Sistemas Cognitivos Artificiales

Roberto Casado Vara

Tema 2: Entrenamiento de redes neuronales

En el tema 2…

- Clase hoy:
  - Funciones de coste
  - Entrenamiento con “gradient descent”
- Semana que viene:
  - Backpropagation

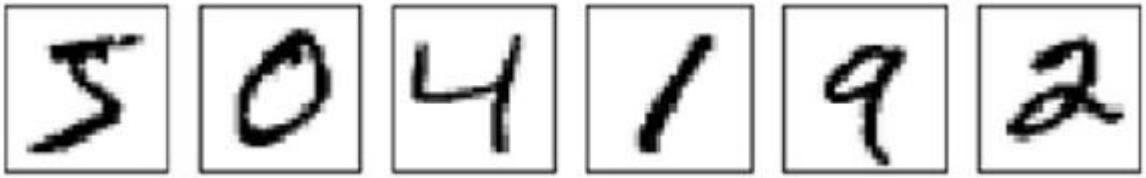
Sistemas Cognitivos Artificiales

Roberto Casado Vara

Tema 2.1: Funciones de coste

MNIST

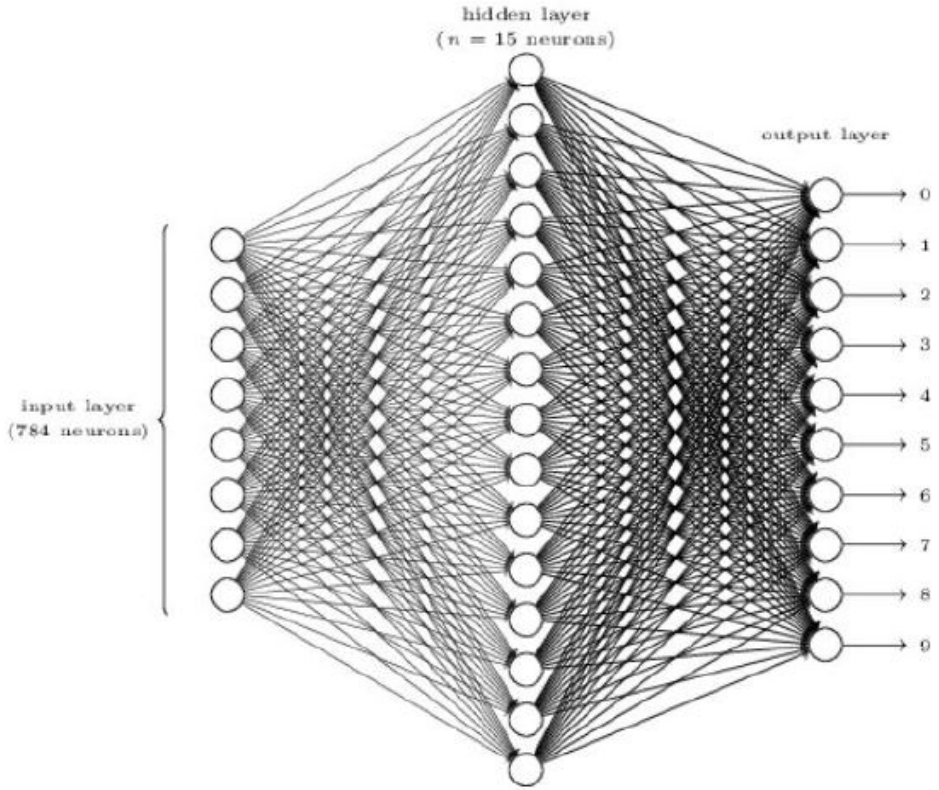
- Vamos a ver un ejemplo de red neuronal que sirve para clasificar dígitos manuscritos.
- Para ello, utilizaremos el **dataset MNIST**, una especie de *"Hello world"* del aprendizaje automático.
- MNIST es un dataset que contiene números manuscritos del 0 al 9.



- El tamaño del dataset es de 60.000 elementos de *training data* y 10.000 elementos de *test data*.
- Es un problema muy sencillo para las técnicas actuales.

