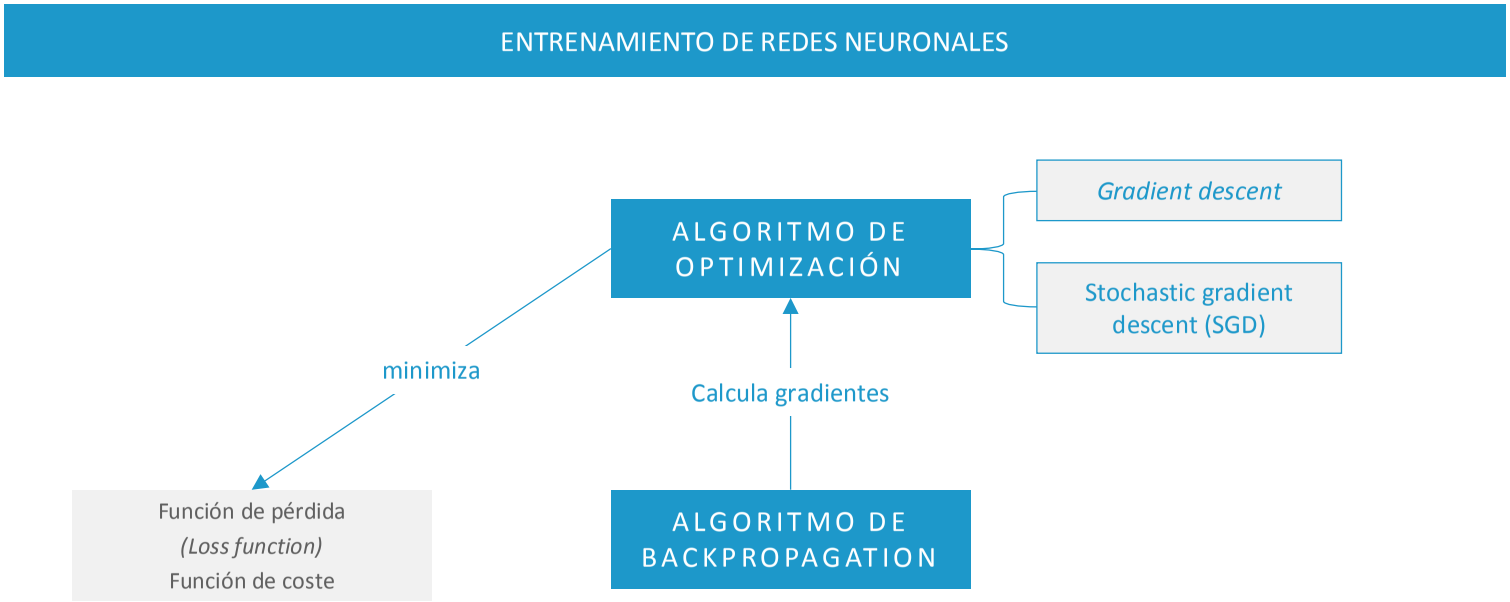
Sistemas Cognitivos Artificiales

Entrenamiento de redes neuronales

Índice

Esquema



Ideas clave

2.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

Trataremos varios conceptos fundamentales en el mundo de las redes neuronales y será el tema con más carga matemática del curso. Es importante comprender los conceptos que se ven aquí: función de coste, entrenamiento como problema de optimización, *gradient descent*, *backpropagation*… y cómo se relacionan entre ellos para dar lugar al entrenamiento de redes neuronales.

El contenido se estructura de la siguiente manera:

- Para empezar, veremos qué es una **función de coste** y un ejemplo de red neuronal utilizado para clasificar imágenes.
- A partir del concepto de función de coste, aprenderemos cómo entrenar una red neuronal a partir del problema de optimización de minimizar dicha función.
- Para ello, estudiaremos el **algoritmo de *gradient descent*** y la versión utilizada en la práctica, ***stochastic gradient descent***.
- Finalmente, veremos cómo calcular los gradientes de una red neuronal de manera eficiente con el **algoritmo de *backpropagation***.

Si bien no es necesario ser capaz de desarrollar de manera completa el algoritmo de *backpropagation* para una red neuronal, es muy importante entender su funcionamiento y ser capaz de desarrollar ejemplos relativamente sencillos como los vistos en clase. Este algoritmo es fundamental para entender conceptos que veremos más adelante durante el curso.

Se recomienda encarecidamente ver los vídeos mostrados en la sección «Lo + recomendado» como soporte para comprender este tema y entender el funcionamiento de las redes neuronales.

2.2. Funciones de coste

Una red neuronal para MNIST

Como hemos indicado, en este tema veremos los fundamentos matemáticos de las redes neuronales y cómo se entrena una red neuronal. Para ello, utilizaremos un problema clásico en el mundo del *machine learning*: MNIST.

MNIST es un dataset de imágenes con números escritos a mano del 1 al 10, siendo el objetivo detectarlos automáticamente.

Si bien ya desde los años 90 existen sistemas basados en redes neuronales entrenadas con este dataset (por ejemplo, para detectar códigos postales en el correo), MNIST sigue siendo utilizado como un *benchmark* para algoritmos de visión por computador. Podría decirse que es una especie de «Hola, mundo» en el mundo del aprendizaje automático.

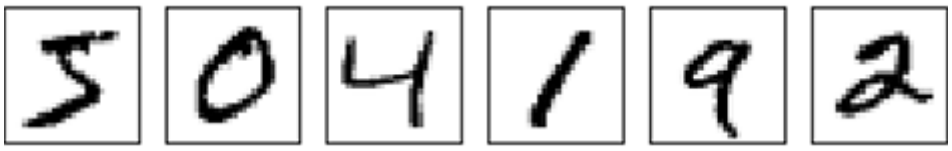


Figura 1. Ejemplo de imágenes en MNIST representando números.
Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Para resolver el problema de clasificar dígitos utilizaremos, como no podía ser de otra manera, una red neuronal.

1. Ya que las imágenes están compuestas de píxeles, alimentaremos a la red con todos los píxeles de la imagen representados en un vector de una dimensión; para ello, se concatenan todas las filas de píxeles de la imagen, una tras otra.

2. Como las imágenes de MNIST son de 28x28 píxeles (muy baja resolución), la *input layer* tendrá 784 neuronas. Cada neurona de esta primera capa representará un píxel, con un valor entre 0 y 1 según la oscuridad del píxel, siendo 0 totalmente blanco y 1 totalmente negro.

