Redes Neuronales clásicas

Índice

Referencias

- Capítulo 10 de *Hands-On Machine Learning with Scikit-Learn and TensorFlow*
- Capítulo 11 de *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

Tiempo estimado = 4 sesiones de 2 horas (2 semanas)

1. Componentes básicos de redes neuronales

Las redes neuronales son un modelo computacional inspirado en cómo se transmiten los impulsos nerviosos entre las células del cerebro.

Para introducir los conceptos esenciales comenzaremos estudiando como se resuelve un problema de clasificación con una única neurona, y las limitaciones que se presentan. Para resolver estas limitaciones se introducirán los circuitos o redes neuronales. Una vez tengamos las herramientas básicas podremos extender su utilidad a problemas de regresión.

Todo problema de clasificación que se aborde con técnicas de aprendizaje automático (*machine learning* o ML) se puede plantear en los siguientes términos:

- Cada ejemplo se describe mediante una lista o vector de atributos o características, que pueden ser continuos o discretos. Todos los vectores de atributos tienen el mismo tamaño para ejemplos de un mismo problema.
- Cada ejemplo lleva asociada al menos una etiqueta.

  Si lleva más de una etiqueta se trata de un problema de *multiclasificación*.
- Al conjunto de ejemplos y etiquetas asociadas proporcionados para que aprenda el sistema se denominan *Conjunto de Entrenaminento*.
- El conjunto de entrenamiento no se debe usar completamente para aprender. Es necesario serparar, al menos, un 10 % para *Validación*.

Con estas directrices introducimos la notación básica que se utilizará durante todo el curso.

- Un conjunto de datos estará compuesto por N ejemplos, todos ellos D-dimensionales.
- Un ejemplo es un vector columna D-dimensional, que por comodidad escribiremos en fila, indicando que está traspuesto con el símbolo $^\intercal$. Así, el ejemplo i-ésimo será:

$$\mathbf{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, \ldots, x_D^{(i)}\right]^\intercal$$

- Un conjunto de N ejemplos es, en general, una matriz $\mathbb{R}^{N\times D}$; es decir que los ejemplos se escriben en filas.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)\intercal} \\ \vdots \\ \mathbf{x}^{(N)\intercal} \end{bmatrix} = \begin{bmatrix} x_1^{(1)}, & x_2^{(1)}, & \ldots & x_D^{(1)} \\ x_1^{(2)}, & x_2^{(2)}, & \ldots & x_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)}, & x_2^{(N)}, & \ldots & x_D^{(N)} \end{bmatrix}$$

- Todos y cada uno de los ejemplos deben pertenecer, al menos, a una de las K clases posibles, representado por un vector fila de K elementos binarios. Cada columna representa una clase. Si su valor es 0 significa que el ejemplo no pertenece a dicha clase, si su valor es 1 entonces sí. Esta representación de las etiquetas se denomina *one-hot*, y sirve igualmente para problemas multiclase.

  En definitiva, las etiquetas asociadas al conjunto $\mathbf{X}$ se pueden representar en la matriz $\mathbf{Y} \in \{0,1\}^{N\times K}$

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}^{(1)} \\ \vdots \\ \mathbf{y}^{(N)} \end{bmatrix} = \begin{bmatrix} y_1^{(1)}, & y_2^{(1)}, & \ldots & y_K^{(1)} \\ y_1^{(2)}, & y_2^{(2)}, & \ldots & y_K^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ y_1^{(N)}, & y_2^{(N)}, & \ldots & y_K^{(N)} \end{bmatrix}$$

En visión artificial los ejemplos son generalmente imágenes, habitualmente en color. En otras palabras el ejemplo i-ésimo $x^{(i)} \in [0, 255]^{H \times W \times C}$, donde $H \times W$ es la resolución de la imagen (altura $\times$ anchura) y C es el número de canales (1 si es en escala de grises, 3 si es RGB, etc.), suponiendo que utilizamos 8 bits para codificar la intensidad de cada pixel en cada canal.

Por tanto, un conjunto de N imágenes será una matriz multidimensional $\mathbf{X} \in [0, 255]^{N \times H \times W \times C}$. A esta estructura de datos, recientemente, se le ha dado el nombre (oficioso) de *tensor*. La Figura 1 muestra un tensor de N imágenes, cada una de ellas representada numéricamente en 3D.

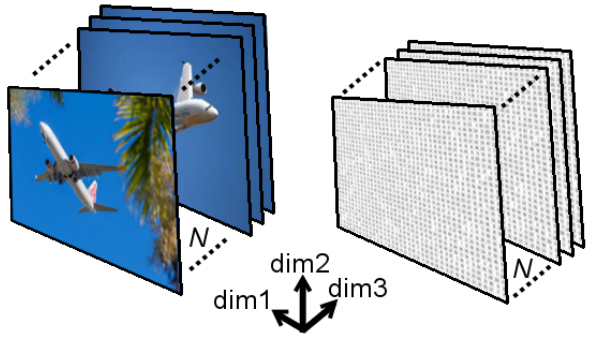


Figura 1: (Izq.) Los conjuntos de ejemplos para problemas de visión artificial son imágenes en color (Der.) Por tanto la matrix $\mathbf{X}$ ya no será 2D. En su lugar se utiliza un tensor 4D.

Por último, la Figura 2 presenta el proceso general de aprendizaje.

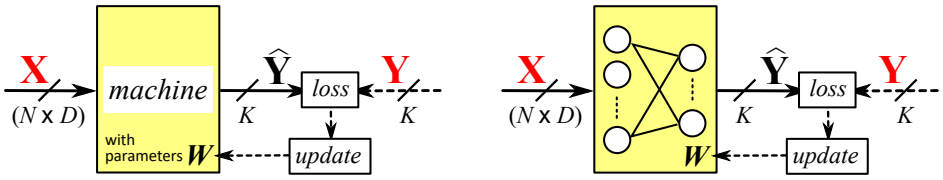


Figura 2: (Izq.) Problema general de clasificación resuelto con ML. (Der.) Cuando la máquina es una red de neuronas el esquema general sigue siendo válido, y los parámetros son los pesos de la red.