Clasificación de dígitos

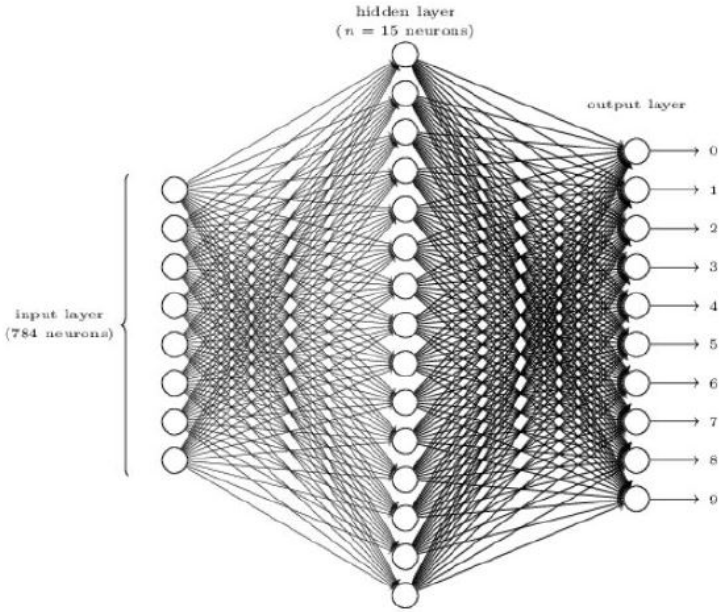
- Tenemos por tanto un problema de clasificación con 10 clases: los números del 0 al 9.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

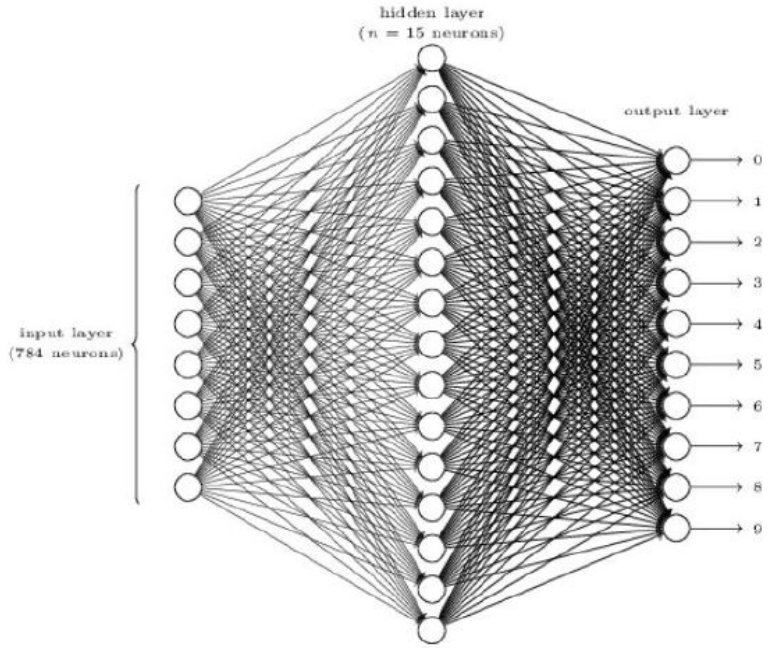
- **Input layer:** Las imágenes están compuestas por píxeles en blanco y negro de tamaño **28x28**.
- Necesitamos un vector para la entrada de nuestra red, por tanto concatenamos todos los píxeles en una fila de **784** elementos. Cada neurona representa el valor de un píxel, con un valor entre **0 y 1** según la **oscuridad** del píxel.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

