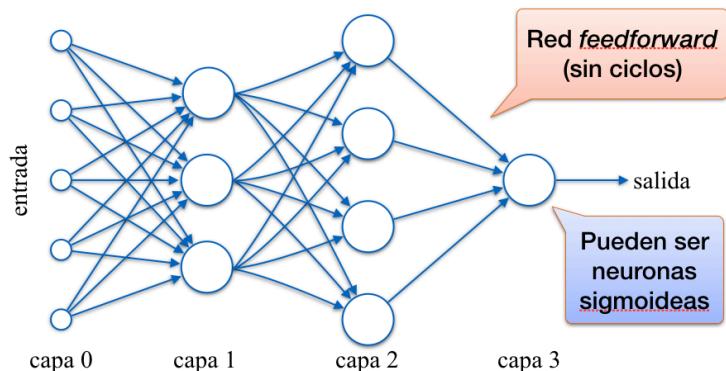
Por ejemplo cuando la frontera de decisión es curva



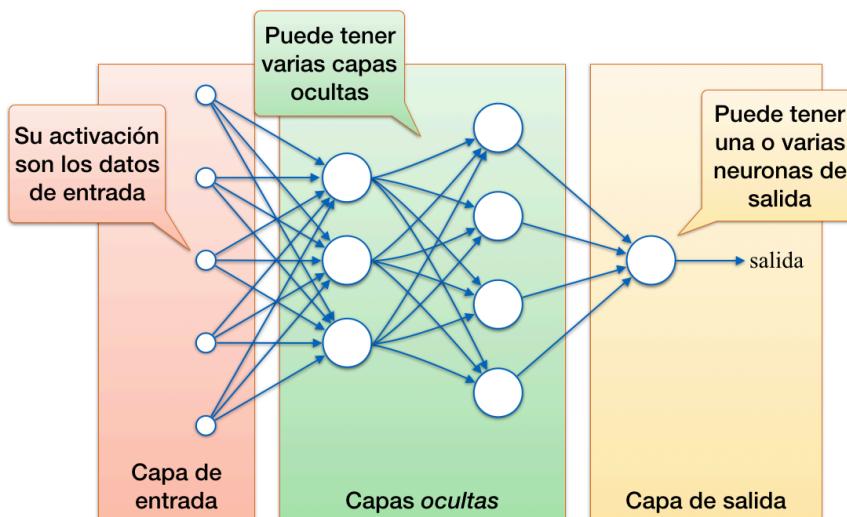
Podemos **combinar varios perceptrones** conectados entre si para implementar funciones más complejas

Percepción Multicapa (MLP)

- Organizaremos las neuronas en forma de **capas**
 - La salida de las neuronas de una capa será la entrada de las de la siguiente
 - A mayor **número de capas**, podrá tomar decisiones **más complejas**



Arquitectura de la red



Tipos de redes

• Shallow Networks

- Sólo una capa oculta

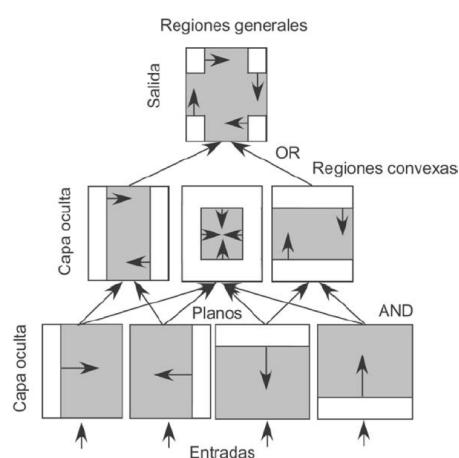
• Deep Networks

- Dos o más capas ocultas
- Habitualmente se entrenan redes entre 5 y 10 capas
- Se entrena mediante **descenso por gradiente** y **back propagation**
- Nuevas técnicas han posibilitado el entrenamiento de estas redes más complejas

Interpretación geométrica

Se demuestra que un perceptrón con dos capas puede **aproximar cualquier función**

Las capas ocultas nos permiten incorporar distintas **estrategias de separación**



Diseño de la red

- El diseño de las capas de entrada y salida suele ser directo

- **Capa de entrada**

- Tenemos en cuenta cómo podemos descomponer los datos que recibimos como diferentes neuronas de entrada

- **Capa de salida**

- Tenemos en cuenta cómo podemos codificar el resultado que queremos obtener como salida