En primer lugar debemos elegir el tipo de "máquina" que vamos a entrenar. Sea cual sea, esta tendrá una serie de parámetros que se pueden modificar para lograr este objetivo.

La máquina se alimenta con el conjunto de entrenamiento $\mathbf{X}$ y, en función del valor de los parámetros en ese momento, produce $\hat{\mathbf{Y}}$, una estimación de las etiquetas para cada uno de los ejemplos introducidos.

La estimación o predicción se compara con $\mathbf{Y}$, el valor real de las etiquetas (a veces denominado *ground truth* o GT). Como es de esperar, habrá una discrepancia entre ambas que se evalúa con una función de pérdida.

Finalmente se produce una actualización de los parámetros de la máquina para reducir esa discrepancia y se vuele a alimentar la máquina con el siguiente ejemplo.

Si la máquina es una red neuronal, el procedimiento es exactamente el mismo, sin importar la estructura de la red (densamente conexa, convolucional, autoencoder, recurrente, ...) o el proposito que tenga (clasificación, segmentación, detección, ...).

1.1. Estructura de una neurona artificial

Morfología

- **Dendritas** terminaciones por las que recibe impulsos
- **Núcleo** zona central de la célula, donde se realizan sus funciones
- **Axón** terminación por la que envía impulsos
- **Sinapsis** La unión axón-dendrita cuando se está transmitiendo el impulso

Equivalencia con el modelo computacional

- Las dendritas son las entradas de la neurona
- En el núcleo se combinan las entradas en una suma ponderada.

  Cada peso representa la fuerza de la sinapsis a través de esa entrada.

  La neurona se *activa* (a veces también se dice "se dispara") si el resultado supera un umbral.
- El axón es la salida de la neurona. Si está activada será 1, en caso contrario 0.

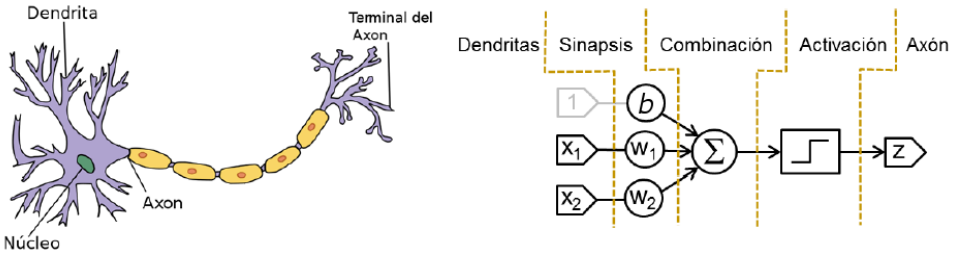


Figura 3: (Izq.) Neurona biológica muy esquemática (Der.) Neurona artificial

En definitiva, una única neurona artificial, *básica*, llamada "Perceptrón" y propuesta por Blatt hacia 1948, es simplemente un modelo lineal umbralizado. Por tanto su utilidad era construir clasificadores lineales, es decir aquellos cuya superficie de separacion entre clases es una recta (para dos atributos), un plano (para tres atributos) o en general un hiperplano (para D atributos).

Por ejemplo, la neurona de la Figura 3 está diseñada para recibir ejemplos 2-dimensionales, es decir $\mathbf{x}^{(i)} = (x_1, x_2)^\intercal$ y producir una salida (escalar) $z = a(w_1 x_1 + w_2 x_2 + b) = a(\mathbf{w}^\intercal \mathbf{x})$, donde a es la función de activación. En la figura se trata de la función escalón con umbral 0.

$$a(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{en otro caso.} \end{cases}$$

1.2. Entrenamiento de una neurona artificial

Como hemos dicho, una única neurona no es más que un clasificador/regresor lineal. Es decir, se trata de una máquina que recibe un ejemplo y produce una estimación de cuál debería ser su salida.

En la Figura 3 la salida de la neurona se llama z porque dicha neurona es un caso particular, para ejemplos bidimensionales y del que no sabemos si depués hay otra neurona conectada a ella. Pero si consideramos que sólo tenemos una neurona, siguiendo la notación introducida, el ejemplo i-ésimo del conjunto sería $\mathbf{x}^{(i)}$ y su salida estimada $\hat{y}^{(i)} = a(\mathbf{x}^\intercal \mathbf{w} + b)$.

Supongamos que la función de activación es la identidad, es decir $a(x) = x$. Entonces podemos encontrar los pesos para un conjunto de datos dado en un sólo paso.

Sean las matrices

$$\mathbf{X}_* = \begin{bmatrix} 1, & x_1^{(1)}, & x_2^{(1)}, & \dots & x_D^{(1)} \\ 1, & x_1^{(2)}, & x_2^{(2)}, & \dots & x_D^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1, & x_1^{(N)}, & x_2^{(N)}, & \dots & x_D^{(N)} \end{bmatrix}, \mathbf{W} = \begin{bmatrix} b, \\ w_1, \\ \vdots \\ w_N \end{bmatrix}, \mathbf{Y} = \begin{bmatrix} y^{(1)}, \\ y^{(2)}, \\ \vdots \\ y^{(N)} \end{bmatrix},$$

tal que $\mathbf{X}_* \cdot \mathbf{W} = \mathbf{Y}$.
Multiplicando por $\mathbf{X}_*^\intercal$ por la izquierda, $\mathbf{X}_*^\intercal \mathbf{X}_* \cdot \mathbf{W} = \mathbf{X}_*^\intercal \mathbf{Y}$.
Y finalmente multiplicando por $(\mathbf{X}_*^\intercal \mathbf{X}_* \cdot \mathbf{W})^{-1}$ por la izquierda, obtenemos

$$(\mathbf{X}_*^\intercal \mathbf{X}_*)^{-1} \mathbf{X}_*^\intercal \mathbf{X}_* \cdot \mathbf{W} = \mathbf{W} = (\mathbf{X}_*^\intercal \mathbf{X}_* \cdot \mathbf{W})^{-1} \mathbf{X}_*^\intercal \mathbf{Y}$$

Sin embargo, en el espíritu de las redes neuronales, no sólo está intentar imitar la morfología de la neurona con un modelo computacional. También se pretende imitar el proceso de aprendizaje; que consiste en reforzar las conexiones sinápticas adecuadamente, según se van mostrando ejemplos al sistema. Además de esta manera se evita tener que invertir la matriz $(\mathbf{X}_*^\intercal \mathbf{X}_* \cdot \mathbf{W})$, que puede tener varios millones de elementos, y por tanto un coste computacional tremendo.