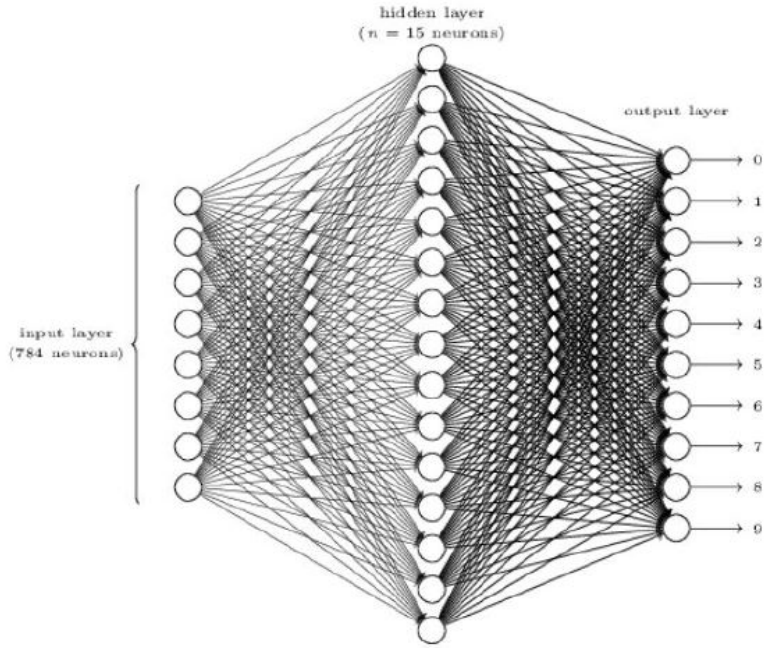
- **Hidden layers:** Utilizaremos una única *hidden layer* (el problema es relativamente fácil) con 15 neuronas. Se puede experimentar con el número de *hidden layers* y de neuronas.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

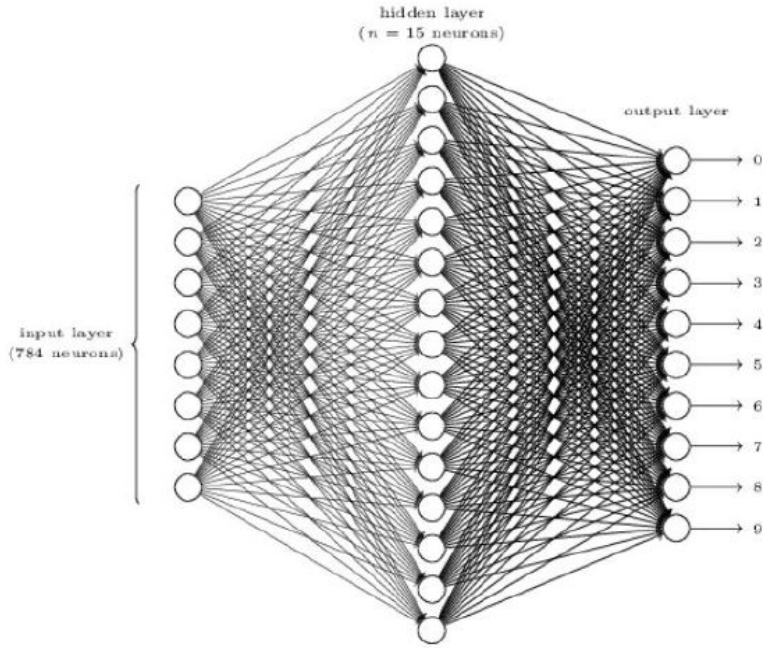
- **Output layer:** Tendremos 10 neuronas de salida, una por cada posible número a clasificar.
- La neurona de esta capa que tenga la **mayor activación de salida** indicará el número resultante.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

- Pregunta: **¿Cuántas neuronas necesitamos en la *hidden layer*?**
- Este número es un **hiperparámetro** de la red y somos libres para elegirlo.
- No hay una fórmula exacta. Normalmente se determina de manera empírica mediante una búsqueda de hiperparámetros.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

