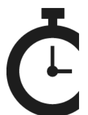
# Gestión de los datos y su uso: aprendizaje autónomo

PID_00247345

Antoni Morell

Tiempo mínimo de dedicación recomendado: 2 horas



Universitat Oberta de Catalunya

*Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.*

# Índice

**Introducción** .......... 5

**1. Aprendizaje supervisado** .......... 6
- 1.1. Regresión lineal .......... 6
- 1.2. Redes neuronales .......... 9
  - 1.2.1. Fase de predicción: clasificación de entradas .......... 10
  - 1.2.2. Fase de entrenamiento: método *backpropagation* .......... 11
- 1.3. Otros métodos .......... 15

**2. Aprendizaje no supervisado** .......... 17
- 2.1. *k-Means* .......... 17
- 2.2. *Principal component analysis* .......... 18
- 2.3. Otros métodos .......... 20

**Bibliografía** .......... 21

# Introducción

El aprendizaje autónomo (en inglés *machine learning*) es un campo de las ciencias de la computación que desarrolla algoritmos para que una máquina sea capaz de aprender a hacer una tarea sin haber sido explícitamente programada para ello. Por ejemplo, las redes neuronales artificiales se pueden interpretar como un sistema capaz de aprender un modelo subyacente en los datos a partir de contrastar unas entradas con unas salidas, por complejo que sea el modelo. Un ejemplo sería tener como entrada una serie de datos médicos de distintos pacientes y, como salida, si padecen o no un determinado tipo de enfermedad. A partir únicamente de estos datos de ejemplo, la red neuronal será capaz de aprender un modelo que relacione entradas con salidas (si es que existe relación alguna), y una vez aprendido el modelo, nos podría servir para hacer predicciones en nuevos pacientes. En este ejemplo, no habríamos programado nuestro algoritmo con unas reglas fruto de un conocimiento experto, sino que habríamos dejado que la máquina aprendiera de los datos.

Hay muchos tipos de algoritmos de *machine learning* (ML), pero una buena forma de clasificarlos es por el tipo de aprendizaje que llevan a cabo. Distinguimos entre **aprendizaje supervisado**, **no supervisado** y **aprendizaje por refuerzo**. En el **aprendizaje supervisado**, dispondremos de una serie de ejemplos de los que conoceremos tanto los parámetros de entrada (que llamaremos características) como la salida, al igual que en el ejemplo anterior. En el **aprendizaje no supervisado**, disponemos de las características o datos de entrada para cada ejemplo, pero se desconoce su salida. Aquí podemos decir que el objetivo está en entender mejor el conjunto de datos, y esto comprende desde algoritmos de *clustering* (que agrupen los datos por similitud) hasta la estimación de su función densidad de probabilidad. Como curiosidad, en el 2012 apareció la noticia de que Google había entrenado una red neuronal profunda con 10 millones de imágenes de vídeos de YouTube sin etiquetar, y consiguió, entre otras cosas, distinguir las imágenes correspondientes a gatos. Esto fue posible gracias a que el sistema construyó su propio modelo de lo que debía ser un gato, y se dio cuenta de que las imágenes de gatos compartían un patrón parecido. Por último, en el **aprendizaje por refuerzo** se plantea un sistema que obtiene una recompensa a partir de una serie de acciones llevadas a cabo, y el objetivo es maximizar esta recompensa. Se diferencia del aprendizaje supervisado en el hecho de que no se conocen explícitamente los pares entrada-salida, sino que hay un conocimiento indirecto del sistema a través de procesos acción-recompensa. En este material, estudiaremos algunos de los algoritmos más representativos dentro del aprendizaje supervisado y no supervisado.

# 1. Aprendizaje supervisado

En los problemas supervisados, como acabamos de comentar, tenemos una serie de datos de entrada cuya salida conocemos. Estos datos de entrada o ejemplos se organizan en una matriz $\boldsymbol{X}$ de $m$ filas y $n$ columnas, donde la *i*-ésima fila $[\boldsymbol{X}]_i = \boldsymbol{x}_i^T$ representa un dato de entrada formado por $n$ características o *features* (las columnas de la matriz). Las salidas a cada uno de los datos de entrada se agrupan en un vector $\boldsymbol{y}$ (o *target*) de dimensión $m$ (número de ejemplos), de modo que los datos $[\boldsymbol{X}]_i = \boldsymbol{x}_i^T$ dan como salida conocida $[\boldsymbol{y}]_i = y_i$. Si no se especifica lo contrario, consideraremos siempre en este material un vector $\boldsymbol{x}$ como un vector columna, y por este motivo hacemos su traspuesta si lo que queremos es un vector fila. Usaremos también la notación $[\boldsymbol{X}]_i$ para especificar la *i*-ésima fila de una matriz. A continuación, veremos dos algoritmos de tipo supervisado, uno enfocado a regresión (*predicción de una variable continua*), y el otro enfocado a clasificación (*predicción dentro de un conjunto posible de valores*).

## 1.1. Regresión lineal

La regresión lineal es uno de los algoritmos más simples dentro de ML, y en realidad está a caballo entre este campo y el análisis estadístico, ya que dada la simplicidad del modelo subyacente, es factible hacer análisis del grado de confianza que tenemos en el modelo, al estilo de lo que hemos visto en el apartado *Análisis estadístico básico* del material «Fundamentos del *big data*: tratamiento de los datos».

En la regresión lineal, plantemos buscar el mejor modelo lineal posible, entendido como aquel que permita reproducir las salidas a partir de las entradas minimizando una determinada métrica de rendimiento, que habitualmente es el error cuadrático medio o MSE. Dado que forzamos un modelo lineal, la predicción $\hat{y}$ que haremos a partir de unos datos de entrada $\boldsymbol{x}$ será $\hat{y} = \boldsymbol{w}^T\boldsymbol{x}$, y tendremos que determinar los coeficientes $\boldsymbol{w}$. Bajo la métrica MSE, por lo tanto, hay que buscar los coeficientes que minimicen:

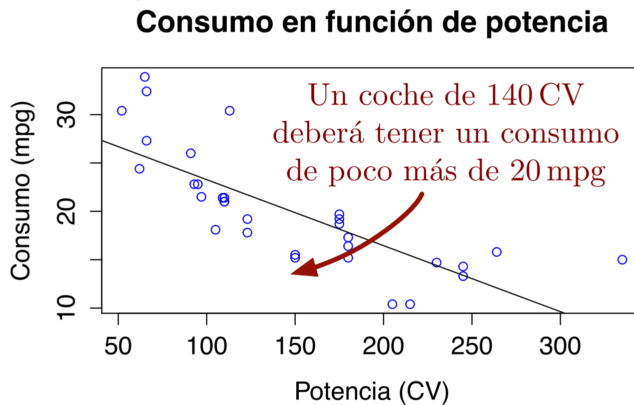
$$\text{MSE} = \frac{1}{m}||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||^2 = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{w}^T\boldsymbol{x}_i - y_i)^2 \quad (1)$$

Esta minimización se puede hacer iterativamente por el método del gradiente [Goodfellow y otros (2016), Sec. 4.3 «Gradient-Based Optimization»], ampliamente usado en ML, o bien en este caso también se puede proceder de forma analítica, forzando $\frac{\partial MSE}{\partial \boldsymbol{w}} = \boldsymbol{0}$, y siendo $\boldsymbol{0}$ un vector columna de ceros (de la misma dimensión que los coeficientes).