3. Para este ejemplo, utilizaremos una sola *hidden layer*, aunque podríamos añadir más si quisiéramos. El número de nodos en esta capa es algo con lo que se puede experimentar, añadiendo o quitando potencia a la red.

4. Finalmente, la *output layer* tendrá 10 nodos, uno por cada posible número a clasificar. Si utilizamos la función de activación *sigmoid* (que vimos en el tema anterior) en cada uno de estos nodos de salida, podemos entender que cada neurona de salida representa la probabilidad de que la imagen sea un número en particular. De este modo, podremos clasificar la imagen en un número con base en el mayor valor obtenido en esta última capa.

En la figura 3 vemos un esquema de esta red neuronal.

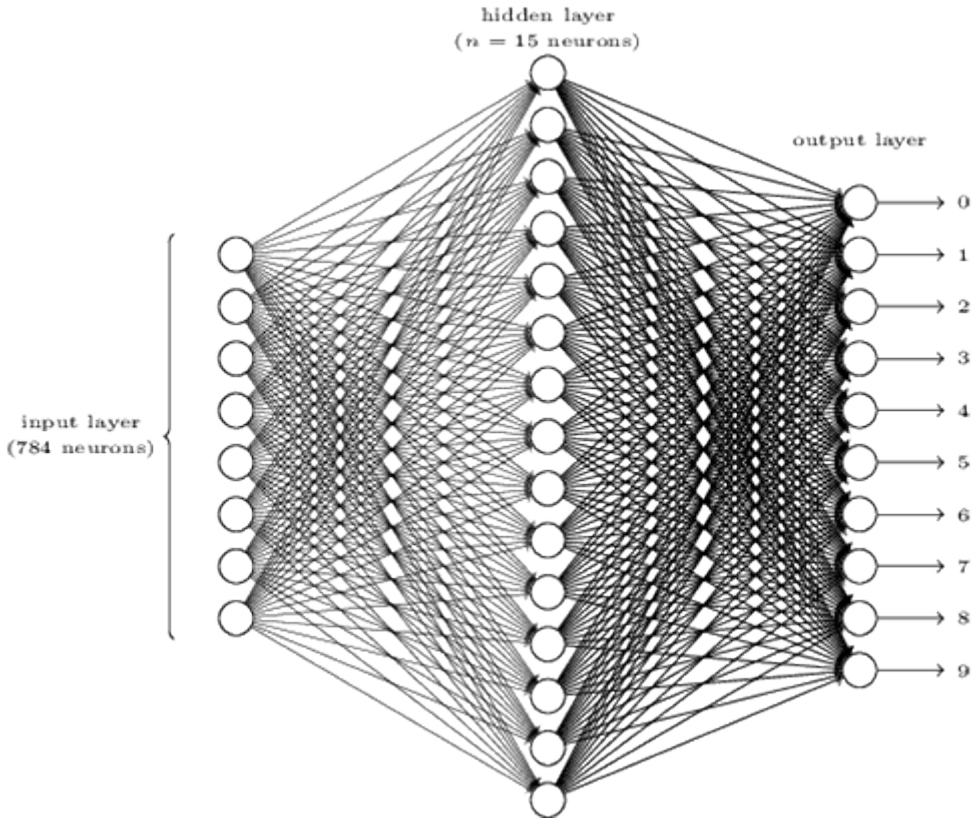


Figura 2. Red neuronal para MNIST.
Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Una pregunta que podríamos hacernos es cuántas neuronas tenemos que tener en la *hidden layer*. Este número es un **hiperparámetro** (*hyperparameter*) de la red y tenemos libertad para elegirlo. Una forma de obtenerlo sería mediante una búsqueda de hiperparámetros, probando varios valores y quedándonos con el que ofrece mejores resultados.

Es importante obtener cierta idea de lo que puede estar pasando en esta capa intermedia. Algo que debería quedar claro es que, al añadir más nodos a la *hidden layer,* estamos añadiendo más potencia a la red, ya que a partir de los píxeles de entrada obtenemos **más representaciones intermedias** de los elementos en una imagen (recordemos, las redes neuronales obtienen conceptos más complejos a partir de conceptos simples). ¿Qué representaciones intermedias están obteniendo estas neuronas a partir de los píxeles? Si bien esto variará cada vez que entrenemos la red neuronal, la red intentará representar de la mejor manera posible con los recursos que tiene el problema a tratar. Por ejemplo, algo que podría estar pasando

es que las neuronas de la *hidden layer* se activen cuando ven ciertos píxeles negros en la entrada para reconocer formas sencillas, como se representa en la figura 4. Así, las neuronas intermedias se activarían al reconocer formas simples, las cuales pueden ser utilizadas a su vez por la última capa para activar las neuronas de salida asociadas a números que están compuestos por esas formas.

Figura 3. Posibles representaciones intermedias en la *hidden layer*.
Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Es importante mencionar que, si bien en este caso podríamos obtener una visualización de lo que está pasando en la capa intermedia, en general las abstracciones que la red neuronal obtiene al ser entrenada pueden escapar completamente a nuestra comprensión. Al fin y al cabo, las redes neuronales son potentes modelos estadísticos capaces de encontrar patrones muy complejos en los datos.

Volviendo a la pregunta de cuántas neuronas necesitamos en la capa intermedia, podemos ya intuir que no hay una respuesta clara. Necesitamos un número suficiente para que la red neuronal obtenga unas representaciones intermedias adecuadas: con pocas neuronas, la red no obtendrá representaciones suficientemente complejas y su precisión se verá afectada. Con más neuronas, llegará un punto en el que estas no ayudarán, ya que la red tiene suficiente capacidad para expresar la variabilidad encontrada en los datos.

Definiendo una función de coste

Empecemos a introducir cierta notación matemática sobre **cuál es nuestro objetivo** al entrenar una red neuronal. Definiremos como:

- x = el *input* o *training example* de la red (en este caso, nuestro vector de 784 píxeles).
- $f(x)$ = el valor deseado de salida de la red, representado como un vector con 10 componentes, uno por cada número.