En su lugar, el aprendizaje ejemplo a ejemplo es escalable a tantos ejemplos como se quiera, y cada uno de ellos, o bien un minilote de unos pocos, producirá solamente un pequeño ajuste en los parámetros de la red, por lo que seguramente necesitaremos muchos ejemplos y/o enseñar muchas veces estos ejemplos al sistema. Evidentemente, este proceso puede ser lento, pero la capacidad de computo en paralelo hoy en día ha reducido mucho este problema.

1.3. Limitaciones de una neurona artificial

Puesto que una neurona artificial no es más que un modelo lineal umbralizado, con una sóla podemos construir clasificadores (o regresores) lineales. Esto significa que la superficie de separacion entre clases es una recta (para dos atributos), un plano (para tres atributos) o en general un hiperplano (para D atributos).

Este modelo habitualmente es demasiado simple. Por ejemplo, es incapaz de resolver un problema tan (aparentemente) sencillo como el siguiente.

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

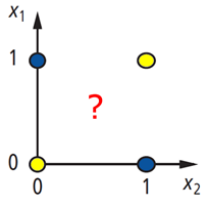
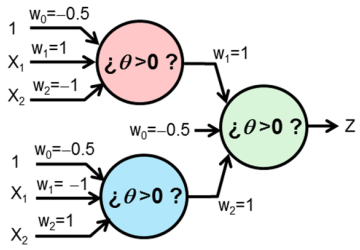


Figura 4: Problema de clasificación no linealmente separable.

1.4. Circuitos neuronales

La verdadera potencia de las neuronas no reside en ellas mismas, sino en los circuitos que forman con otras. Por ejemplo, el problema de la XOR se puede resolver si conectamos 3 neuronas en el circuito de la Figura 5.

De modo que un circuito neuronal puede resolver un problema de clasificación no lineal; pero ¿de qué manera hay que conectar las neuronas entre sí para formar un circuito útil? En este curso aprenderemos varias de las *topologías* o *arquitecturas* más frequentes. En la Figura 6 se puede apreciar que el *zoo* de redes neuronales es bastante extenso.



X1	X2	Y	N#1	Z#1	N#2	Z#2	N#3	Z#3
0	0	0	-0.5	0	-0.5	0	-0.5	0
0	1	1	-1.5	0	0.5	1	0.5	1
1	0	1	0.5	1	-1.5	0	0.5	1
1	1	0	-0.5	0	-0.5	0	-0.5	0

¡ Conseguido !

Figura 5: Construcción de una puerta XOR con neuronas

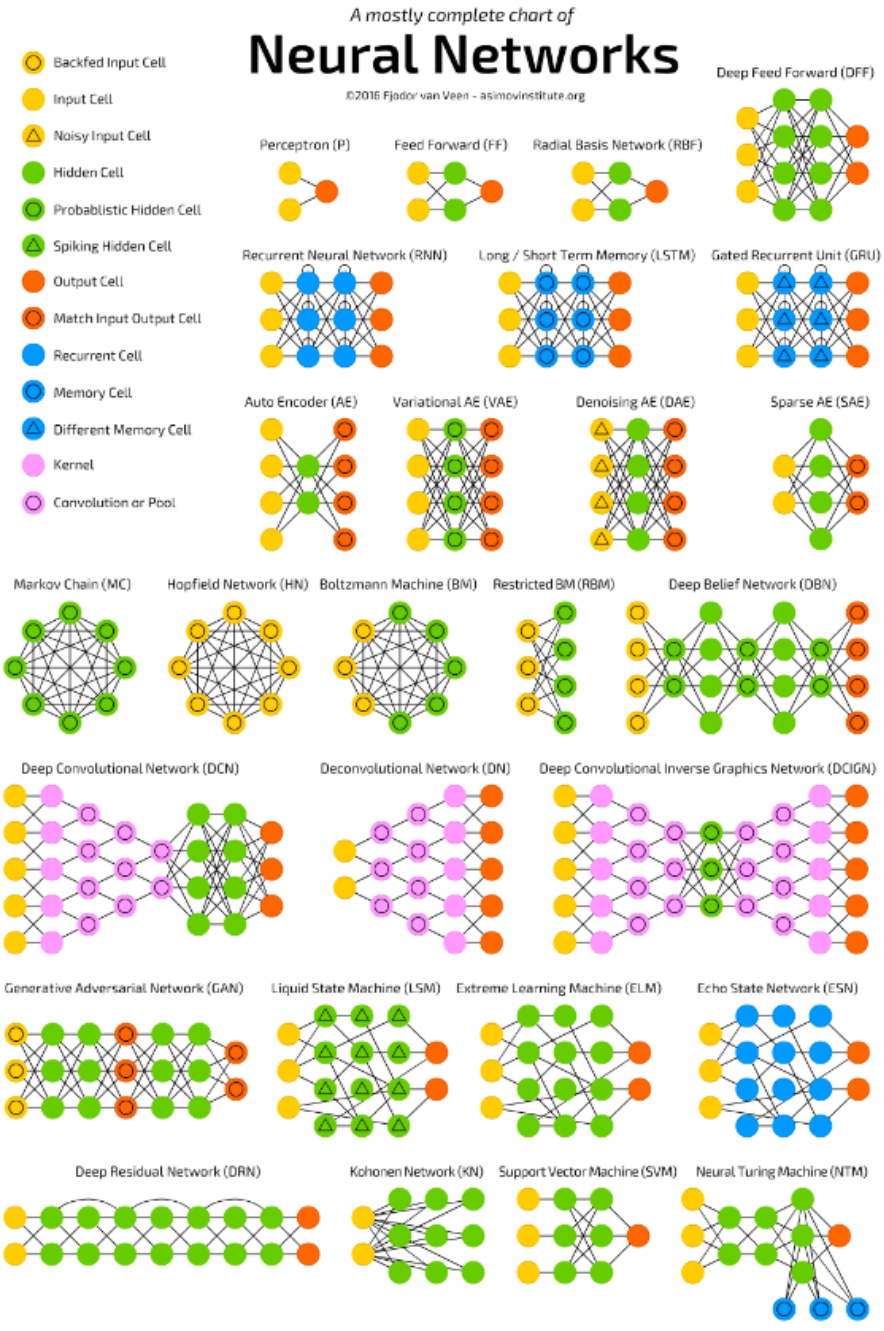


Figura 6: Redes neuronales más importantes. (Fuente: Pinterest)