Si optamos por la vía analítica, la solución es $\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}\boldsymbol{y}$ (ver desarrollo en [Devore (2011), Sec. 5.1.4]), a lo que también nos referimos como ecuaciones normales, y emplea la seudoinversa de $\boldsymbol{X}$. Si optamos por la vía iterativa, entonces se propone una solución inicial $\boldsymbol{w}_0$ que se va mejorando sucesivamente, siguiendo la dirección opuesta al gradiente (evaluado en la última solución conocida). Es decir, la $n$-ésima actualización se hace a partir de $\boldsymbol{w}_{n-1}$ como $\boldsymbol{w}_n = \boldsymbol{w}_{n-1} - \alpha \frac{\partial MSE}{\partial \boldsymbol{w}}(\boldsymbol{w}_{n-1})$. Cabe destacar la importancia de la elección del parámetro $\alpha$, al que habitualmente se le llama tasa de aprendizaje (*learning rate*), ya que marca lo rápido o lento que el algoritmo encuentra los mejores valores de $\boldsymbol{w}$ o, de manera equivalente, el mejor modelo posible que representa a los datos. No obstante, existe un compromiso en la elección de este valor, pues valores demasiado grandes harán que el algoritmo diverja de la solución óptima, y valores demasiado pequeños ralentizarán demasiado el aprendizaje. Incluso cuando el algoritmo converge, si el valor de $\alpha$ es demasiado grande, la solución obtenida se moverá alrededor de la óptima pero sin llegar nunca a alcanzarla. Por este motivo, en modelos más complejos como las redes neuronales se puede optar por estrategias como la de establecer una tasa de aprendizaje decreciente, con la idea de que el algoritmo aprenda rápido al principio, pero pueda también ajustarse a la solución óptima cuando esté cerca de la misma. Otra de las estrategias que ayudan a los algoritmos de gradiente es la de normalizar los datos (típicamente, quitar la media y normalizar por la desviación estándar cada una de las entrada), pues las curvas de nivel del MSE son más favorables. En el caso de la regresión lineal, pasaríamos de óvalos a algo próximo al círculo.

El modelo de regresión considerado, $\hat{y} = \boldsymbol{w}^T\boldsymbol{x}$, se podría mejorar simplemente añadiendo un término de sesgo, de forma que nuestra predicción pasaría a ser $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$, y no forzamos que para entradas nulas la salida tenga que ser nula. Para añadir este término de sesgo sin tener que modificar nada de lo que hemos dicho hasta ahora, podemos simplemente modificar nuestros datos de entrada, añadiendo un 1 al principio de cada vector $\boldsymbol{x}_i$, de forma que la matriz de datos pasa a ser $\boldsymbol{X} \leftarrow [\boldsymbol{1}\ \boldsymbol{X}]$. Por lo tanto, la nueva matriz de datos tendrá $m$ filas y $n+1$ columnas, siendo la primera columna todo unos. Ahora el vector de coeficientes $\boldsymbol{w}$ incrementará en consecuencia su longitud en una unidad, y el primer coeficiente del vector $\boldsymbol{w}$ hará el papel del sesgo $b$ en nuestro nuevo modelo. Esta estrategia es habitual en muchas de las técnicas de ML. En la figura 1, podemos ver un ejemplo de regresión lineal aplicado al caso de medir el consumo de combustible de un automóvil (en millas por galón) en función de su potencia (en caballos). Son datos extraídos del paquete *datasets* de R, dataframe *mtcars*, que contiene datos de 32 vehículos de los años 73-74. Observad que, una vez conocida la recta de regresión, aprendida de los datos disponibles (puntos en la figura), podemos predecir el consumo de un coche para la potencia que deseemos.

Figura 1. Ejemplo de aplicación de la regresión lineal al cálculo del consumo de un automóvil en función de su potencia




Por otro lado, hasta el momento hemos planteado el cálculo de $\boldsymbol{w}$ como si para ello se usaran todos los datos disponibles. En ML en general no es así, pues es necesario comprobar lo bien o mal que generalizan nuestros algoritmos. En modelos más complejos como las redes neuronales (las veremos a continuación), sería posible dotar de suficiente capacidad al sistema como para «memorizar» la salida correspondiente a cada uno de los datos de entrada. Sin embargo, esto no nos sería de utilidad, puesto que lo más probable es que el sistema no funcionara bien con nuevos datos. En definitiva, no habría sido capaz de aprender el modelo subyacente en los datos, solo memorizarlos. Por este motivo, se divide el conjunto de datos disponibles en 2 o 3 grupos.

Consideremos por ahora 2 grupos, y en el siguiente apartado veremos la necesidad de introducir un tercero. En el caso de 2 grupos, distinguiremos entre *datos de entrenamiento* (*training*) y *datos de test*. La proporción no es matemática, pero suele estar en torno a 70 % para entrenamiento y 30 % para test. Así pues, dividiremos nuestros datos $\boldsymbol{X}, \boldsymbol{y}$ en los grupos $\boldsymbol{X}^{tr}, \boldsymbol{y}^{tr}$ y $\boldsymbol{X}^{test}, \boldsymbol{y}^{test}$ según la proporción indicada. El primero servirá para obtener los coeficientes óptimos $\boldsymbol{w}^*$, mientras que el segundo nos servirá para obtener una medida no viciada por los datos de *training* del MSE, es decir,

$$\mathrm{MSE}^{test} = \frac{1}{m^{test}}||\boldsymbol{X}^{test}\boldsymbol{w}^* - \boldsymbol{y}^{test}||^2 = \frac{1}{m^{test}}\sum_{i=1}^{m^{test}}(\boldsymbol{w}^{*T}\boldsymbol{x}_i^{test} - y_i^{test})^2.$$