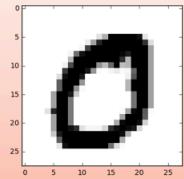
- El diseño de las **capas ocultas** no es trivial

Ejemplo: MNIST

Base de datos MNIST (<http://yann.lecun.com/exdb/mnist/>)

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Ejemplo: MNIST

- **Capa de entrada**
 - Imágenes de 28x28 píxeles
 - Niveles de gris de [0, 1]
- 
- ↓
- **784 neuronas** de entrada con valores [0, 1]
-
- **Capa de salida**
 - Número del 0 al 10
- 
- ↓
- **10 neuronas** con valores {0, 1}

- **¿Cómo ajustamos los pesos?**

¡Puede ser entrenada!

Entrenamiento

- No le decimos a la máquina **cómo** resolver un problema
 - La máquina **lo aprende** a partir de observar ejemplos
- Necesitamos entrenar la red con un **gran número de ejemplos** para ajustar los pesos que produzcan la salida deseada
- Dividiremos la base de datos en **dos conjuntos**:
 - **Conjunto de entrenamiento (training)**
 - Ejemplos con los que ajustamos los pesos de la red
 - **Conjunto de prueba (test)**
 - Ejemplos para validar los resultados se clasificación de la red

Ejemplos de la base de datos

Para cada ejemplo tendremos una **entrada x** y una **salida esperada $y(x)$**

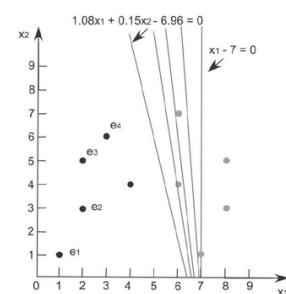
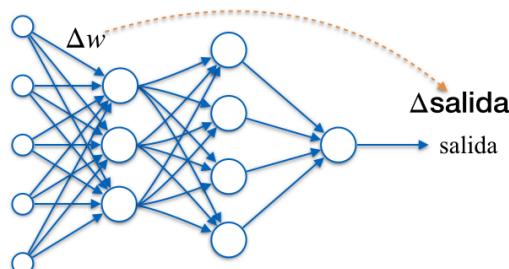


Por ejemplo, en el caso del reconocimiento de dígitos la salida esperada para diferentes ejemplos podría ser:

x	$y(x)$
0	$[1,0,0,0,0,0,0,0,0]^T$
1	$[0,1,0,0,0,0,0,0,0]^T$
5	$[0,0,0,0,0,1,0,0,0,0]^T$

¿Cómo entrenamos la red?

- **Idea:** Supongamos que una pequeña modificación en un peso provoca una pequeña modificación de la salida



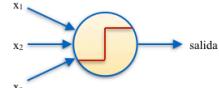
- Podemos ir **modificando los pesos y biases** de forma que nos acerque a la salida deseada
- **Problema:** La salida del perceptrón no se modifica poco a poco, es un escalón

Neurona sigmoidea

- Definimos: $z = w \cdot x + b$

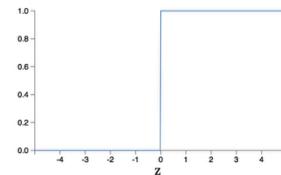
Entrada ponderada de la neurona

Perceptrón

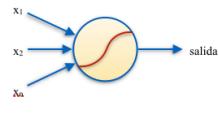


Función de activación

$$f(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases}$$

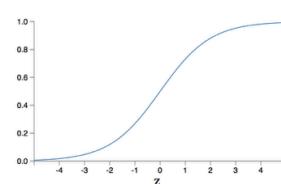


Neurona sigmoidea



Función de activación

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



Propiedades de la neurona sigmoidea

Con valores de z de gran magnitud equivale a la función escalón

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



$$z \ll 0 : \lim_{z \rightarrow -\infty} \sigma(z) = 0$$

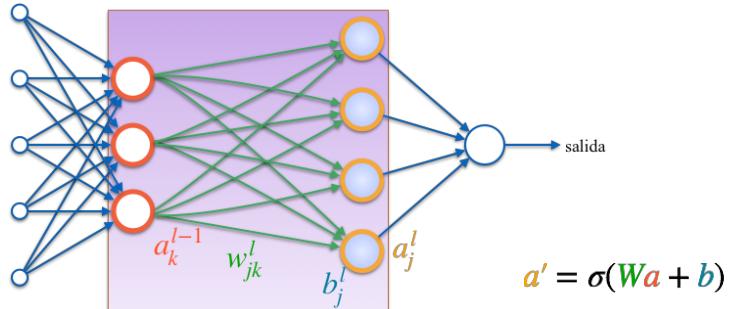
$$z \gg 0 : \lim_{z \rightarrow \infty} \sigma(z) = 1$$