2. Redes neuronales con capas ocultas

La primera manera, casi intuitiva, de organizar las neuronas en una red es mediante *capas*, de manera que cada neurona de una capa se conecte a todas las neuronas de la capa anterior. Por este motivo recibe el nombre de Red Neuronal Densamente Conexa (*Fully Connected Neural Network*, FCNN), también llamado Perceptron multicapa (*Multilayer perceptron* MLP).

Obviamente, sólo las neuronas de la primera y última capa no pueden cumplir esta regla. Como la primera capa no tiene predecesoras, sus neuronas sólo tienen una dendrita, por la que reciben datos de entrada. Igualmente, como la última capa no tiene sucesoras, la salida de sus neuronas es la salida que produce la FCNN. En definitiva, vista desde fuera, una FCNN tiene una capa visible de entrada de datos y una capa visible de salida de datos. El resto de capas está dentro de la caja negra y se denominan "capas ocultas".

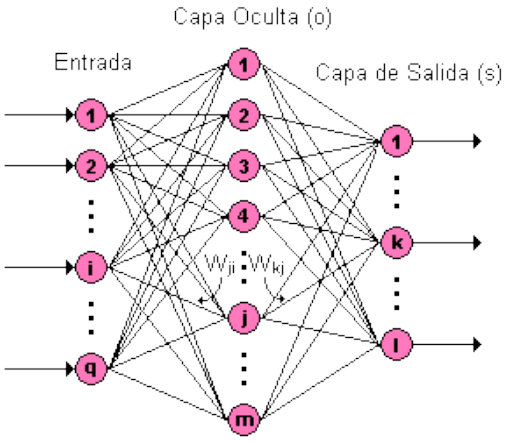


Figura 7: Red neuronal densamente conexa (FCNN) con 1 capa oculta.

En la Figura 7 se muestra una FCNN con 1 capa oculta. En ella además se resalta el hecho de que la neurona i-ésima de la capa de entrada y la neuronal j-ésima de la capa oculta están unidas con un peso w_{ji}. Igualmente, la neurona j-ésima de la capa oculta y la neuronal k-ésima de la capa de salida están unidas con un peso w_{kj}. Como ya hemos dicho, cada unión tiene un peso y cada neurona realiza, primero, la suma ponderada de sus entradas, y luego aplica la función de activación al resultado para producir su salida.

Muchos de los conceptos que se explicarán sobre las FCNN son aplicables a cualquier otra topología de red neuronal, por lo que no se volverá sobre ello en los temas siguientes salvo si hay alguna diferencia importante.

2.1. Diseño

> **Nota:** A lo largo de este curso utilizaremos la biblioteca de Tensorflow y de Keras para implementar, entrenar y probar diferentes tipos de redes neuronales con diferentes propósitos. Por ese motivo, la explicación del diseño de las redes estará muy próxima tanto a la terminología de estas bibliotecas como al problema específico para el que se utilizan.

Una FCNN organiza las neuronas en las siguientes capas consecutivas:

- 1 capa de entrada
- N capas ocultas
- 1 capa de salida con activación lineal o *softmax* según vayamos a construir un regresor o un clasificador, respectivamente.

Llamaremos indistintamente parámetros o pesos de la FCNN a los pesos de las conexiones entre neuronas. El resto de variables que configuran la red, así como otros elementos que modifican el comportamiento serán parámetros de configuración o *hiperparámetros*.

Así, hay un conjunto de hiperparámetros que viene dado por las dimensiones de los ejemplos: el número de neuronas de entrada y de salida.

- El número de neuronas de entrada debe coincidir con la dimensionalidad del conjunto de entrenamiento. Según la notación introducida, habrá D neuronas de entrada.
- El número de neuronas de salida debe coincidir con el número de clases diferentes. Según la representación *one-hot* de la salida y con la notación introducida serán K.

Para configurar el resto de la FCNN se deben tomar varias decisiones:

- ¿Cuántas capas ocultas tendrá?
  → Cuando se utilizan más de 3 o 4 capas ya se considera que es una "red profunda" (*deep neural network*). Es importante tener esto en cuenta a la hora de elegir la función de activación para que el entrenamiento de estas redes sea eficiente.
- ¿Cuántas neuronas habrá en cada capa oculta?
  → El número de neuronas en cada capa está relacionado con la tarea para la cual se diseña la red. En una FCNN suele ser en forma de rombo o de trapecio con la base ancha en la primera capa oculta y la base estrecha en la última.
- ¿Que función de activación tendrá cada neurona de la red?
  → La función de activación juega un papel muy importante en la capacidad de aprendizaje de la red. Además se debe elegir convenientemente en el caso de las neuronas de salida, según la tarea que deba desarrollar la red. La Figura 8 muestra varias de las funciones de activación más utilizadas.

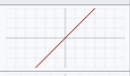
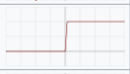
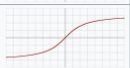
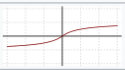
Name	Plot	Equation	Derivative (with respect to x)	Range	Order of continuity
Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$	C^∞
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$	C^{-1}
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$	C^∞
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$	C^∞
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$	$\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$	C^∞
Softsign [7][8]		$f(x) = \frac{x}{1 + \lvert x \rvert}$	$f'(x) = \frac{1}{(1 + \lvert x \rvert)^2}$	$(-1, 1)$	C^1
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$	C^0
Leaky rectified					

Figura 8: Diferentes funciones de activación. (Fuente: Wikipedia)

Para comprender mejor como se deben tomar estas decisiones es necesario presentar cómo funciona la FCNN cuando recibe información (etapa *feed forward*) y como se entrena para que haga predicciones correctas (etapa *back propagation*).