Del mismo modo, podemos calcular $\mathrm{MSE}^{tr}$ para medir lo bien que nuestro modelo ajusta los datos, pero la medida de calidad buena es $\mathrm{MSE}^{test}$. Dicho esto, las dos son importantes, ya que nos permiten intuir los posibles problemas de nuestros algoritmos. Si $\mathrm{MSE}^{tr}$ es muy bajo, pero $\mathrm{MSE}^{test}$ es bastante mayor, podemos tener lo que se conoce como un problema de varianza u *overfitting*: nuestro sistema memoriza más que aprende. Si las dos cantidades son altas, podemos tener un problema de sesgo o *underfitting*: nuestro sistema no tiene la suficiente capacidad (o complejidad) para representar bien las salidas a partir de la entradas. Por ejemplo, en el caso anterior ya vemos a partir de la nube de puntos que una relación lineal con la potencia no explica perfectamente el consumo de combustible o quilometraje. Una opción, pues, sería considerar modelos más complejos como las redes neuronales que veremos a continuación. Otra opción sería pensar que la potencia solo no nos vale, y debemos recurrir a más características de los vehículos.

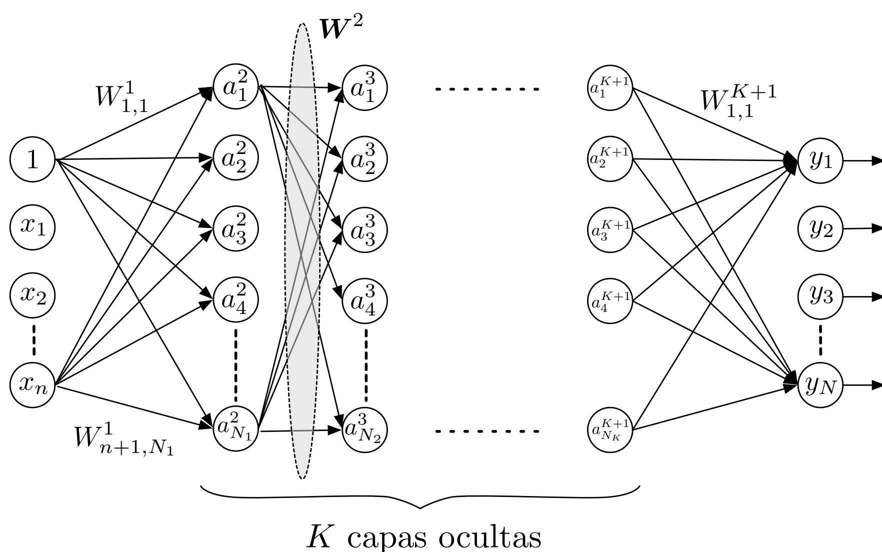
Además, es buena practica aleatorizar los datos, es decir, aplicar una misma permutación aleatoria a las filas de $\boldsymbol{X}$ y de $\boldsymbol{y}$. De esta forma, nos aseguramos tener una representación variopinta en los dos conjuntos. Por ejemplo, supongamos el problema de predecir el consumo de carburante de un vehículo a partir de su cilindrada, tipo de carburante, potencia y número de cilindros. Supongamos que $\boldsymbol{X}$ tiene cierta estructura, por ejemplo los datos de los coches diésel están primero y, además, ordenados por cilindrada; y luego los de gasolina, con la misma ordenación. No parece que tenga mucho sentido entrenar el algoritmo para vehículos diésel y gasolina de baja cilindrada, y luego aplicar los coeficientes $\boldsymbol{w}^*$ hallados a predecir el consumo de los coches de gasolina más potentes. En general, obtendremos un mejor modelo si el algoritmo ve ejemplos de todos los tipos, y por este motivo, no está de más hacer una aleatorización de los datos como paso previo a cualquier algoritmo de ML.

En R tenemos la función `lm`, que nos calcula los coeficientes de un modelo de regresión lineal y, además, nos proporciona información estadística que mide lo bien que se adecua nuestro modelo a los datos. Para predecir, tenemos la función `predict.lm`. A continuación trataremos las redes neuronales, que nos permiten sintetizar modelos tan complejos como queramos, aunque siempre hay una limitación práctica en el tiempo de computación necesario para entrenarlas. Se pueden aplicar a problemas de regresión como el que acabamos de ver, pero seguramente se han hecho más populares por su aplicación a problemas de clasificación, caso en el que nos centraremos.

### 1.2. Redes neuronales

Existen distintos tipos y arquitecturas de redes neuronales. Cuando se componen de más de una capa, hablamos de redes profundas o *deep learning*, concepto que está en auge en los últimos años. En esta parte del módulo, nos centraremos en la estructura más básica de red neuronal (NN), la denominada red neuronal *feed-forward* (FFNN). La denominación NN surge del hecho de que están formadas por la interconexión de unas unidades a las que llamamos neuronas, que responden de forma parecida a lo que harían las neuronas de nuestro cerebro. En otras palabras, se inspiran en modelos biológicos de comunicación entre neuronas a través de sus sinapsis. En la figura 2, podemos ver la estructura de la red. Consta de una primera capa denominada capa de entrada, una serie de $K$ capas intermedias denominadas capas ocultas y una capa de salida denominada como tal. Denominaremos $N_k$ al número de neuronas en la $k$-ésima capa oculta, mientras que $N$ será el número de neuronas en la capa de salida.

Figura 2. Red neuronal *feed-forward*



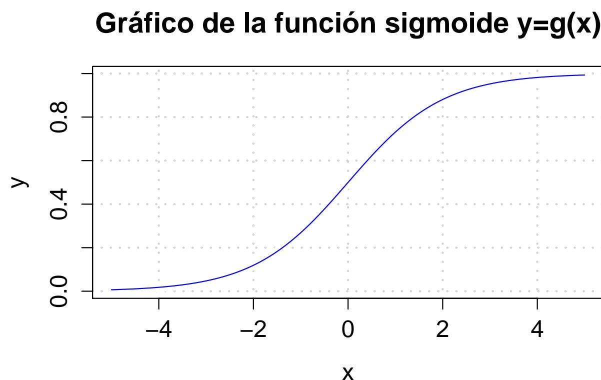

Cada una de las unidades o neuronas en cada capa se interconecta con todas las neuronas de la capa siguiente (de ahí la denominación *feed-forward*) a través de la matriz de interconexión $\boldsymbol{W}^k$. El superíndice $k$ indica la capa de partida. Por ejemplo, $\boldsymbol{W}^1$ describe la matriz de interconexión de la capa de entrada con la primera capa oculta, y $\boldsymbol{W}^{K+1}$ la interconexión de la última capa oculta con la de salida. El elemento $W^k_{i,j}$ es un peso que denota la intensidad de la conexión entre la *i*-ésima neurona de la capa $k$ con la *j*-ésima neurona de la capa $k+1$. La capa de entrada (capa 1) se puede considerar pasiva, ya que cada una de las neuronas únicamente propaga el valor a su entrada, sin modificación alguna (podemos interpretar que aplican la función identidad). Por el contrario, el resto de las capas son activas y su salida es el resultado de aplicar una cierta función de activación al valor de entrada, normalmente funciones no lineales. Para problemas de clasificación entre $N$ clases de salida, lo más habitual es usar la sigmoide en las capas ocultas y la función *softmax* en la capa de salida, como veremos a continuación.