Por ejemplo, si x representa una imagen con un 3, entonces:

$$f(x) = (0,0,0,1,0,0,0,0,0,0)$$

$f(x)$ representa la función real que queremos imitar, una función que a cada imagen asigna el número representado. Nuestra red neuronal también es una función que a partir de x, de los pesos w y *biases* b obtiene a su vez otro vector con 10 componentes. La definiremos como $a(x, w, b)$ o solo a para simplificar.

Nuestro objetivo es obtener los parámetros w y b de la red neuronal que mejor aproximen la función real $y(x)$ para todos los valores x, esto es, para todos los *training examples* en nuestro dataset. Para cuantificar cómo de buena es esta aproximación, definimos la función de coste (*cost function*):

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

Donde:

- w y b representan todos los parámetros de la red neuronal.
- n es el número de *data points* en nuestro dataset.
- El sumatorio es sobre todas las imágenes del dataset.

Como podemos comprobar, por cada ejemplo estamos obteniendo el cuadrado de la distancia vectorial, un número siempre positivo, entre el valor real y la salida de nuestra red. Para una imagen que nuestra red clasifica de manera correcta como un 0, podríamos tener $a(x) = (0.9, 0.1, 0.1, 0.2, 0.1, 0.1, 0, 0\ ,0, 0)$, que como vector está cerca de $f(x) = (1,0,0,0,0,0,0,0,0,0)$. De este modo, podemos ver que la función de coste es una **suma de valores que miden el error** que nuestra red está obteniendo al clasificar las imágenes.

Si $C(w, b)$ es muy grande, nuestra red no está haciendo un buen trabajo. Por lo tanto, nuestro objetivo al entrenar la red será minimizar este error y acercar en la medida de lo posible $C(w, b)$ a 0. Al final, y como es común en el mundo de *machine learning*, el problema a resolver es realmente un **problema de optimización**. El objetivo es obtener los parámetros w y b que minimicen el valor de C.

Podríamos preguntarnos por qué el error se mide con el cuadrado de la distancia entre vectores. Este tipo de error es, de hecho, muy típico en problemas de aprendizaje automático y se conoce como **MSE (*Mean Squared Error*)**. ¿Por qué no, entonces, minimizar directamente el número de fallos cometidos por la red? La respuesta es que necesitamos una función diferenciable para poder resolver el problema de optimización. Como veremos próximamente, el algoritmo que utilizaremos para minimizar la función, *gradient descent*, necesita una función de coste diferenciable.

Es importante mencionar que el *mean squared error* no es la única manera de medir el error. Hay otras funciones posibles que derivan en diferentes problemas de optimización y, por tanto, en diferentes resultados a la hora de entrenar.

2.3. Entrenamiento con *gradient descent*

Acabamos de ver que entrenar una red neuronal es equivalente al problema de optimización de minimizar la función de coste $C(w, b)$. Esta función es muy compleja y depende de la arquitectura de la red y de los parámetros w y b que conforman esa red. Sin entrar de momento en esta complejidad, vamos a tratar el problema general de cómo minimizar una función, es decir, de **buscar los valores de las variables que proporcionan el valor mínimo**. Esto nos llevará a un algoritmo que podemos usar para entrenar redes neuronales, conocido como *gradient descent*.

Algoritmo *gradient descent*

Supongamos que queremos minimizar una función $C(v)$, donde v puede ser un vector de varias variables. Una forma de minimizar esta función es recurrir al cálculo y tratar de encontrar el mínimo analíticamente calculando la derivada. Sin embargo, esto no va a ser posible para funciones tan complejas y con tantos parámetros como las que representan las redes neuronales. Para hacer las cosas más fáciles de visualizar, supongamos que C es una función de dos variables: v_1 y v_2, como en la figura 5.

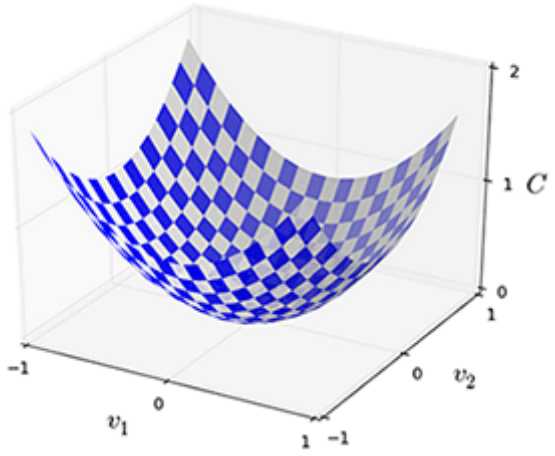


Figura 4. Gráfica simple de una función de dos variables.
Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

La idea del algoritmo a desarrollar es la siguiente: empezaremos en un punto (v_1, v_2) al azar y buscaremos la dirección de máxima pendiente hacia abajo, dando un pequeño paso en esa dirección. Este proceso se realizará repetidamente hasta llegar al mínimo buscado.

Hagamos esto algo más formal. Según el cálculo, sabemos que la variación en C con pequeñas variaciones en v sigue la siguiente aproximación:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Buscamos hacer $\Delta C < 0$, para así minimizar la función. Definimos el vector de variación de v como:

$$\Delta v = (\Delta v_1, \Delta v_2)^T$$

Y el gradiente de C, que define la dirección de mayor inclinación, como:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

Con esto, podemos reescribir la primera ecuación como:

$$\Delta C \approx \nabla C \cdot \Delta v$$

Ecuación 1

Supongamos que elegimos:

$$\Delta v = -\eta \nabla C$$

Donde:

- η es una pequeña constante positiva conocida como ***learning rate***.

Esto significa que:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

Lo cual, como la norma al cuadrado de un vector es siempre un valor positivo, hace que nuestra aproximación sea $\Delta C \leq 0$. Por tanto, si elegimos Δv como arriba, estamos haciendo decrecer el valor de la función, tal y como buscábamos.

Con esto, podemos definir la regla de movimiento para movernos de v a un punto v' donde C tenga un valor menor como:

$$v' = v - \eta \nabla C$$

Esta es la **ecuación de movimiento** o ***update rule*** de *gradient descent*. Esta regla será aplicada de manera repetida, de modo que continuaremos calculando el gradiente de C y actualizando v, haciendo decrecer C hasta llegar a un mínimo. Como podemos ver en la figura 6, la idea es calcular el gradiente y dejarnos caer poco a poco por la pendiente que define el valor negativo de este.