2.2. Etapa *Feed forward*

Supongamos que la red está ya diseñada y que recibe un ejemplo $\mathbf{x} = [x_1, \ldots, x_D]^\intercal$.
La etapa *feed forward* consiste en "alimentar" la FCNN con un ejemplo y hacerlo avanzar a través de las diferentes capas ocultas hasta producir una salida $\hat{y}$. Vamos a verlo en detalle.

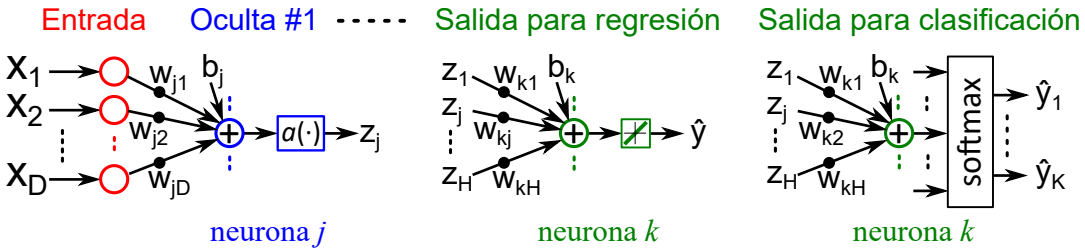


Figura 9: Diferentes capas que podemos encontrar en una FCNN.

- Cada neurona de la capa de entrada recibe un atributo del ejemplo, y no realiza ningún cambio sobre él, simplemente lo transmite hacia delante.
- Cada neurona de la 1ª capa oculta realiza la suma ponderada de todos los valores producidos en la capa anterior, que en este caso era el ejemplo dado, y a continuación aplica la función de activación a ese valor.

  Ej. En la Figura 9, la neurona j-ésima, de la 1ª capa oculta genera la salida

  $$z_j^{[1]} = a\left(\sum_{i=1}^{D} w_{ji}x_i + b_j\right) = a\left(\mathbf{w}_j^\intercal \mathbf{x} + b_j\right) = a\left(\mathbf{x}^\intercal \mathbf{w}_j + b_j\right).$$

- Cada neurona de la 2ª capa oculta realiza la suma ponderada de todos los valores producidos en la capa anterior. Si en la 1ª capa oculta hubo H_1 neuronas, entonces será la suma ponderada de los H_1 valores producidos. A continuación se aplica la función de activación a ese valor.

  Ej. En la Figura 9, la neurona k-ésima, de la 2ª capa oculta recibe como entradas las salidas de la capa anterior. De manera que su salida será

  $$z_k^{[2]} = a\left(\sum_{i=1}^{H_1} w_{ki}z_i^{[1]} + b_k\right) = a\left(\mathbf{w}_k^\intercal \mathbf{z}^{[1]} + b_k\right) = a\left(\mathbf{z}^{[1]\intercal}\mathbf{w}_k + b_k\right),$$

  donde el superíndice entre corchetes indica la capa que produce dicho vector.

  El proceso se repite tantas veces como capas ocultas haya.
- En la capa de salida se realiza la suma ponderada de todos los valores producidos en la capa anterior, es decir la última capa oculta.
  - Si se trata de un problema de regresión para aproximar una función, la capa de salida tendrá una neurona que es quien produce la estimación, igual a la suma ponderada de sus entradas.
  - Si se trata de un problema de clasificación con K clases posibles, entonces el resultado de cada neurona de la capa de salida se transforma de modo que la suma de todas ellas sea igual a 1. Un modo de hacer esto es emplear la función *softmax* en cada una de ellas. Si hay K clases entonces habrá K neuronas de salida que producen respectivamente $z_1^{[out]}, \ldots, z_K^{[out]}$. Entonces la salida estimada k-ésima es

    $$\hat{y}_k = \frac{\exp\left(z_k^{[out]}\right)}{\sum_{k=1}^{K}\exp\left(z_k^{[out]}\right)}$$

    Y la clase estimada será aquella con el $\hat{y}$ mayor.

Lo interesante de esta etapa *feed forward* es que la información fluye siempre hacia delante, desde la capa de entrada hasta la capa de salida. En cada etapa debemos calcular la salida de todas sus neuronas, que será un dato esencial para poder entrenar la red como veremos a continuación.

2.3. Aprendizaje de la red con *Back propagation*

El metodo de aprendizaje de una red neuronal tiene 2 pilares fundamentales:

- función de pérdida, y
- decenso de gradiente.

Para explicarlo nos centraremos en la tarea de clasificación con una FCNN. Los cambios necesarios para hacer regresión, o para otro tipo de red neuronal, habiendo aprendido esto son mínimos.

2.3.1. Función de pérdida

Una FCNN habrá aprendido cuando no cometa errores al responder ante los ejemplos que reciba. Como sabemos la etiqueta de cada ejemplo, sabemos en cuantos nos equivocamos. El objetivo, por tanto, es lograr el menor número de equivocaciones posible, y para ello lo único que podemos cambiar son los pesos de la red. Es decir, modificando los pesos, modificamos el error; y buscamos la *combinación* de pesos óptima, que es aquella que minimiza el error.

La **función de perdida** (*loss*) es la función que devuelve una medida del error cometido para un ejemplo dado. Esta medida no tiene porqué ser la diferencia entre el valor real y el estimado. Por eso es preferible decir "medida del error", o mejor aún "perdida".

Como no tenemos sólo un ejemplo para entrenar, sino muchos, se denomina **funcion de coste** a aquella que agrupa la perdida de varios ejemplos, y es la que se utiliza para corregir los pesos; aunque algunos autores únicamente usan el término "función de pérdida" indistintamente.