### 1.2.1. Fase de predicción: clasificación de entradas

Antes de ver cómo la red aprende de los datos, supongamos que disponemos de un sistema ya entrenado para una cierta tarea, es decir, conocemos los valores adecuados de los pesos de interconexión $\boldsymbol{W}^k$ de todas sus capas, y observemos cómo se haría la predicción de un nuevo dato $\boldsymbol{x}$. Lo primero sería añadirle un 1 (término de sesgo), es decir, $\boldsymbol{x} \leftarrow \left[1\ \boldsymbol{x}^T\right]^T$, para conformar la capa de entrada. Esto ya se encuentra reflejado en la figura 2. Luego calcularíamos las salidas (equivalentemente activaciones) a cada una de las neuronas de la primera capa oculta (segunda capa de la red), como $a^2_i = g([\boldsymbol{W}^1]_i\, \boldsymbol{x})$, donde la función $g$ sería en este caso la función sigmoide $g(x) = 1/(1+e^{-x})$. Como aclaración, debemos recordar que los superíndices como el 2 en $a^2_i$ identifican la capa de partida, y en ningún caso se refieren a potenciación

(por ejemplo, elevar al cuadrado o cubo). Esta función, que podemos ver en la figura 3, tiene todas sus imágenes entre 0 y 1, siendo $g(0) = 1/2$. El valor de activación puede interpretarse, en general, como el grado de reconocimiento de una característica de los datos de entrada. En la primera capa oculta se hace a partir de una combinación lineal de las entradas, pero a medida que avanzamos en la NN vamos añadiendo no linealidades para posibilitar la detección de patrones complejos en los datos.

Figura 3. Función sigmoide




Una vez calculadas las $N_1$ funciones de activación de la primera capa, tendremos un vector de activaciones $\boldsymbol{a}^2 = [a_1^2, \ldots, a_{N_1}^2]$. Este vector ahora se aplicará a la segunda capa, de modo que la *i*-ésima activación se calculará como $a_i^3 = g([\boldsymbol{W}^2]_i\, \boldsymbol{a}^2)$. El proceso se repite para tantas capas ocultas como tengamos. Finalmente, en la capa de salida hacemos lo mismo, pero esta vez usando la función *softmax*, la cual requiere todas las activaciones obtenidas en la última capa oculta, a las que denominamos $\boldsymbol{a}^K$. En concreto, la salida correspondiente a la *i*-ésima clase se calcula como $y_i = s(\boldsymbol{a}^{K+1}, \boldsymbol{W}^{K+1})_i = \frac{e^{[\boldsymbol{W}^{K+1}]_i\, \boldsymbol{a}^{K+1}}}{\sum_j e^{[\boldsymbol{W}^{K+1}]_j\, \boldsymbol{a}^{K+1}}}$. Observad que en esta última capa se aplica un proceso de normalización (forzamos las salidas a sumar siempre 1). Esto es así para que cada una de las salidas se pueda interpretar como una probabilidad de que los datos pertenezcan a esta clase y, por lo tanto, deben sumar 1. Decidiremos entonces que los datos corresponden a la clase que nos da una probabilidad mayor. A veces se emplea la función sigmoide también a la salida, y la interpretación en ese caso es que la salida *i*-ésima nos proporciona una probabilidad estimada entre 0 y 1 de que los datos sean esa clase, sin sumar 1. No obstante, la decisión final también sería la clase con mayor valor de salida.

### 1.2.2. Fase de entrenamiento: método *backpropagation*

El método de *backpropagation* (BP) es seguramente uno de los hechos más relevantes en aprendizaje automático. Sin embargo, y aunque fue desarrollado ya en los años setenta y ha evolucionado desde entonces, hasta principios de este siglo no ha tomado especial importancia, gracias a la mejora de la capacidad de cálculo de nuestros ordenadores, el principal hándicap en *backpropagation* y redes neuronales. Como curiosidad, esta es la razón por la que ahora se habla del gran invierno que ha habido en inteligencia artificial hasta el año 2000, aproximadamente. También es la razón por la

que otras técnicas, como las *support vector machines*, (SVM) habían tomado mayor relevancia en el pasado.

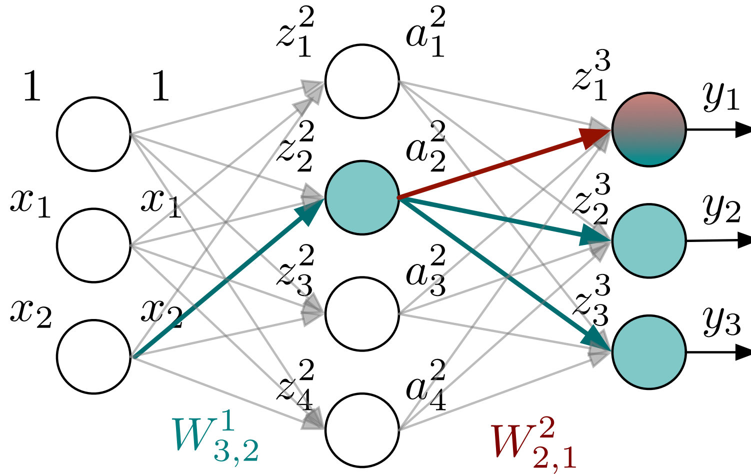
Una vez hecho este apunte histórico, lo primero que hay que decir de BP es que se trata del método que nos permite entrenar de forma eficiente una FFNN. No obstante, el objetivo sigue siendo el mismo que en regresión lineal, es decir, conseguir minimizar una determinada función de coste. Lo más habitual es emplear MSE como en el caso de la regresión lineal, cuando usamos la NN para regresión, y emplear la función de entropía cruzada cuando la utilizamos para clasificación, así que de ahora en adelante consideraremos esta última opción. Como apunte fuera del alcance del módulo, hay que decir que las dos funciones de coste se inspiran en la estimación de máxima verosimilitud [Goodfellow y otros (2016) Goodfellow, Bengio y Courville, Sec. 5.4, Sec. 6.2.2]. En el caso que nos ocupa, teniendo en cuenta los $m^{tr}$ ejemplos de los que dispongamos para entrenar la red y las $N$ posibles clases de salida, la función de coste vale