Podemos ver la función *sigmoidea* como un **escalón suavizado**

Pequeños cambios de los pesos Δw y del *bias* Δb producen pequeños cambios en la salida

$$\Delta \text{salida} \approx \sum_j \frac{\partial \text{salida}}{\partial w_j} \Delta w_j + \frac{\partial \text{salida}}{\partial b} \Delta b$$

Cálculo de la activación en cada capa

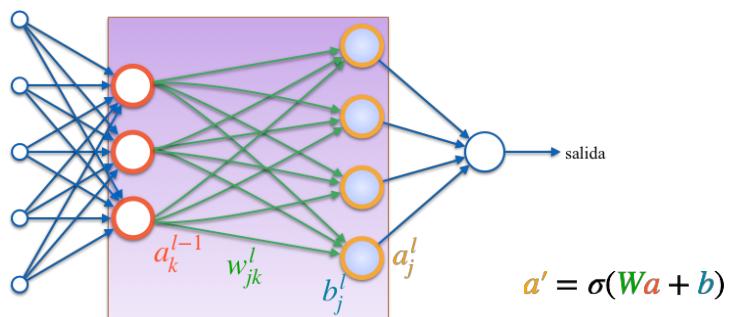


$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad a' = \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \end{bmatrix}$$

Tema 9. Redes Neuronales

25

Cálculo de la activación en cada capa

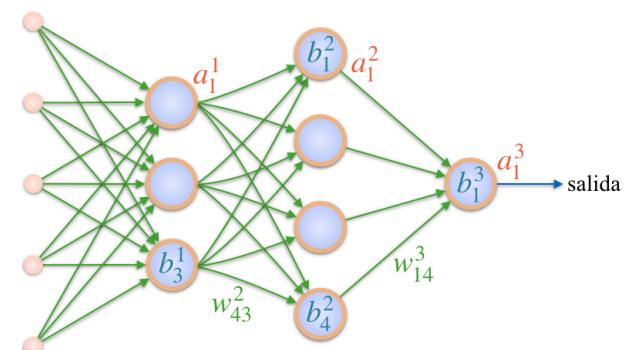
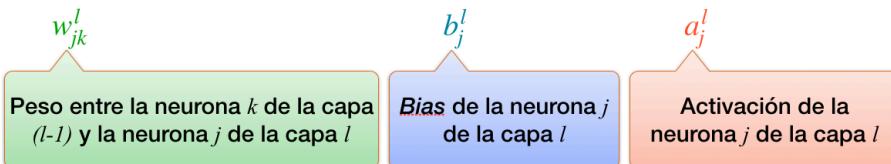


$$\begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$

Tema 9. Redes Neuronales

26

Notación de la red



Cálculo de la activación

Entrada:

Datos de entrada:

 x

Matrices de pesos de cada capa:

 $w = [W^1, W^2, \dots, W^L]$

Tuplas de biases de cada capa:

 $b = [b^1, b^2, \dots, b^L]$

Algoritmo FeedForward

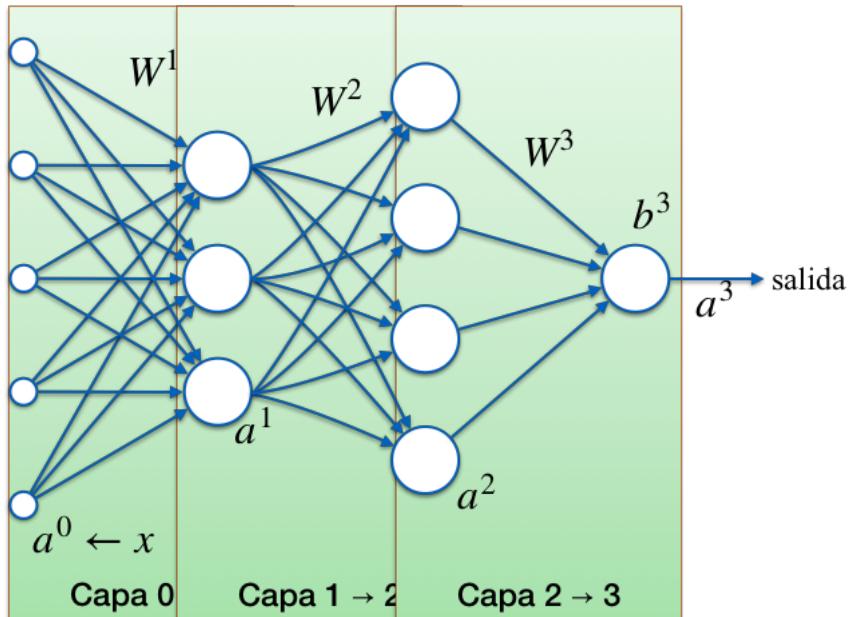
$$a \leftarrow x$$

Para cada capa $l \in [1, \dots, L]$

$$a \leftarrow \sigma(W^l a + b^l)$$

Devolver a

Cálculo de la activación



Función de coste

Debemos buscar los **pesos w** y **biases b** de toda la red que produzcan las salidas deseadas.

Dado:

a : **salida real** de la red para la entrada x , pesos w y biases b

$y(x)$: **salida esperada** de la red para la entrada x

n : **Número total de ejemplos** de entrenamiento (*training*)

Podemos definir una **función de coste C** que evalúe el error cuadrático medio (MSE) de clasificación de la red:

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Debemos minimizar su valor

Minimización del coste

Pequeños cambios en los pesos y *biases* producirán pequeños cambios en el coste

La función **es derivable**



Debemos actualizar los **pesos w y biases b** de forma que contribuyan a minimizar el coste $C(w,b)$

Utilizamos **descenso por gradiente**

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l} \quad b_j^l \leftarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l}$$

Tasa de aprendizaje

¿Cómo calculamos la **derivada parcial de C** respecto a cada variable?



Cálculo del gradiente

Podemos expresar la **activación de una neurona** en función de las activaciones de la capa anterior

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) \quad \rightarrow \quad a^l = \sigma (W^l a^{l-1} + b^l)$$

Podemos calcular la activación calculando previamente como paso intermedio la **entrada ponderada z**

$$a^l = \sigma (z^l)$$

$$z^l = w^l a^{l-1} + b^l$$

Cálculo del gradiente

Podemos expresar la función de coste a optimizar a partir de la activación de la **última capa L**



$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Podemos expresar el coste como la suma del **coste individual** para cada entrada x

Backpropagation

Nos permite **calcular las derivadas parciales** de la función de coste para cada ejemplo individual de entrenamiento



- Dado que la entrada x es fija, la salida esperada $y(x)$ también lo será
- El coste se calculará en función de las activaciones a
- Deberemos modificar pesos w y *bias* b para optimizarlo

Calcula el **error obtenido en la salida de la red**, y lo propaga hacia las capas anteriores

Definimos:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

“Error” producido en la neurona j de la capa l

Backpropagation

Error en la capa de salida L

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

Regla de la cadena

$$a_j^L = \sigma(z_j^L)$$

$$C = \frac{1}{2} \sum_j (y_j - a_j^L)^2 \quad \rightarrow \quad \frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{bmatrix}$$

Backpropagation

Error de una capa l a partir de la siguiente ($l+1$)

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Propaga el error a capas anteriores

$$\delta_k^l = \frac{\partial C}{\partial z_k^l} = \sum_j \frac{\partial C}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_k^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_k^l} = \sum_j \delta_j^{l+1} w_{jk}^{l+1} \sigma'(z_k^l)$$

$$z_j^{l+1} = \sum_{k'} w_{jk'}^{l+1} a_{k'}^l + b_j^{l+1} = \sum_{k'} w_{jk'}^{l+1} \sigma(z_{k'}^l) + b_j^{l+1}$$

Backpropagation

Derivada parcial respecto a los biases b

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Coincide con el error de la correspondiente neurona

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l$$

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

Backpropagation

Derivada parcial respecto a los pesos w

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

$$z_j^l = \sum_{k'} w_{jk}^l a_{k'}^{l-1} + b_j^l$$

Algoritmo *backpropagation*

Entrada:

Datos de entrada: x

Matrices de pesos de cada capa: $w = [W^1, W^2, \dots, W^L]$

Tuplas de *biases* de cada capa: $b = [b^1, b^2, \dots, b^L]$

Algoritmo BackPropagation

$a^0 \leftarrow x$
Para cada capa $l \in [1, \dots, L]$
 $z^l = W^l a^{l-1} + b^l \quad a^l = \sigma(z^l)$