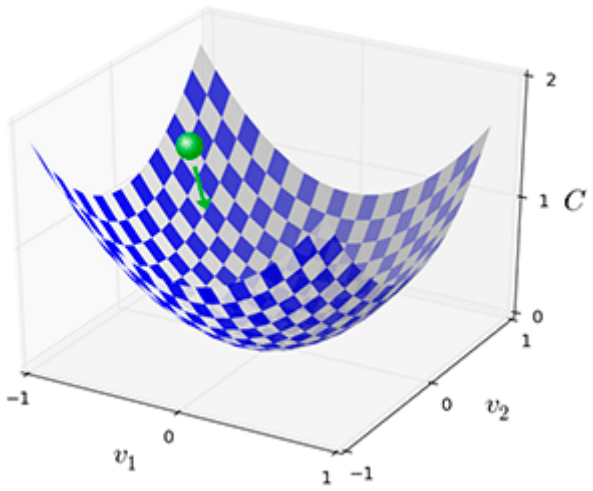


Figura 5. Movimiento de *gradient descent* en busca del mínimo de la función.
Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

No hemos comentado el papel de η, el *learning rate*, en este proceso. Como el nombre indica, este valor refleja en cierta manera la **velocidad** a la que *gradient descent* funcionará. La idea es que η sea un valor lo suficientemente pequeño para que tengamos una buena aproximación en la «Ecuación 1».

- Si elegimos un valor demasiado grande de η, podríamos pasarnos y actualizar v de modo que incluso vayamos a un valor de C mayor.
- Por otro lado, un valor demasiado pequeño de η haría al algoritmo avanzar de manera muy lenta.

La elección de η es por tanto algo complicado, volveremos a tratar el tema en el futuro.

Gradient descent aplicado a redes neuronales

Ahora que conocemos el funcionamiento de *gradient descent* para una función en general, es momento de aplicarlo en nuestro estudio de las redes neuronales. Como sabemos, los parámetros en nuestra función C son los pesos w y los *biases* b, de modo que nuestra posición, en vez de venir dada por un vector v, viene dada por todos los valores w y b de los nodos que componen la red. De este modo, nuestra *update rule* de *gradient descent* para una red neuronal será:

$$w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Para todos y cada uno de los valores w_k y b_l. Sin embargo, notemos un pequeño problema a la hora de entrenar redes neuronales con el algoritmo aquí descrito. Nuestra **función de coste** tiene la forma:

$$C = \frac{1}{n} \sum_x C_x$$

Donde:

$$C_x = \frac{\|y(x) - a\|^2}{2}$$

Representa el coste de un *training example*. Como vemos, para calcular el gradiente de C, lo que tenemos que hacer en realidad es calcular el gradiente con respecto a cada punto x en los datos:

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x$$

¡Esto es demasiado costoso cuando tenemos un gran número de *training examples*! Imaginemos un dataset de un millón de puntos... Si por cada *update* en *gradient descent* necesitamos calcular un millón de gradientes (y cada uno respecto de una gran cantidad de variables), el entrenamiento de redes neuronales se hace casi impracticable.

Stochastic Gradient Descent

Para solucionar este problema, en la práctica las redes neuronales no se entrenan calculando el gradiente completo, sino una estimación del mismo obtenida con una muestra aleatoria de *training examples*. Haciendo una media sobre una pequeña muestra de ejemplos, pongamos 128, podemos obtener una aproximación al gradiente real lo suficientemente buena como para minimizar la función sin tener que derivar con respecto a cada punto en el dataset. Se puede pensar en esto como en una especie de encuesta electoral, donde se obtiene una idea del voto sin tener que preguntar a cada uno de los electores. El algoritmo se denomina *stochastic gradient descent*, abreviado comúnmente «para los amigos» como **SGD**.

Más formalmente, los m *training examples* $X_1, \ldots, X_m$ a elegir se denominan ***batch*** o ***mini-batch***, con m siendo el tamaño de *batch*. Nuestra estimación es tal que:

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}$$

De este modo, nuestra regla de aprendizaje con *stochastic gradient descent* para redes neuronales queda como:

$$w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Donde:

- j suma sobre los *training examples* de la *batch* $X_1, \ldots, X_m$.

El entrenamiento ocurre *batch* a *batch*. Primero, se elige una muestra aleatoria de m ejemplos, se calculan los gradientes y se actualizan los parámetros. Este proceso se repite hasta agotar todos los *training examples* del dataset. Cuando esto ha ocurrido, decimos que se ha completado una ***training epoch***. En ese momento, se vuelve a empezar con una nueva *epoch* y así sucesivamente hasta que el entrenamiento esté completo.

2.4. *Backpropagation*

Acabamos de ver SGD, un algoritmo utilizado para buscar el mínimo de una función que, aplicado sobre nuestra función de coste, nos lleva a entrenar una red neuronal. SGD utiliza los gradientes respecto a cada parámetro de la red para actualizar estos mismos parámetros y buscar los valores que llevan al

mínimo error o coste. Sin embargo, no hemos visto aún **cómo calcular esos gradientes**.

Una primera idea podría ser, de nuevo, buscar una solución analítica del gradiente de la función final respecto del peso. Pero, tal y como vimos en el apartado anterior, esto se vuelve realmente complejo y difícil para redes neuronales grandes, peor aún si tenemos que buscar una fórmula para cada parámetro individual de la red.

Otra opción sería calcular los gradientes de manera numérica mediante la fórmula de la derivada. Sin embargo, esto se vuelve intratable, ya que necesitaríamos realizar cálculos relativamente complejos por cada uno de los parámetros de la red. Y sabemos que las redes neuronales se caracterizan por poder tener un gran número de ellos.

Sería genial obtener una formulación directa del valor de los gradientes de cada parámetro que fuera computacionalmente eficiente de calcular, permitiendo por tanto entrenar redes neuronales profundas con miles o millones de parámetros. La solución al problema viene dada por el **algoritmo de *backpropagation*** (o propagación hacia atrás), introducido originalmente en los años 70, pero cuya importancia fue puesta en valor en un famoso trabajo de David Rumelhart y Geoffrey Hinton en 1986, *Learning representations by back-propagatin errors*.

Este algoritmo ha sido clave en el desarrollo del *deep learning*, permitiendo la explosión del campo al aportar una forma efectiva de entrenar redes neuronales.