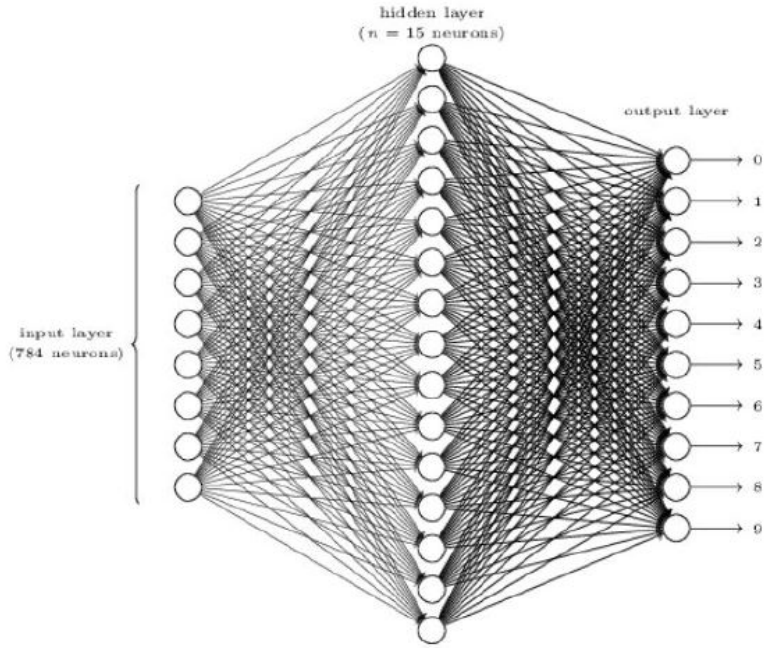
Clasificación de dígitos

- Intentemos obtener una intuición de lo que está pasando en la hidden layer. Debería estar claro que al **añadir más nodos** a esta capa, **estamos añadiendo más potencia a la red**, ya que a partir de los píxeles de entrada obtenemos más representaciones intermedias.
- En general, las representaciones intermedias que obtiene una red neuronal no son fáciles de entender y escapan a nuestra comprensión. Sin embargo, en este ejemplo, al ser sobre imágenes, se puede visualizar qué neuronas se activan al ver ciertos números, por lo que podemos deducir qué formas o features está aprendiendo a reconocer la red.

Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

- Pregunta: **¿Cúantas neuronas necesitamos en la *hidden layer*?**
- Con esto podemos comprender que no hay una respuesta clara.
- Necesitamos un número suficiente para que la red obtenga unas **representaciones intermedias adecuadas**. A partir de cierto punto, tener más neuronas no va a ayudar, ya que la red tiene suficiente capacidad para expresar la variabilidad encontrada en los datos.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Funciones de coste

Notación:

- ***x***: *input* o *training example* (en nuestro caso, vector de 784 píxeles).
- ***y(x)***: valor deseado de salida de la red, representado como vector de 10 componentes.
- Ej: *y(x)* = *(0, 0, 0, 1, 0, 0, 0, 0, 0, 0)* representa la salida deseada de una imagen que es un 3.

- *y(x)* es la función que queremos aprender. Es una función que asocia a cada imagen en forma de píxeles el número correspondiente, en forma de un vector de 10 elementos.
- La red neuronal también es una función, dependiente de los parámetros *w* y *b* y de la imagen *x*. Definiremos esta función como ***a(x, w, b)*** o como ***a*** para simplificar.

Clasificación de dígitos

- **Objetivo**: obtener (**"aprender"**) los parámetros *w* y *b* de la red neuronal que mejor aproximen la función *y(x)* para todos los valores x, esto es, para todos los *training examples* de nuestro dataset.
- Necesitamos cuantificar de algún modo cómo de buena es la aproximación de *a(x, w, b)* a *y(x)*. Para ello definimos la **función de coste** o **cost function**:

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

donde:

1. *a* es la salida de la red para *a(x, w, b)*.
2. *n* es el número de *data points* en nuestro dataset.
3. *w* y b son los parámetros de la red neuronal.
4. El sumatorio sobre *x* es sobre todos los *training examples* de nuestro dataset.