En definitiva, sea ℓ una función que devuelve la pérdida calculada a partir del ejemplo de entrada $\mathbf{x}^{(i)}$, sea $\mathbf{y}^{(i)}$ el vector *one-hot* de la etiqueta asociadas y sea $\mathbf{W}$ el conjunto de todos los pesos de la red en ese momento. Entonces la configuración óptima de los pesos buscada, $\mathbf{W}^*$, se puede expresar formalmente como:

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)})$$

En definitiva, estamos ante un problema de optimización, cuyo mínimo se encontrará allí donde la derivada respecto del argumento $\mathbf{W}$ sea cero porque $\mathbf{x}^{(i)}$, $\mathbf{y}^{(i)}$ son los ejemplos que tenemos y no los podemos modificar. Dada la cantidad de pesos que hay en una FCNN, si pretendemos derivar e igualar a cero la función ℓ vamos a enfrentarnos a un problema muy complejo. Si descartamos la *solución analítica* debemos optar por la optimización numérica. Uno de los métodos más habituales es el descenso del gradiente.

2.3.2. El descenso del gradiente

El descenso del gradiente (*gradient descent*, GD) consiste en avanzar por el espacio de soluciones, que en nuestro caso es el espacio N_W dimensional, en la dirección del gradiente de la función ℓ.

Recordar que el gradiente es el vector normal al plano tangente a una superficie. En nuestro caso, dicha superficie es ℓ y su gradiente es

$$\nabla \ell = \left(\frac{\partial \ell}{\partial w_1}, \frac{\partial \ell}{\partial w_2}, \ldots, \frac{\partial \ell}{\partial w_{N_W}} \right)$$

donde w_i es cada uno de los pesos que aparezcan en la FCNN. En la Figura 10 podemos ver el gradiente proyectado (flechas azules) en varios puntos del espacio de soluciones que en este caso es bidimensional ($N_W = 2$), o sea el plano $\overline{w_1, w_2}$.

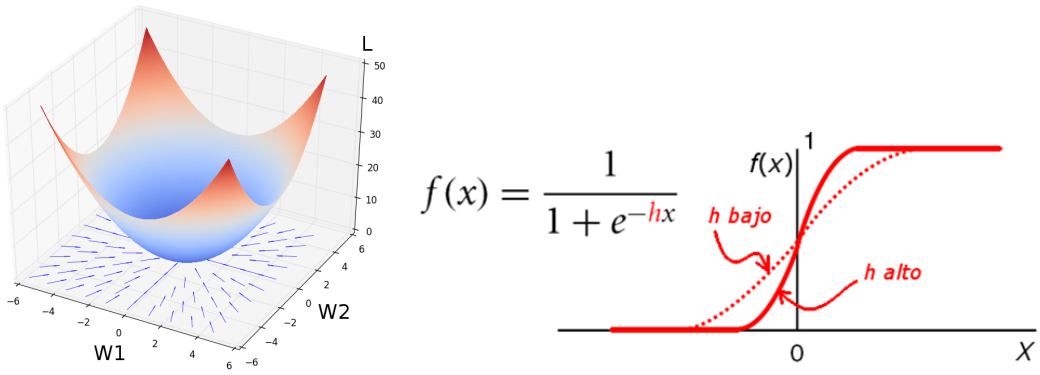


Figura 10: (Izq.)Una superficie ℓ y su gradiente sobre en el plano $\overline{w_1, w_2}$. (Der.) Función sigmoide

Las flechas azules apuntan hacia el mínimo de la función. Esto significa que si avanzamos en esa dirección descenderemos más rápidamente. Una vez hayamos avanzado en esa dirección una cierta distancia hay que volver a comprobar si la dirección de máximo descenso es la misma o es otra. En definitiva, el descenso del gradiente es un algoritmo iterativo, cuyo funcionamiento muy simplificado se puede resumir en los siguientes pasos:

Datos: Conjuntos $\mathbf{X}$ e $\mathbf{Y}$

0. Inicializar los pesos de la FCNN, por ejemplo aleatoriamente alrededor del 0. Esto es equivalente a elegir un punto aleatorio $\mathbf{w} \in \mathbf{W}$.
1. Calcular el gradiente de la función de perdida en $\mathbf{w}$.
2. Elegir la dirección de mayor descenso, es decir aquella dirección donde el gradiente es mayor.
3. El nuevo valor de $\mathbf{w}$ será igual al actual + una cierta cantidad η en la dirección elegida. Por ejemplo, si $\mathbf{w} = (w_1, w_2, \ldots, w_i, w_j, w_k, \ldots,)$ y la dirección elegida fuera j, entonces el nuevo peso sería $\mathbf{w}' = (w_1, w_2, \ldots, w_i, w_j + \eta, w_k, \ldots,)$.
4. Comprobar si $\ell(\mathbf{w}'; \mathbf{X}, \mathbf{Y}) > \ell(\mathbf{w}, \mathbf{X}, \mathbf{Y})$, es decir si hemos alcanzado el mínimo buscado. Si es cierto se detiene el algoritmo; si no, se repite desde 1 con $w = w'$.

Resultado: $w^* = w$

Como en toda optimización numérica, la solución encontrada puede ser un mínimo local; es decir, puede haber una solución mejor. Sin embargo, tiene la ventaja de que no hay que resolver ninguna ecuación, y el gradiente se puede aproximar numéricamente con multitud de herramientas mateméticas. Pero para poder aplicarlas, tal y como se ha explicado, es necesario conocer la función ℓ en cada punto $\mathbf{w}$ para los datos $\mathbf{X}$ e $\mathbf{Y}$. Además hay otro problema. La función de activación que se utilice debe ser derivable. La función de activación escalón (*binary step*, en la Figura 8) no lo es. Historicamente la primera función de activación derivable que se utilizó fue la sigmoide o *soft step*, mostrada en la Figura 10, que dio lugar al método de propagación hacia detrás, en inglés *back propagation*.