Expresiones simples de derivadas

Antes de continuar con la exposición del algoritmo de *backpropagation*, veamos un rápido repaso del concepto de derivada. La **fórmula de la derivada** es:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Su significado es la velocidad de cambio de una función con respecto a una variable alrededor de una región infinitesimalmente pequeña a su alrededor.

De la misma manera que podemos hacer derivadas de funciones de una variable, podemos hacer derivadas de funciones de varias variables:

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

Por ejemplo: si $x = 4$ e $y = -3$, la derivada de f respecto de y es 4. Esto significa, intuitivamente, que si incrementamos un poco el valor de y dejando fijo el resto de variables (en este caso x), f se vería incrementada en 3 veces ese pequeño cambio en y. Así, podemos ver la derivada como una **medida de la «sensitividad» de una función** respecto a cambios en una de sus variables. Como sabemos, el gradiente en este caso sería:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [y, x]$$

Es frecuente utilizar los términos «derivada» y «gradiente» de manera intercambiable y también es común llamar «el gradiente en x» a la derivada parcial de f respecto de x.

Otros ejemplos de valores de derivadas que nos serán útiles son

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 1$$

$$f(x,y) = max(x + y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

Regla de la cadena

El concepto básico sobre el que se asienta el algoritmo de *backpropagation* es el de la regla de la cadena. Si no estamos muy familiarizados con esta, sería recomendable repasar el concepto con una rápida búsqueda por Internet. Veamos aquí un sencillo ejemplo que nos llevará a nuestro primer ejemplo de *backpropagation*.

Imaginemos la función $f(x,y,z) = (x + y)z$. Esta función es muy sencilla y podríamos obtener las derivadas parciales directamente. Sin embargo, para entender el concepto de composición, llamemos $q = x + y$, con lo que $f = qz$. La regla de la cadena nos dice que:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

De este modo, podemos obtener la derivada parcial de f respecto de x mediante el producto de dos derivadas más sencillas. Si no estás muy familiarizado con las derivadas, deberías probar a calcular la derivada de f respecto de x tanto directamente a partir de la definición de f como a partir de la regla de la cadena, y cerciorarte de que el resultado es el mismo.

Un primer ejemplo de *backpropagation*

Sigamos utilizando nuestra función $f(x,y,z) = (x + y)z$ para ver un ejemplo básico del algoritmo de *backpropagation*. Imaginemos que ponemos las operaciones

de esta función en forma de un circuito, y calculemos paso a paso los valores de f y de los gradientes respecto a las variables:

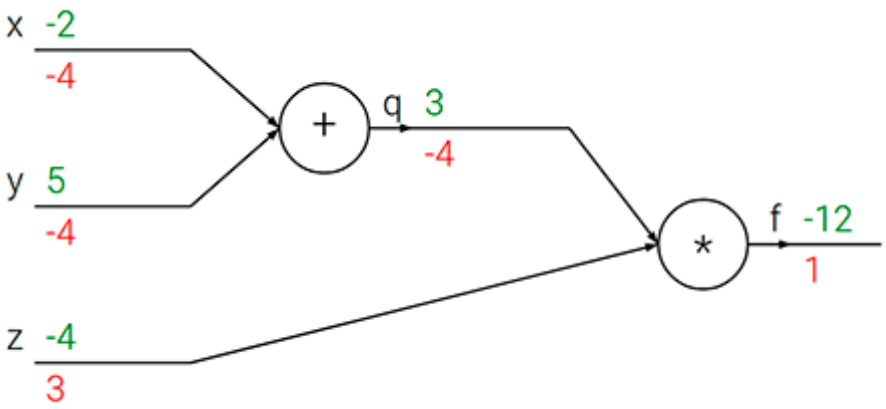


Figura 6. $f(x, y, z)$ con valores en verde y gradientes en rojo.
Fuente: http://cs231n.github.io/optimization-2/

En este ejemplo tenemos como valores iniciales:

$$x = -2$$
$$y = 5$$
$$z = -4$$

De este modo, obtenemos el valor de:

$$q = 3$$
$$f = -12$$

Este primer paso en el que vamos calculando los valores hacia adelante es lo que se conoce como ***forward pass***. Una vez tenemos esto, aplicaremos recursivamente la regla de la cadena hacia atrás en lo que se conoce como ***backward pass***.

Como convención, el primer valor de gradiente sería el gradiente de f respecto a sí misma, que en este caso es 1, si bien esto no es demasiado importante. Empecemos ahora con z. Sabemos que $f = qz$, y por tanto, la derivada de f con respecto de z

es q. Como sabemos del *forward pass* que el valor de q es 3, tenemos que el gradiente en z es entonces 3. De manera similar, tenemos que el gradiente en q es -4.

Necesitamos calcular ahora el gradiente de f respecto de x. Por la regla de la cadena, sabemos que:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

- El gradiente de f respecto de q lo acabamos de obtener en el paso anterior y tiene valor -4.
- Asimismo, el gradiente de q respecto de x es 1, por tanto, al multiplicar ambos valores tenemos que el gradiente en x es -4.
- De manera similar, tenemos el mismo valor para el gradiente en y.

Podemos ver cómo **el proceso de *backpropagation* es local**. Basta con saber el gradiente local de cada nodo en este circuito con respecto a sus *inputs* y el valor del gradiente que viene propagándose desde el *output*, como se ve en la siguiente imagen:

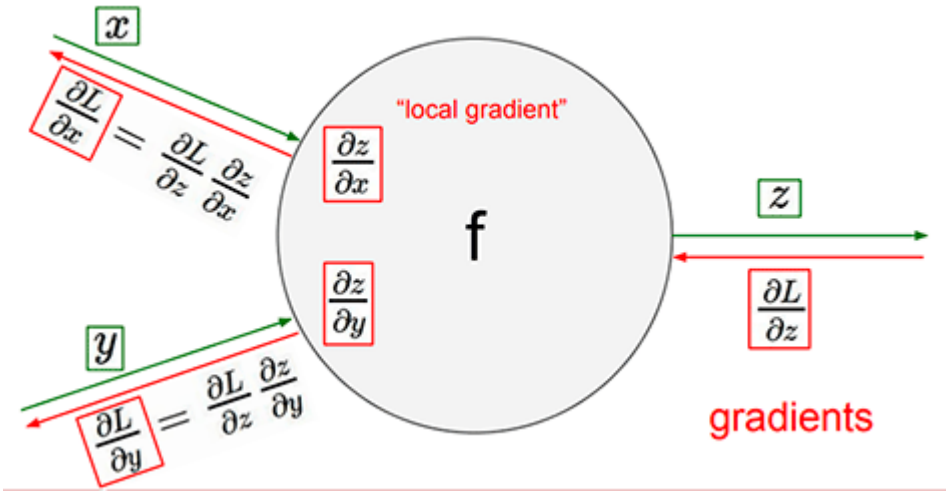


Figura 7. Gradiente local.
Fuente: http://cs231n.stanford.edu/

La regla de la cadena nos dice que basta con multiplicar el valor del gradiente que viene propagado desde arriba con el valor del gradiente respecto a cada *input*.