Clasificación de dígitos

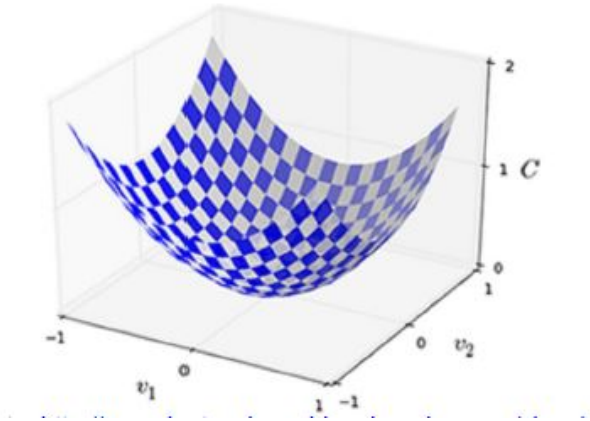
$$C(w,b) \equiv \frac{1}{2n}\sum_{x}\|y(x)-a\|^2$$

- La función de coste es una suma de valores que miden el **error** (o ***loss***, o **pérdida**) que nuestra red *a* está cometiendo a la hora de aproximar la función real *y*.
- Estamos obteniendo el cuadrado de la distancia vectorial, un número siempre positivo, entre el valor real de salida y la salida de nuestra red.
  - Por ejemplo, podríamos tener:
    - *a(x, w, b)* = (0.9, 0.1, 0.1, 0.2, 0.1, 0.1, 0, 0, 0, 0)
    - *y(x)* = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
    - Para este ejemplo, el error sería:
    - $(1 - 0.9)^2 + (0 - 0.1)^2 + (0 - 0.1)^2 + (0 - 0.2)^2 + \ldots$
    - Sumamos los errores de todos los datos *x*.

Clasificación de dígitos

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

- Nuestro objetivo al entrenar la red neuronal es conseguir que nuestra aproximación al objetivo real sea lo mejor posible, esto es, minimizar en la medida de lo posible *C(w, b)*.
- El problema a resolver es un **problema de optimización**: queremos obtener los valores de *w* y *b* que **minimicen** el valor de *C*.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Clasificación de dígitos

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

- El error podría ser medido de otras formas. Existen distintas **funciones de pérdida** o **loss functions** para medir el error en el que está incurriendo el modelo. En este caso estamos utilizando la función de pérdida conocida como **Mean Squared Error** (MSE).
  - Normalmente, para problemas de clasificación se utiliza ***cross-entropy loss***.
- Durante el curso, utilizaremos los términos de coste y pérdida de manera intercambiable. Técnicamente, podríamos decir que la diferencia está en que la *loss function* mide el error en un ejemplo mientras que la función de coste es la suma de errores de todos los ejemplos.

Clasificación de dígitos

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

- Hemos definido una función *y(x)* que modela el problema a solucionar.
- Buscamos encontrar una función que aproxime *y* de la mejor manera posible. Utilizamos una red neuronal modelada con parámetros *w* y *b* para ello: *a(x, w, b)*.
- Para cuantificar esta aproximación, definimos una función de coste *C* que nos da una medida del error que comete nuestra red neuronal a partir de los datos a utilizar.
- Entrenar la red neuronal se convierte en un **problema de optimización**: queremos encontrar los valores de los parámetros *w* y *b* que minimizan *C*.

Sistemas Cognitivos Artificiales

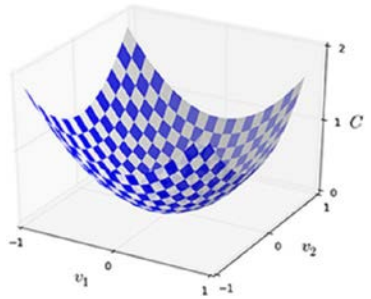
Roberto Casado Vara

Tema 2.2: Entrenamiento con “gradient descent”

Mínimo de una función de varias variables

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

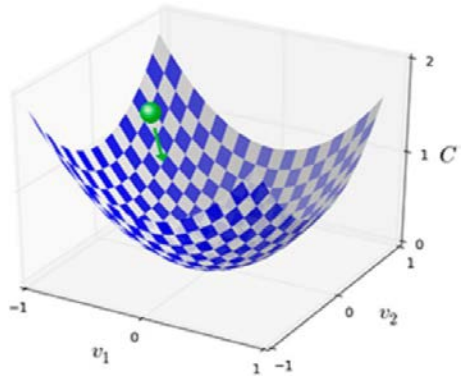
- Vamos a ver primero cómo minimizar una función general *C(v)*, donde *v* puede ser un vector de varias variables.
- Una forma de obtener el mínimo de una función es calcularlo analíticamente mediante la derivada: *C'(v)* = 0. Sin embargo, esto no va a ser factible para funciones tan complejas y con tantos parámetros como el coste asociado a una red neuronal.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Gradient descent: idea general

- Empezamos en un punto (v_1, v_2) al azar.
- Encontramos la dirección de máxima pendiente hacia abajo.
- Damos un pequeño paso en esa dirección.
- Repetimos el proceso desde el nuevo punto obtenido hasta llegar a un mínimo.