$$C = -\frac{1}{m^{tr}} \sum_{i=1}^{m^{tr}} \sum_{j=1}^{N} t_{i,j} \log y_{i,j} \qquad (2)$$

donde $\boldsymbol{y}_i = [y_{i,1}, \ldots y_{i,N}]$ es la salida de la NN para el ejemplo i-ésimo, mientras que $\boldsymbol{t}_i = [t_{i,1}, \ldots t_{i,N}]$ es el objetivo o *target* que indica a qué clase en realidad corresponde el i-ésimo dato de entrenamiento. Como aclaración, se debe decir que $\boldsymbol{t}_i$ es un vector con todo ceros y un único 1 en la clase correspondiente, mientras que $\boldsymbol{y}_i$ es un vector de valores entre 0 y 1. Por lo tanto, solo contribuirán a $C$ las salidas $y_{i,j}$ cuyo *target* es $t_{i,j} = 1$. Si la red funcionara perfectamente sacando $y_{i,j} = 1$, entonces $\log y_{i,j}$ sería 0 y tampoco habría contribución. A medida que $y_{i,j}$ se hace pequeño, $-\log y_{i,j}$ y, en consecuencia, $C$ crecen.

La función de coste depende de los pesos $\boldsymbol{W}^1, \ldots, \boldsymbol{W}^{K+1}$ y la estrategia de aprendizaje, al igual que en regresión lineal, pasa por optimizar $C$ y encontrar el valor óptimo de los pesos. La manera de hacerlo es exactamente igual, es decir, usando el método del gradiente o métodos alternativos. La dificultad aquí radica en encontrar los gradientes de los pesos, ya que la función que hay que tratar es bastante más compleja. Aquí es donde entra en juego el método de *backpropagation*. Tiene dos fases: la primera hacia delante (*forward*), en la que simplemente calculamos los valores de salida para todos los ejemplos de entrenamiento, y la segunda hacia atrás (*backward*), en la que usamos todos los valores intermedios calculados en la primera fase y, gracias al uso de la regla de la cadena, calculamos las diferentes derivadas para formar el gradiente de $C$ respecto a los pesos de la red. Con el objetivo de ilustrar el funcionamiento del método, consideraremos la red de juguete de la figura 4, con una sola capa intermedia ($K = 1$) con $N_1 = 4$ neuronas, tres clases de salida ($N = 3$) y 2 características ($n = 2$) en los datos de entrada. Observad que en la figura ya se ha añadido una neurona a la entrada con valor 1 para el término de sesgo. Además, a diferencia de la figura 2, en la que los valores dentro de las neuronas indicaban sus valores de salida, en 4 indicamos tanto entradas como salidas.

Figura 4. Red neuronal de ejemplo para *backpropagation*




Para nuestro ejemplo, consideramos un único vector de entrada $\boldsymbol{x} = [x_1, x_2]^T$ con *target* $\boldsymbol{t} = [t_1, t_2, t_3]^T$ (que, recordemos, solo puede valer $[1, 0, 0]^T$, $[0, 1, 0]^T$ o bien $[0, 0, 1]^T$), y empezamos con la primera fase de *backpropagation*. El primer paso es obtener las entradas a las neuronas de la capa oculta, que denominamos $z_i^2$. Por ejemplo, la entrada a la tercera neurona valdrá $z_3^2 = W_{1,3}^1\, 1 + W_{2,3}^1\, x_1 + W_{3,3}^1\, x_2$. Las salidas a esta capa oculta o activaciones valdrán $a_i^2 = g(z_i^2)$. A continuación, calculamos las entradas a las neuronas de la capa de salida, que denominamos $z_i^3$. Por ejemplo, la entrada a la primera neurona de salida valdrá $z_1^3 = W_{1,1}^2\, a_1^2 + W_{2,1}^2\, a_2^2 + W_{3,1}^2\, a_3^2 + W_{4,1}^2\, a_4^2$. Finalmente, las salidas se calculan como $y_i = s(\boldsymbol{z}^3)_i$. Por ejemplo, $y_1 = \frac{e^{z_1^3}}{e^{z_1^3}+e^{z_2^3}+e^{z_3^3}}$. Recordemos que nuestra función de coste valdrá para este caso $C = -t_1 \log y_1 - t_2 \log y_2 - t_3 \log y_3$, y que para poder optimizar con un método de gradiente, necesitaremos calcular todas las derivadas de tipo $\frac{\partial C}{\partial W_{i,j}^1}$ y $\frac{\partial C}{\partial W_{i,j}^2}$ para los pesos que unen la capa de entrada con la capa oculta y los que unen la capa oculta con la de salida, respectivamente.

Empecemos con el cálculo, por ejemplo, de $\frac{\partial C}{\partial W_{2,1}^2}$. Para ello, aplicaremos la regla de la cadena como sigue:

$$\frac{\partial C}{\partial W_{2,1}^2} = \frac{\partial C}{\partial z_1^3} \cdot \frac{\partial z_1^3}{\partial W_{2,1}^2} = (y_1 - t_1) \cdot a_2^2 \tag{3}$$

Donde hemos aplicado la propiedad de que la derivada de la entropía cruzada (nuestra función de coste) aplicada a la función *softmax* es simplemente $y_i - t_i$ (se puede comprobar fácilmente). Luego, dado que $z_1^3 = W_{1,1}^2\, a_1^2 + W_{2,1}^2\, a_2^2 + W_{3,1}^2\, a_3^2 + W_{4,1}^2\, a_4^2$, es fácil ver que $\frac{\partial z_1^3}{\partial W_{2,1}^2} = a_2^2$. Lo interesante de todo esto es que hemos obtenido el gradiente deseado como una función simple de valores que ya habíamos calculado durante la primera fase del método (en este caso, $y_1$ y $a_2^2$). Observad también que la denominación *backpropagation* encaja con el cálculo de derivadas, siempre yendo hacia atrás (de derecha a izquierda) según el diagrama de la NN.