Un ejemplo ligeramente más complejo

Veamos aquí un ejemplo un poco más complejo aplicado a lo que sería la activación *sigmoid* en una red neuronal. Dejaremos aquí escritos los ingredientes básicos para calcular los valores, lo cual se deja como ejercicio para el alumno.

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

Los valores de las derivadas necesarias son:

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

Y el circuito obtenido con sus valores de *forward pass* en verde y de *backward pass* en rojo es:

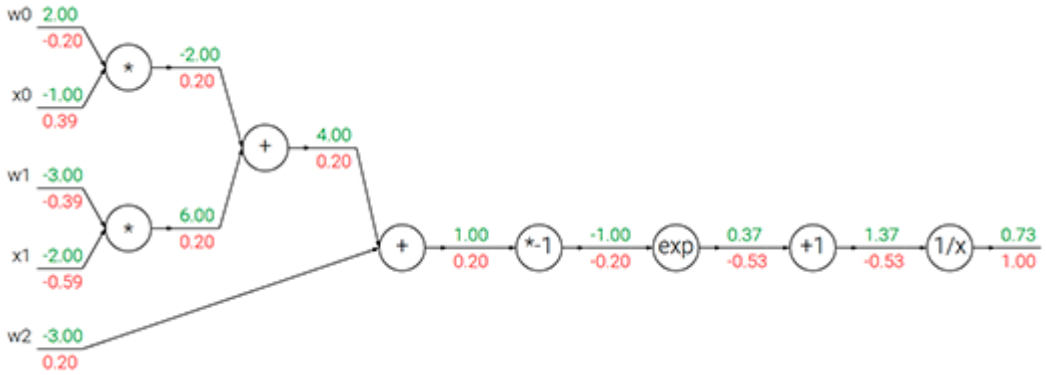


Figura 8. Ejemplo de activación *sigmoid*.
Fuente: http://cs231n.github.io/optimization-2/

Algunos patrones en *backpropagation*

Al hacer estos ejemplos, es posible que nos hayamos fijado en ciertos patrones que aparecen en las operaciones más comunes que aplicamos.

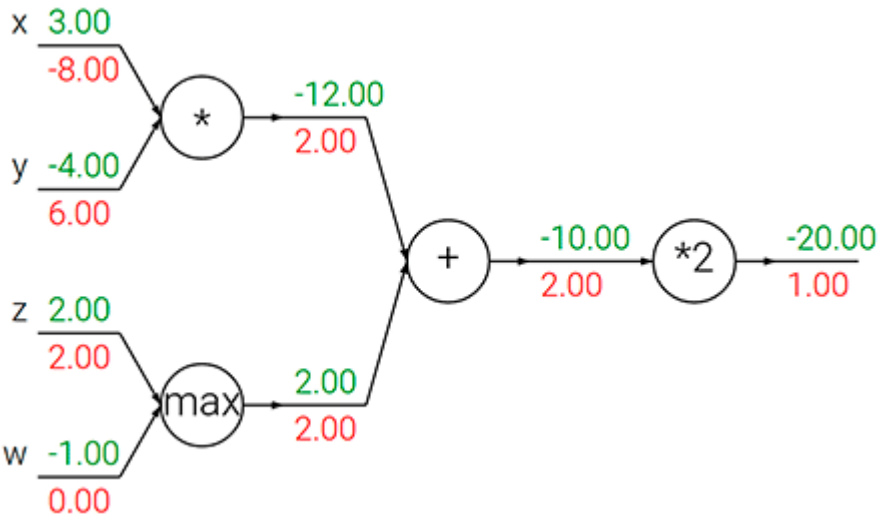


Figura 9. Patrones que aparecen en las operaciones más comunes.
Fuente: http://cs231n.github.io/optimization-2/

Suma: la suma simplemente coge el gradiente de su salida y lo envía para atrás a todos sus *inputs*.

Producto: para un *input* se multiplica el valor del otro *input* por el valor del gradiente de salida.

Max: el mayor valor del *input* se lleva todo el valor del gradiente de salida, mientras que el menor *input* se queda gradiente 0.

Algoritmo de *backpropagation*: recapitulando

Hasta ahora, hemos visto cómo aplicar *backpropagation* a circuitos sencillos de operaciones. Una red neuronal no es más que un circuito o grafo de computación más grande; de hecho, su propia forma es ya la de un grafo de computación donde en cada nodo tenemos una función de activación aplicada al producto de los pesos por los *inputs* más el *bias*. Si bien podríamos descomponer esto hasta el nivel de operaciones básicas (como hemos hecho en los ejemplos de arriba), también es posible considerar nodos que engloben operaciones más grandes, como la función *sigmoid*.

Por ejemplo: la derivada de la función *sigmoid* tiene una derivación analítica muy simple:

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

$$\rightarrow \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = \big(1-\sigma(x)\big)\sigma(x)$$

En definitiva, para aplicar el algoritmo de *backpropagation*, por cada operación en un nodo que definamos es necesario calcular el valor de la salida en el *forward pass* a partir de sus *inputs* y aportar el cálculo de gradientes de las *inputs* a partir del gradiente propagado.