La propagación hacía detrás es un método para actualizar cada uno de los pesos de la red neuronal empezando por los de la última capa, siguiendo por los de la penúltima, y así, marcha atrás, hasta los de la primera. Actualizar significa sumar una cierta cantidad al valor del peso para obtener un valor nuevo. Esta cantidad se denomina "término de pérdida, δ" y depende de la capa donde está la neurona que estamos actualizando. Así, para una neurona j hay dos casos posibles:

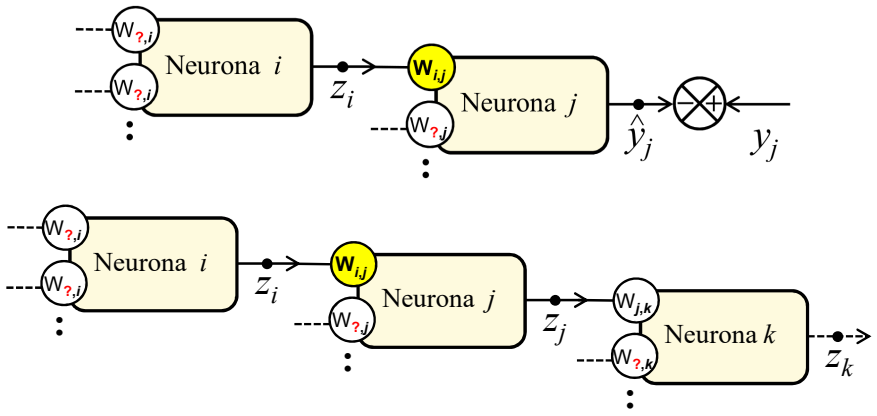


Figura 11: Neuronas involucradas en la retropropagación, caso 1 (arriba) y caso 2 (abajo).

1. Cuando la neurona j pertenece a la capa de salida.

   En este caso, el peso w_{ij}, según la Figura 11, se actualiza con la regla

   $$w'_{ij} = w_{ij} + \eta\delta z_i,$$

   donde δ es el término de pérdida. Si la función de activación es la sigmoide y la función de pérdida es el cuadrado del error, entonces

   $$\delta = y_j(1 - y_j)(\hat{y_j} - y_j).$$

2. Cuando la neurona j pertenece a cualquier capa oculta.

   La regla de actualización del peso w_{ij} es la misma, pero ahora el término de pérdida es

   $$\delta = z_j(1 - z_j)\sum_{\forall k} w_{jk}\delta_k.$$

Los detalles del algoritmo y la justificación matemática de la utilización de la sigmoide se pueden encontrar en cualquier libro de redes neuronales y están implementados en todas las herramientas software. En la asignatura de Matemáticas se darán más detalles sobre ello. Lo más relevante para esta asignatura es comprender los siguientes puntos:

- Para poder utilizar el descenso del gradiente es necesario que las funciones de activación de las neuronas sean derivables.
- Para aplicar el método *back propagation* en primer lugar se debe realizar una etapa de *feed forward*, guardando cada uno de los resultados obtenidos a la salida de cada neurona.
- Después se debe calcular el término de pérdida de cada neurona perteneciente a la capa de salida, y con ello calcular los nuevos pesos de esta capa. El término de pérdida se debe guardar porque se utiliza a continuación.
- Después se debe calcular el término de pérdida de cada neurona perteneciente a la última capa oculta, y con ello calcular los nuevos pesos de esta capa. Así hasta actualizar todos los pesos.
- Las expresiones del término de pérdida dependen de la función de pérdida y de la función de activación.

Hoy en día se disponen de bibliotecas en casi todos los lenguajes de programación que automatizan este proceso, pero no se puede programar lo que no se conoce previamente. Como hemos dicho, los detalles y justificación matemática son temas de otra asignatura. Nosotros vamos a centrarnos en su construcción con Keras y Tensorflow. Y con los elementos vistos hasta ahora ya podemos comenzar a utilizar estas bibliotecas.

3. Mejorando la red

Hemos pasado de una sola neurona a un circuito o red de neuronas llamado FCNN, cuyos parámetros, los pesos de la red, se pueden aprender, es decir encontrar aquellos que optimizan una función de coste para un conjunto de ejemplos de entrenamiento.

Tanto la red como los elementos involucrados en el método de aprendizaje son configurables. La red puede tener más o menos capas, con más o menos neuronas, con unas u otras funciones de activación; hay varias maneras de calcular la pérdida, se pueden mirar los ejemplos uno a uno o por lotes; etc. Todas estas variantes se controlan con los hiperparámetros, es decir los parámetros del algoritmo o del método (no los de la red).

3.1. Construcción

- **Número de capas ocultas.** Típicamente una FCNN para clasificación o regresión, es decir para los tareas *clásicas* de ML, tiene 1, 2 ó 3 capas ocultas.
- **Número de neuronas.** No hay una doctrina sobre ello. Normalmente el número de neuronas de entrada es superior al de salida, por lo que la FCNN suele tener forma de embudo o de rombo.
- **Funciones de activación.** Lo más habitual es utilizar la función sigmoide. Si el número de capas aumenta es preferible elegir la unidad rectificada lineal (ReLU). El motivo se explica en el tema siguiente. La función de la activación de la última capa depende de la tarea que se desea realizar (ver Figura 9).

3.2. Regularización

La regularización es una penalización extra sobre los pesos de la red, que se añade a la función de pérdida. De esta manera se penaliza que los pesos sean grandes, o dicho de otro modo: mantiene los pesos *atados* entre sí para que ninguno crezca demasiado.

3.3. Optimización

- **Función de pérdida** (*loss*). La función de pérdida es seguramente el elemento más importante, puesto que es parte esencial de la función objetivo a optimizar. Para la selección del *loss* es vital tener claro qué tipo de tarea estamos realizando: clasificación o regresión.

  Si estamos clasificando, lo ideal es poder comparar la distribución de clases con la distribución real. La entropía cruzada (*cross entropy*) y la divergencia KL se utilizan para esto precisamente. Si estamos haciendo regresión, entonces es porque las etiquetas pertenecen a un conjunto continuo, y es preferible utilizar el error cuadrático medio u otros similares
- **Ritmo de aprendizaje** (*learning rate*). El *learning rate* controla cuanto ajustamos los pesos de la red respecto de lo que indica el gradiente. En otras palabras, es la constante que acompaña al gradiente en la regla de actualización de pesos. Cuanto más pequeño es, más pequeños son nuestros pasos por la superficie de optimización
- **Impetu** (*momentum*). El *momentum* es una técnica para continuar en la dirección de descenso que llevamos en caso de un cambio en la misma. Esencialmente añade una componente en la dirección anterior, con un cierto porcentaje del módulo, que se suma a la dirección calculada por el gradiente en la iteración actual.