El cálculo de $\frac{\partial C}{\partial W^1_{i,j}}$ es algo más intrincado, pero la filosofía se mantiene. Tomemos como ejemplo el cálculo de $\frac{\partial C}{\partial W^1_{3,2}}$. En este caso, debemos ir atrás desde la función de coste, pero tenemos más de un camino posible. Si vamos paso a paso, el primero es:

$$\frac{\partial C}{\partial a^2_2} = \frac{\partial z^3_1}{\partial a^2_2} \cdot \frac{\partial C}{\partial z^3_1} + \frac{\partial z^3_2}{\partial a^2_2} \frac{\partial C}{\partial z^3_2} + \frac{\partial z^3_3}{\partial a^2_2} \frac{\partial C}{\partial z^3_3} = (y_1 - t_1) \cdot W^2_{2,1} + (y_2 - t_2) \cdot W^2_{2,2} + (y_3 - t_3) \cdot W^2_{2,3} \quad (4)$$

Donde los términos de tipo $\frac{\partial C}{\partial z^3_i}$ ya los hemos calculado para los gradientes entre capa oculta y capa de salida, y valen $y_i - t_i$. El término $\frac{\partial z^3_1}{\partial a^2_2}$ vale $W^2_{2,1}$ puesto que, como hemos visto, $z^3_1 = W^2_{1,1}\, a^2_1 + W^2_{2,1}\, a^2_2 + W^2_{3,1}\, a^2_3 + W^2_{4,1}\, a^2_4$. De forma parecida, calculamos $\frac{\partial z^3_2}{\partial a^2_2}$ y $\frac{\partial z^3_3}{\partial a^2_2}$. Dando un paso más atrás en la NN, ya podemos calcular $\frac{\partial C}{\partial W^1_{3,2}}$ aplicando de nuevo la regla de la cadena, es decir:

$$\frac{\partial C}{\partial W^1_{3,2}} = \frac{\partial a^2_2}{\partial W^1_{3,2}} \cdot \frac{\partial C}{\partial a^2_2} = \frac{\partial C}{\partial a^2_2} \cdot a^2_2 \cdot (1 - a^2_2) \cdot x_2 \quad (5)$$

Para el cálculo del término $\frac{\partial a^2_2}{\partial W^1_{3,2}}$, consideramos $\frac{\partial a^2_2}{\partial W^1_{3,2}} = \frac{\partial a^2_2}{\partial z^2_2} \cdot \frac{\partial z^2_2}{\partial W^1_{3,2}}$. La primera parte, $\frac{\partial a^2_2}{\partial z^2_2}$, corresponde a la derivada de la función sigmoide simplemente. Es fácil comprobar a partir de su definición que $g'(y) = y \cdot (1 - y)$, así que aplicando este resultado tenemos $\frac{\partial a^2_2}{\partial z^2_2} = a^2_2 \cdot (1 - a^2_2)$. La segunda parte la podemos obtener fácilmente si observamos en la figura que $z^2_2 = 1\, W^1_{1,2} + x_1\, W^1_{2,2} + x_2\, W^1_{3,2}$, así que $\frac{\partial z^2_2}{\partial W^1_{3,2}} = x_2$. Juntándolo todo, obtenemos el resultado de (5) y, como en el caso anterior, los gradientes dependen de valores que hemos obtenido en la primera fase (hacia delante) del método.

Hasta ahora, hemos considerado que solo teníamos un ejemplo para entrenar, es decir, $m^{tr} = 1$. No obstante, cualquier algoritmo de ML necesitará un cierto número de ejemplos (en general, cuantos más, mejor, aunque puede llegar un punto en el que más datos no nos sirvan ya para aprender).

Existen entonces varias estrategias para calcular los gradientes, que se conocen como:

1) *full-batch*;
2) *stochastic gradient descent* (*SGD*) y
3) *mini-batch*.

En el primer caso, hacemos quizá lo más lógico, que es calcular primero los gradientes para todos los ejemplos de *training*, luego hacemos un promedio y posteriormente actualizamos los pesos, y así hasta converger. En el segundo caso, se calculan los gradientes para un solo ejemplo y con esto ya se actualizan los pesos. Luego se pasa al siguiente ejemplo, y así sucesivamente. La tercera posibilidad está a medio camino entre los dos, y consiste en dividir los datos de *training* en paquetes (los *batch*). Para cada paquete se calcula un gradiente promedio, se actualizan pesos y luego se pasa al siguiente paquete. En términos generales, la alternativa *full-batch* es la que mejor estima los gradientes, mientras que SGD puede ser útil para aprender de un flujo de

datos que nos va llegando (y que posiblemente no almacenaremos). La mayoría de los paquetes software hoy día trabajan con la estrategia *minibatch*, ya que es la que suele funcionar mejor en un ámbito de computación (para adaptarse a los recursos disponibles, por ejemplo memoria, y ofrecer el menor tiempo de computación posible).

**Regularización**

El concepto de regularización es muy importante a la hora de introducir una FFNN (y otros algoritmos, también la regresión lineal), ya que nos permite ajustar el compromiso entre sesgo y varianza. Se podría decir que la regularización nos permite frenar la capacidad de aprendizaje de un algoritmo, y será de aplicación cuando tengamos un problema de varianza (por ejemplo, porque la capacidad de nuestra NN es demasiado grande y acaba pudiendo memorizar los datos). Existen varias técnicas de regularización [Goodfellow y otros (2016) Goodfellow, Bengio y Courville, cap. 7], y aquí explicaremos una de las más simples y habituales, la penalización por la norma al cuadrado. Consiste simplemente en modificar ligeramente nuestra función de coste, añadiendo la norma al cuadrado de los pesos, es decir:

$$C^{reg} = C + \frac{\lambda}{2} \sum_{i,j,k} W_{i,j}^2 \qquad (6)$$

De tal forma que los gradientes pasan a ser $\frac{\partial C^{reg}}{\partial W_{i,j}^k} = \frac{\partial C}{\partial W_{i,j}^k} + \lambda\, W_{i,j}^k$, con un parámetro de regularización $\lambda \geq 0$ que hay que ajustar. Se penalizan entonces los pesos grandes en la NN (más cuanto mayor sea $\lambda$), haciendo que no crezcan tanto como lo harían si no hubiese regularización. En un ámbito práctico, se hace la red un poco más insensible a los datos, de modo que se evita que saturen ciertas zonas de la red, lo que le dificulta ajustarse a los datos de entrenamiento, es decir, memorizar. De esta forma, se reduce el *overfitting*. Es importante tener en cuenta que la función de coste $C^{reg}$ se utilizará solo a efectos de entrenamiento, y que para medir el rendimiento de la NN, tanto en *training* como en test, se seguirá empleando $C$.