Por ejemplo, el código para una operación producto sería:

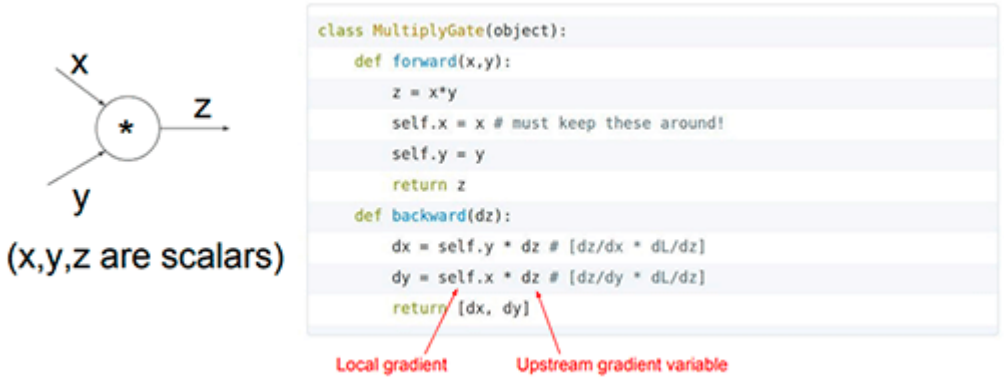


Figura 10. Código para una operación producto.
Fuente: http://cs231n.stanford.edu/

La belleza de romper la computación en pequeños nodos radica en que podemos aplicar la misma idea a funciones arbitrariamente complejas. En particular, las arquitecturas de redes neuronales que veremos durante el curso, que usan conceptos más complejos que las de una simple red neuronal clásica, pueden ser entrenadas igualmente mediante *backpropagation* siempre que sus operaciones sean diferenciables y puedan representarse en forma de un grafo de computación. Como veremos en el tema de frameworks de *deep learning*, así es como funcionan librerías de software como TensorFlow.

Recapitulando, para un solo *training example* x, el algoritmo final de *backpropagation* quedaría así:

1. *Forward pass*: aplicar el valor de x a la red neuronal y guardar todos los valores intermedios de salida de cada operación o neurona.

2. *Backward pass*: una vez obtenido el valor final de salida, calcular el valor de la función de coste. La función de coste, que tiene que ser una función diferenciable, también tiene un *input* y unos gradientes respecto a ese *input*. De manera recursiva, propagar el gradiente hacia atrás mediante el uso de la regla de la cadena y calcular los gradientes de cada parámetro.

3. Al terminar el *backward pass*, tendremos calculados los valores de los gradientes de cada parámetro de la red neuronal. Con esto, es posible aplicar *stochastic gradient descent*.

En la práctica (como hemos visto antes), el proceso de *stochastic gradient descent* se hace sobre *mini-batches*. Por tanto, por cada *batch* conteniendo m *training examples*, calcularemos los gradientes de los parámetros para cada *training example* y, finalmente, aplicaremos la *update rule* sumando sobre los valores de cada *input* en la *batch*:

$$w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Varios comentarios finales sobre *backpropagation*

Como hemos visto, el algoritmo de *backpropagation* define, a partir de conceptos matemáticos relativamente sencillos como la regla de la cadena, una forma efectiva de entrenar redes neuronales. Mediante los procesos de *forward* y *backward pass* podemos obtener los valores necesarios para entrenar una red neuronal de complejidad arbitraria sobre una *batch* de *inputs*, evitando el cálculo de complejas derivadas de manera analítica o de gradientes obtenidos numéricos en cada parámetro.

Es importante mencionar que en la práctica se suelen utilizar cálculos vectorizados, donde los parámetros e *inputs* son matrices o vectores. Si bien los gradientes de cada nodo son un poco más difíciles de obtener analíticamente mediante cálculo vectorial, lo cierto es que se siguen las mismas reglas básicas.

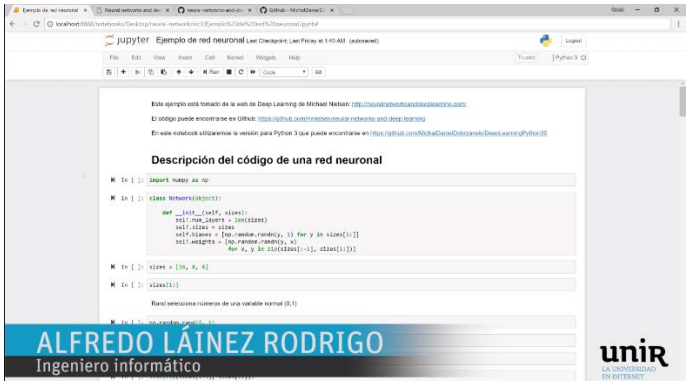
Similarmente, en la literatura es frecuente ver el algoritmo de *backpropagation* totalmente desarrollado para el caso de las redes neuronales básicas. La idea es la misma que la vista aquí, pero con todas las cuentas hechas para el caso particular de una red neuronal. Este proceso es un muy buen ejercicio para entender el algoritmo, aunque resulta también algo lioso.

Lo + recomendado

Lecciones magistrales

Implementación de una red neuronal

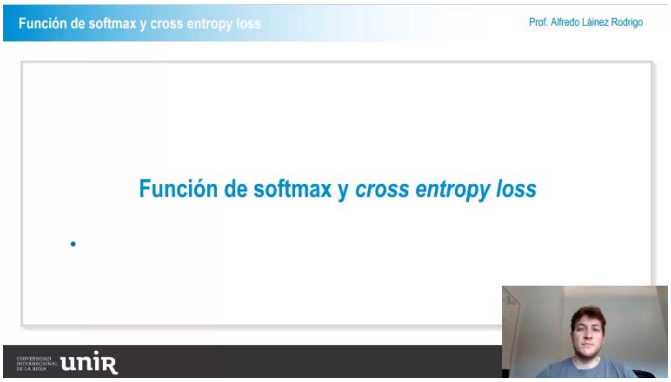
En este vídeo veremos un ejemplo práctico sobre el código interno que hace funcionar una red neuronal mediante librerías de bajo nivel. El ejemplo está tomado de *Neural Networks and Deep Learning* de Michael Nielsen, cuyo código está en Github y se utilizará la versión para Python 3.



Accede a la lección magistral a través del aula virtual

Función de *softmax* y *cross entropy loss*

En esta magistral vamos a ver cómo funciona un clasificador de tipo softmax, que se caracteriza por ser estándar para problemas de clasificación con clases excluyentes.



Accede a la lección magistral a través del aula virtual

Ejecución de la red neuronal

Continuaremos trabajando en un proceso de implementación de una red neuronal. En este caso, entrenaremos a la red neuronal con datos reales utilizando el conocido dataset de MNIST (clasificación de números manuscritos del 1 al 10) con el objetivo de que la red reconozca un número escrito a mano de manera automática.