Alimenta la red hacia adelante

$\delta^L \leftarrow (a^L - y) \odot \sigma'(z^L)$
Para cada capa $l \in [L-1, \dots, 1]$
 $\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

Propaga el error hacia atrás

Devolver $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l, \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$

Gradiente de la función C

Descenso por gradiente estocástico

Utilizar **todos los ejemplos** de entrenamiento para calcular el gradiente resulta demasiado lento

$$\nabla C = \frac{1}{n} \sum_{i=1}^n \nabla C_{x_i}$$

Podemos estimarlo a partir de una **muestra** de m datos de entrenamiento seleccionados aleatoriamente: **mini-batch**

$$X_1, X_2, \dots, X_m \quad \nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j}$$

Actualización de los pesos con el *mini-batch*

Actualizamos los pesos a partir de la **suma de todas las entradas** del *mini-batch*

$$w_{jk}^l \leftarrow w_{jk}^l - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{X_i}}{\partial w_{jk}^l} \quad b_j^l \leftarrow b_j^l - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{X_i}}{\partial b_j^l}$$

Podemos almacenar el *mini-batch* en una **matriz X**

- Cada columna ($1 \dots m$) será un ejemplo de entrada X_i
- Se puede calcular la actualización de los pesos en una única operación matricial
- Las activaciones de las capas pasan a ser matrices

$$a^0 \leftarrow X$$

Tamaño del *mini-batch*

Alto

Tardamos más en realizar cada actualización de la red

Bajo

Mayor error en la estimación del gradiente

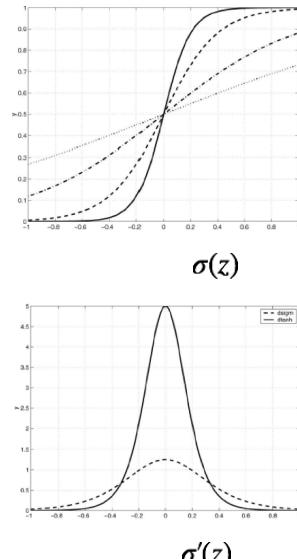
Online learning

- Mini-batch de 1 elemento
- Actualiza rápidamente los pesos
- Considerable error en la estimación del gradiente
 - Tras varias iteraciones, el “rumbo” se corrige
- No aprovecha librerías matriciales optimizadas para el hardware

Velocidad de aprendizaje

¿De qué depende?

$$\begin{aligned}
 w_{jk}^l &\leftarrow w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l} \\
 b_j^l &\leftarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l} \\
 \frac{\partial C}{\partial w_{jk}^l} &= a_k^{l-1} \delta_j^l \\
 \frac{\partial C}{\partial b_j^l} &= \delta_j^l \\
 \delta^l &= ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \\
 \delta^L &= (a^L - y) \odot \sigma'(z^L)
 \end{aligned}$$



Tema 9. Redes Neuronales

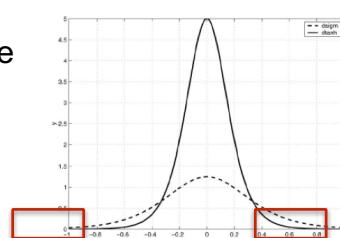
43

Consideraciones

Si la entrada ponderada de la neurona anterior es muy baja o muy alta, la derivada de la sigmoidea será cercana a 0

El gradiente será bajo

Esto hará que aprenda más lentamente



Se dice que la neurona está **saturada**

Se puede solucionar usando una función de coste alternativa

Por ejemplo, *cross-entropy*

Tema 9. Redes Neuronales

44

Ajuste de la red

- Es importante ajustar correctamente **diferentes parámetros**
- **Tasa de aprendizaje**
 - Demasiado alta: No converge (*overshooting*)
 - Demasiado baja: Aprendizaje lento
- **Número de épocas**
- **Tamaño de las capas ocultas**
- **Tamaño del minibatch**
- ¿La **base de datos** de entrenamiento es adecuada?

Bibliografía

Escolano et al. Inteligencia Artificial. Thomson-Paraninfo
2003. Capítulo 4.

Mitchell, Machine Learning. McGraw Hill, Computer
Science Series. 1997

Reed, Marks, Neural Smithing. MIT Press, CA Mass
1999

Neural Networks and Deep Learning. Libro digital:

<http://neuralnetworksanddeeplearning.com/index.html>

Neural Networks. Canal de YouTube 3blue1brown

<https://youtu.be/aircAruvnKk>