Para finalizar, diremos que en R tenemos, entre otras, la función `neuralnet` del paquete con el mismo nombre que trabaja con NN, pero sobre todo en *deep learning* es más habitual usar otros lenguajes de programación como Python, en el que tenemos buenas librerías como `Theano` o `TensorFlow`, este último desarrollado por Google.

### 1.3. Otros métodos

Dentro del aprendizaje supervisado, existen otras arquitecturas que en los últimos años han adquirido bastante popularidad y que se enmarcan dentro de lo que llamamos aprendizaje profundo. Como ya hemos comentado, hablamos de aprendizaje profundo o *deep learning* cuando tenemos NN con varias capas ocultas bajo la idea de que sucesivas capas son capaces de sintetizar mejores características (a partir de los datos) que los propios datos. Los algoritmos de *deep learning* han tenido bastante éxito en dos campos de aplicación que son el procesado de imágenes y el tratamiento de se-

cuencias de lenguaje (texto o habla). En el primero, han funcionado bastante bien las *redes neuronales convolucionales* [Goodfellow y otros (2016) Goodfellow, Bengio y Courville, cap. 9], que mezclan los conceptos de red neuronal y filtrado discreto. En concreto, se aplican filtros digitales cuyos coeficientes hay que determinar de la misma forma que se determinan los pesos de la NN. El hecho de convolucionar estos filtros con los datos de entrada (el filtro va pasando por la imagen) permite especializarse en la detección de patrones relevantes, independientemente de la posición que ocupen en la imagen. En problemas relacionados con secuencias, han tenido éxito las *redes neuronales recursivas y recurrentes* [Goodfellow y otros (2016) Goodfellow, Bengio y Courville, cap. 10], pues su arquitectura permite tener cierta memoria dentro de la NN.

# 2. Aprendizaje no supervisado

En aprendizaje no supervisado tenemos datos sin etiquetar, es decir, no hay una salida asociada. En este caso, el objetivo principal se puede decir que es entender mejor los datos y dependiendo del algoritmo que usemos, estará más enfocado a agrupar datos, sacar características relevantes o modelar los datos. De hecho, uno de los usos de algoritmos no supervisados es el de generar nuevas entradas para alimentar sistemas supervisados. En este material, hablaremos de dos de las técnicas más conocidas y empleadas, el algoritmo *k-Means* y *principal component analysis* (PCA).

## 2.1. *k-Means*

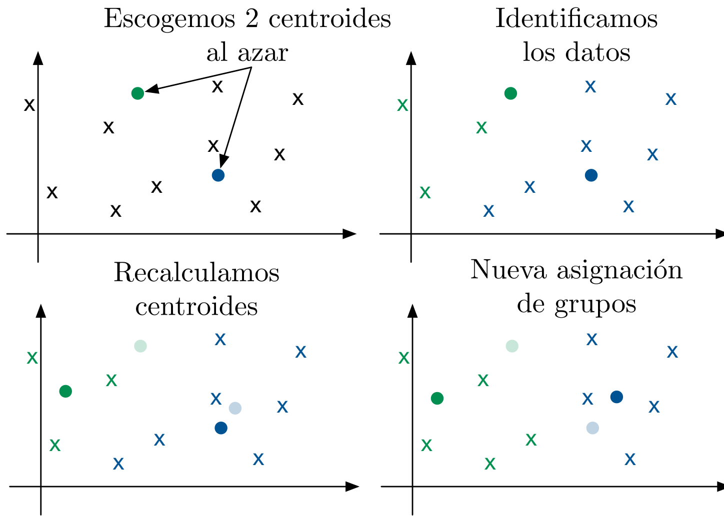
El algortimo *k-Means* es un algoritmo de *clustering* y se especializa en dividir nuestros datos $n$-dimensionales en $K$ grupos ($K$ es un valor dado). El criterio de agrupación es por distancia euclidiana, y lo que perseguimos es, por un lado, etiquetar cada uno de los datos como perteneciente a uno de los grupos y, por otro lado, obtener un centroide de cada grupo, que se puede interpretar como aquel punto que mejor representa al grupo. La idea de fondo es que la suma de distancias de cada uno de los puntos (datos) respecto a su centroide sea lo más pequeña posible.

El algoritmo para obtener los centroides y etiquetar los datos es muy simple, y consta de dos pasos que se van repitiendo hasta que se converge a una solución. Antes de ejecutar estos pasos, se hace una inicialización que consiste, simplemente, en elegir $K$ puntos al azar en el espacio $n$-dimensional de los datos, los cuales serán nuestra primera estimación de los centroides. Los denominamos $\boldsymbol{\mu}_1^0, \ldots, \boldsymbol{\mu}_K^0$ (donde el superíndice indica número iteración, no potenciación). Entonces, el primer paso que hay que ejecutar es recorrer todos los datos y etiquetarlos según su distancia euclidiana a los centroides $\boldsymbol{\mu}_1^k, \ldots, \boldsymbol{\mu}_K^k$. En concreto, cada dato se asocia al centroide que tiene más cercano. El segundo paso consiste en obtener el punto medio de cada grupo de datos acabados de etiquetar, con lo que actualizamos los centroides para la siguiente iteración, es decir, obtenemos $\boldsymbol{\mu}_1^{k+1}, \ldots, \boldsymbol{\mu}_K^{k+1}$. En la figura 5, podemos ver como ejemplo los primeros paso del algoritmo para un conjunto de datos y $K = 2$.

Una vez que el algoritmo ha convergido, conseguimos separar los datos en $K$ grupos de similaridad, a la vez que obtenemos el elemento más representativo del grupo, el centroide, aunque sea artificial. Cabe decir que el resultado puede ser diferente en función de los centroides iniciales elegidos. Aunque el algoritmo es muy simple, puede ser de gran utilidad cuando la dimensionalidad de los datos hace imposible una exploración visual. La mayoría de los programas de análisis de datos o estadística per-

miten agrupar los datos usando *k-Means*. En R, por ejemplo, la función se denomina `kmeans` (del paquete *stats*).