Accede a la lección magistral a través del aula virtual

No dejes de ver

Vídeos sobre redes neuronales de 3Blue1Brown

Esta serie de tres vídeos acerca de redes neuronales es sencillamente magnífica. Las visualizaciones y las explicaciones hechas aquí ayudarán a entender mucho mejor este tema y el curso en general. Los vídeos disponen de subtítulos en castellano.

Hay un cuarto vídeo que explica detenidamente el desarrollo de *backpropagation* para una red neuronal sencilla. Si bien no es necesario entender las matemáticas y ser capaz de realizar las derivaciones presentes en el vídeo, es un ejercicio de gran interés para entender cómo funciona este algoritmo.

Accede a los vídeos a través del aula virtual o desde la siguiente dirección web:

Vídeo 1: https://youtu.be/aircAruvnKk

Vídeo 2: https://youtu.be/IHZwWFHWa-w

Vídeo 3: https://youtu.be/Ilg3gGewQ5U

Vídeo 4: https://youtu.be/tIeHLnjs5U8

+ Información

Webgrafía

Neural networks and Deep Learning

Parte de este tema se basa en este libro/web escrito por Michael Nielsen. Es un gran recurso con gran lujo de detalles para comprender las redes neuronales.

Accede a la página web a través del aula virtual o desde la siguiente dirección:
http://neuralnetworksanddeeplearning.com/

Bibliografía

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 3*(23), 533-536.

Test

1. Marca las respuestas verdaderas sobre el algoritmo *stochastic gradient descent:*

A. En SGD obtenemos una estimación del gradiente. Si bien el gradiente no es exacto, la dirección de este es más o menos correcta, de modo que el proceso de minimización todavía funciona.

B. El tamaño de la *batch* m cuanto más pequeño, mejor para acelerar el entrenamiento. De hecho es buena idea usar valores de m como 1, 2 o 3.

C. La elección de m afecta a la velocidad de entrenamiento de nuestra red neuronal.

D. Hay que elegir m con un balance, de modo que la aproximación al gradiente sea buena, pero no necesitemos utilizar una gran cantidad de puntos del dataset.

2. Seleccionar un motivo por el que en SGD utilizamos *batches* aleatorias hasta agotar todos los *training examples* del dataset (completar una *epoch*), en vez de utilizar siempre elementos al azar, es decir, sin la necesidad de utilizar todos los *training examples* del dataset antes de repetir elementos:

A. Programar una lógica de muestreo con reemplazamiento es más difícil y computacionalmente costoso.

B. Una estimación utilizando elementos siempre al azar es una estimación incorrecta y la red neuronal nunca llegaría a aprender nada.

C. Todos los *training examples* del dataset son importantes y poseen una variabilidad que la red neuronal tiene que saber explicar. Al forzar a la red a entrenar con todos ellos, permitimos que todos estos datos sean tomados en cuenta.

3. Para aplicar *gradient descent*, la función de coste tiene que ser (marca la respuesta correcta):

A. Diferenciable.

B. Integrable.

C. Contener cuadrados de valores.

D. Convexa.

4. Si aplicamos un valor demasiado grande de *learning rate* (marca la respuesta correcta):

A. La red neuronal aprendería y lo haría de manera más rápida.

B. La red neuronal aprendería igualmente, el efecto de la *learning rate* no es realmente importante.

C. La red neuronal podría no aprender, ya que SGD acabaría convergiendo en un máximo en vez de en un mínimo.

D. La red neuronal podría no aprender. La aproximación local de la derivada deja de tener efecto y podríamos *overshoot* a un punto lejano donde el valor de la función es de hecho mayor.

5. Si un dataset tiene 10 000 puntos y utilizamos *batches* de 10 *examples* para entrenar una red neuronal (marca la respuesta correcta):

A. Una *epoch* o época de entrenamiento consiste en 10 *steps*.

B. Una *epoch* o época de entrenamiento consiste en 100 *steps*.

C. Una *epoch* o época de entrenamiento consiste en 1000 *steps*.

D. Una *epoch* o época de entrenamiento consiste en 10 000 *steps*.

E. Ninguna de las anteriores.

6. Las funciones de coste o *loss functions* (marca todas las respuestas correctas):

A. Definen un valor de error que queremos minimizar.

B. Definen un valor de coste que queremos maximizar.

C. Son siempre variaciones de la función *mean squared error*.

D. Implican un objetivo distinto a la hora de entrenar una red neuronal y, por tanto, son una parte importante a definir en un problema de *machine learning*.

E. Son poco importantes a la hora de entrenar.

7. Durante el algoritmo de *backpropagation* (marca la respuesta correcta):

A. No es necesario almacenar los valores de salida de cada nodo, solo nos interesa el valor final de la función de coste tras ejecutar el *forward pass*.

B. Guardamos los valores de salida de cada nodo, ya que son necesarios para el cálculo de gradientes durante el *backward pass*.

C. Guardamos los valores de salida de cada nodo, ya que son necesarios para aplicar *gradient descent*, pero no son usados en el cálculo de gradientes.

D. Ninguna de las anteriores.

8. Marca todas las respuestas correctas acerca del algoritmo de *backpropagation*:

A. Es una forma eficiente de calcular los gradientes necesarios para redes neuronales arbitrariamente complejas.

B. Se llama *backpropagation* ya que el gradiente se propaga de manera recursiva hacia atrás.

C. Fue un gran avance, ya que calcular fórmulas de los gradientes analíticamente se vuelve muy complejo para las redes neuronales.

9. En una operación suma $q = x + y + z$, si el gradiente propagado de salida es 5 (marca la respuesta correcta):

A. El gradiente es 15 en todas las variables.

B. El gradiente es 1 en todas las variables.

C. El gradiente es 5 en todas las variables.

D. No es posible saber el valor del gradiente con la información suministrada.

10. El número de neuronas a utilizar en las *hidden layer* (marca las respuestas correctas):

A. Suele estar comprendido entre 10 y 100 neuronas. Menos es insuficiente y más es superfluo.

B. Puede ser obtenido mediante una búsqueda de hiperparámetros.

C. A más neuronas, más gradientes a calcular y, por tanto, más requisitos de memoria y tiempo de ejecución.

D. No es muy importante a la hora de entrenar una red neuronal.

E. A más neuronas, más capacidad de representación que tiene la red.