3.4. Ejecución

- **Tamaño del lote** (*batch size*). La red puede recibir los ejemplos uno a uno, o en lotes de un cierto tamaño. En el último caso el ajuste de los pesos es un promedio sobre el resultado del lote. Los lotes suelen ser de 16, 32 ó 64 ejemplos.
- **Número de épocas** (*epochs*). El conjunto de entrenamiento se puede pasar varias veces durante el proceso de aprendizaje. Cada una de esas veces se denomina "època".

4. Ejercicios teóricos

P.1: ¿Qué dimensiones tienen las matrices $\mathbf{X}$ y $\mathbf{Y}$ en los siguientes conjuntos de datos?

a) El MNIST son 60K imágenes de resolución 28×28 en escala de grises. Cada imagen es un número entero entre el 0 y el 9 escrito a mano, escaneado y reducido a ese tamaño.

b) El CIFAR-10 son 60K imágenes a color de resolución 32×32. Las imágenes pueden ser de aviones, coches, pájaros, gatos, ciervos, perros, ranas, caballos, barcos o camiones.

P.2: ¿Qué dimensiones tiene una imagen cualquiera $\mathbf{x}^{(i)}$ de cada uno de los conjuntos de la pregunta anterior?

P.3: ¿Cuántas neuronas de entrada y de salida tendrá una FCNN construida para aprender a clasificar los conjuntos de datos citados en la pregunta 1?

P.4: Dibujar una FCNN para un conjunto de entrenamiento 5-dimensional donde sus datos pertenecen a 3 posibles clases diferentes, con 3 capas ocultas. La primera capa oculta tiene las mismas neuronas que la capa de entrada. La segunda capa oculta tiene 2 más que la anterior. La tercera capa oculta tiene 1 más que la capa de salida.

P.5: ¿Qué función de activación es más apropiada para la última capa de un problema de reconocimiento de 100 clases distintas y excluyentes (problema multiclase)? ¿Y para 100 clases distintas pero no excluyentes (problema multietiqueta)? Ayuda: razonar según el funcionamiento de las siguientes opciones: lineal, relu, sigmoide, tanh o softmax.

P.6: ¿Qué función de pérdida es más apropiada para cada uno de los problemas de la pregunta anterior.

5. Ejercicios Prácticos

- Construir clasificadores para el MNIST o el CIFAR-10 probando diferentes arquitecturas y opciones.
- Para empezar en Keras:
  - First example: a densely-connected network
  - Sequential model methods
- Componentes de Keras para FCNN (menú a la izquierda)
  - *Core layers*:

    dense, activation, input, reshape

- *Merge layers*:

  add, subtract, multiply, average, maximum, concatenate, dot

- *Losses*:

  mean_squared_error, mean_absolute_error,
  binary_crossentropy, categorical_crossentropy

- *Metrics*:

  La métrica se utiliza para evaluar el rendimiento del modelo; es decir es similar a la función de pérdida, pero los resultados de esta evaluación no se utilizan para entrenar el modelo, sólo para mostrar el progreso del aprendizaje.

- *Optimizers*:

  SGD, RMSprop, Adam

- *Activations*:

  sigmoid, tanh, relu, linear, softmax

- *Regularizers*:

  $l1$, $l2$, $l1\_l2$

- Integrar la red neuronal en un proyecto de aprendizaje automático. La Figura 12 muestra las 3 fases: entrenamiento, validación y puesta en producción. La última podemos simularla con un subconjunto de ejemplos que nunca se hayan mostrado al modelo durante el entrenamiento ni la validación.

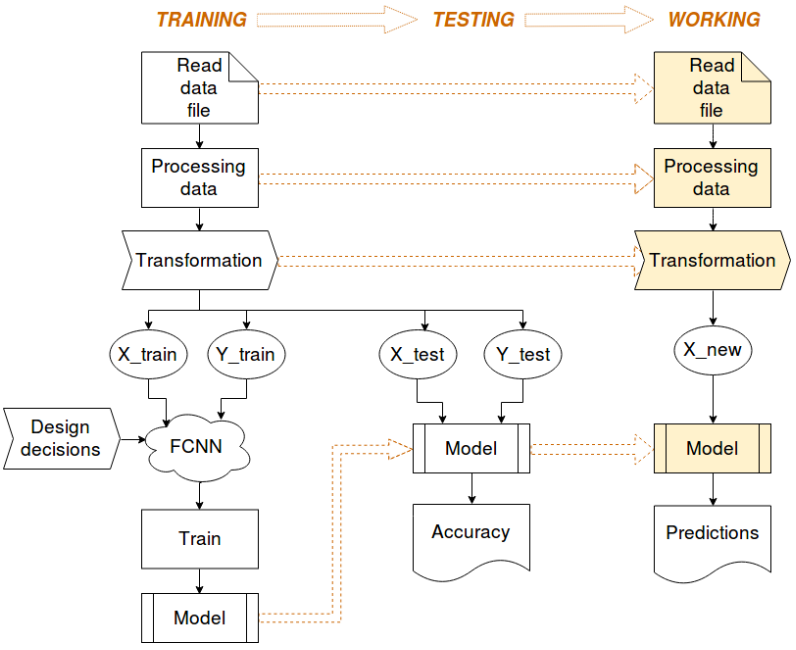


Figura 12: ProyectoML

- En el entrenamiento debemos aprender a leer los datos y procesarlos. Esto habitualmente supone aplicar algún tipo de modificación de los datos iniciales, por ejemplo la conversión de cadenas de texto en valores discretos, así como el escalado de cada característica al intervalo $[0, 1]$ ó $[-1, 1]$ o alternativamente la normalización a media cero y desviación unidad. Estas conversiones y transformaciones se deben conservar para aplicarlas después a los datos nuevos (ya sea en validación o en producción)
- Debemos tomar una serie de decisiones de diseño que predeterminan la arquitectura de la red y el modo en el que aprenderá

- El modelo obtenido se debe probar con datos nuevos para evaluar su rendimiento verdadero.
- Puedes utilizar cualquier conjunto de datos. Lo más inmediato es elegir alguno de los que carga Keras. También puedes utilizar otros conjuntos de datos y leerlos como arrays numpy. Esto es más interesante para trabajos posteriores.

! Es importante dedicar tiempo a este ejercicio ya que una de las prácticas será muy parecida.