Figura 5. Ilustración del algoritmo *k-Means*




### 2.2. *Principal component analysis*

*Principal component analysis* (PCA) es un método que tiene muchos campos de aplicación (por ejemplo, en el procesado estadístico de señal), además de emplearse también en ML. Lo podemos entender como un método de reducción de dimensionalidad en el que lo que se persigue es representar datos $n$-dimensionales en $k$ dimensiones ($k < n$), siguiendo un criterio objetivo. Bajo esta perspectiva, nos planteamos encontrar una base $\boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k]$ de vectores ortonormales sobre la que proyectar nuestros datos. Supongamos que nuestros datos $\boldsymbol{X}$ ($m$ filas o ejemplos de $n$ características cada uno) son de media nula (si no se cumple, entonces calculamos la media empírica y la sustraemos), y tomemos un dato cualquiera al que llamamos $\boldsymbol{x}$ (lo tomamos, de forma excepcional, como vector fila). Entonces la proyección sobre la base $\boldsymbol{U}$ se calcula como $\boldsymbol{z} = \boldsymbol{x}\,\boldsymbol{U}$, y a partir de la proyección podríamos reconstruir el dato original calculando $\hat{\boldsymbol{x}} = \boldsymbol{z}\,\boldsymbol{U}^T$ (aquí, tanto $\hat{\boldsymbol{x}}$ como $\boldsymbol{z}$ son vectores fila). Tiene sentido, pues, buscar la base de proyección que menor error nos dé en la reconstrucción, lo que corresponde al siguiente problema de optimización:

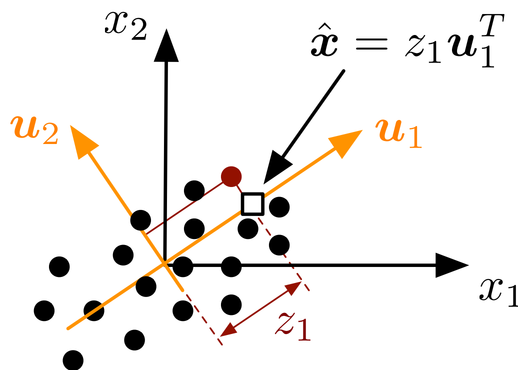
$$\begin{aligned} \min_{\boldsymbol{U}} \quad & \mathrm{E}\left\{||\boldsymbol{x} - \boldsymbol{x}\,\boldsymbol{U}\,\boldsymbol{U}^T||^2\right\} \\ \text{s.t.} \quad & \boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I} \end{aligned} \tag{7}$$

La solución se puede encontrar aplicando el método de Lagrange, de modo que se llega a la condición $\boldsymbol{R_x}\,\boldsymbol{u}_i = \lambda_i\,\boldsymbol{u}_i$ donde $\boldsymbol{R_x} = \mathrm{E}\{\boldsymbol{x}^T\boldsymbol{x}\}$, es decir, la matriz de covarianza de $\boldsymbol{x}$, que en un caso práctico podremos estimar a partir de $\boldsymbol{X}$ como $\hat{\boldsymbol{R}}_{\boldsymbol{x}} = \frac{1}{m-1}\boldsymbol{X}^T\boldsymbol{X}$. La condición a la que hemos llegado nos dice que los vectores

que deben formar la base de proyección $\boldsymbol{U}$ tienen que ser los autovectores de la matriz de covarianza de los datos. Más específicamente, ordenaremos los autovectores según sus autovalores de mayor a menor, y tomaremos los primeros $k$ autovectores, ya que como veremos ahora serán los que mejor representarán nuestros datos. De hecho, el error cuadrático medio de reconstrucción vale $1 - \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{n} \lambda_i}$.

En la figura 6, vemos un ejemplo de lo que está haciendo PCA, que se puede resumir en explotar la correlación existente en los datos para encontrar los vectores de proyección que mejor explican la varianza en los datos. En la figura, vemos cómo el vector $\boldsymbol{u}_1$ (asociado al autovalor más grande) es el que mejor explota esta correlación, ya que la proyección de cada punto sobre su autovector, es decir, $\boldsymbol{u}_1$, es la que nos permite representar los datos con un único número en vez de dos (la proyección $z_1$) con menor error de reconstrucción. Además, podemos intuir que las proyecciones $z_1, z_2$ no exhiben correlación alguna entre ellas [Goodfellow y otros (2016) Goodfellow, Bengio y Courville, Sec. 5.8.1]. Solo tenemos que observar los puntos de datos respecto a los ejes $\boldsymbol{u}_1$ y $\boldsymbol{u}_2$, y veremos que se trata de una nube de puntos sin ninguna relación de linealidad común en sus componentes.

Figura 6. *Principal component analysis* (PCA)




PCA se puede entender como un método de reducción de dimensionalidad, pero también como un método que nos proporciona una representación alternativa de los datos. Esta representación alternativa puede servir para mejorar nuestros algoritmos de ML, por ejemplo acelerando el entrenamiento de una NN, si gracias a ello podemos reducir dimensionalidad, perdiendo poca varianza en los datos (es equivalente a decir que el error de reconstrucción es bajo). También podemos emplear PCA en el análisis exploratorio de datos, ya que reducir dimensionalidad nos facilitará la representación visual. Cualquiera de los programas de análisis de datos existentes hoy día nos permitirá trabajar con PCA. En R, por ejemplo, tenemos la función `prcomp` del paquete *stats*. Además, en otros paquetes como caret (con distintos métodos de clasificación y regresión), PCA se ofrece como un método de preprocesado de datos. En cualquier caso, con cualquier librería de álgebra lineal que nos permita calcular la *singular value decomposition* (SVD) obtendremos autovalores y autovectores para poder aplicar PCA.

### 2.3. Otros métodos

Existen otros métodos más complejos y, a su vez, relevantes dentro del análisis no supervisado, como por ejemplo los *autoencoders* [Goodfellow y otros (2016), cap. 14] o las *deep belief networks (DBN)* [Goodfellow y otros (2016), cap. 20]. Los primeros son simplemente NN que intentan sacar a su salida los mismos datos de la entrada. Con esto, conseguimos que las neuronas de las capas intermedias aprendan características más relevantes sobre los datos que los propios datos. Las DBN son estructuras que tratan de generar un modelo probabilístico acorde con nuestros datos, lo que es útil tanto para distinguir grupos de datos con patrones parecidos como para generar datos artificiales que sigan el patrón de un determinado grupo. Esta es la herramienta que usaron en Google para identificar imágenes de gatos de manera no supervisada. En este caso, el algoritmo fue capaz de aprender cómo debía ser el patrón de píxeles que conforman la imagen de un gato (en un ámbito de pdf). Esto permite tanto identificar a gatos en imágenes, como generar nuevas imágenes de gatos artificiales, puesto que, en el fondo, hemos aprendido el modelo de lo que es la imagen de un gato.

## Bibliografía

**Devore, J. L.** (2011). *Probability and Statistics for Engineering and the Sciences*. 8a ed.). Brooks/Cole.
ISBN-13: 978-0-538-73352-6.

**Goodfellow, I.; Bengio, Y.; Courville, A.** (2016). *Deep Learning*. MIT Press.
http://www.deeplearningbook.org.