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Gradient descent: formalización

- La **derivada** de una función en un punto con respecto a una variable nos da una aproximación **local** de cuánto cambia el valor de la función al variar esa variable.

- Supongamos que tenemos una función con dos variables v_1 y v_2. La variación en C, denotada como ΔC, al realizar pequeñas variaciones en sus variables, puede aproximarse como:

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

- El objetivo es hacer $\Delta C < 0$.

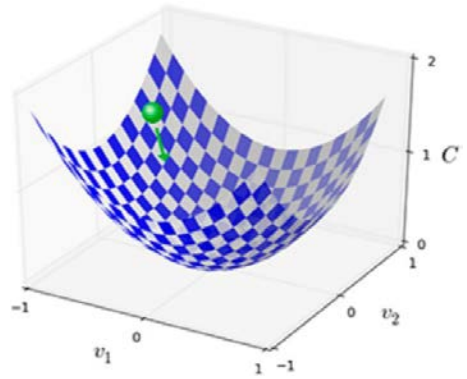
Gradient descent: formalización

- 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

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



Fuente: http://neuralnetworksanddeeplearning.com/chap1.html

Gradient descent: formalización

Con

$$\Delta v = (\Delta v_1, \Delta v_2)^T \quad y \quad \nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

podemos reescribir

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

como el producto escalar:

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

Gradient descent: formalización

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

Supongamos que elegimos $\Delta v = -\eta \nabla C$

donde η (*eta*) es una pequeña constante positiva llamada **learning rate**.

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

Como η y la norma al cuadrado de un vector son positivas, tenemos que $\Delta C < 0$, como queríamos demostrar.

Hemos demostrado que si nos movemos con $\Delta v = -\eta \nabla C$ estamos obteniendo un valor menor para C.

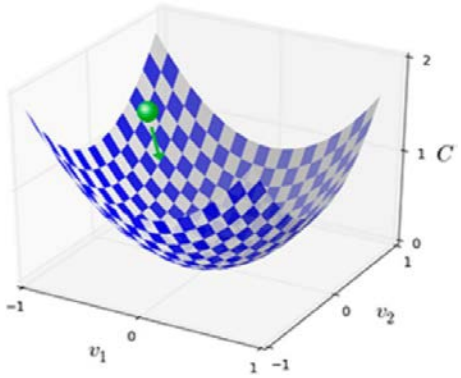
Gradient descent

Hemos demostrado que si nos movemos con $\Delta v = -\eta \nabla C$ estamos obteniendo un valor menor para *C*.

Podemos definir la regla de movimiento para movernos de un punto v a un punto v', donde *C* tiene un valor menor, como:

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

Esta es la **update rule** de **gradient descent**.



Gradient descent

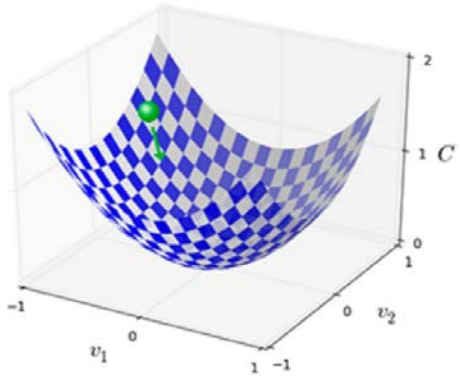
Algoritmo gradient descent:

Aplicamos la ***update rule***

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

hasta llegar a un mínimo.

Intuición: nos “dejamos caer” en la dirección de máxima variación hasta llegar a un mínimo de la función.

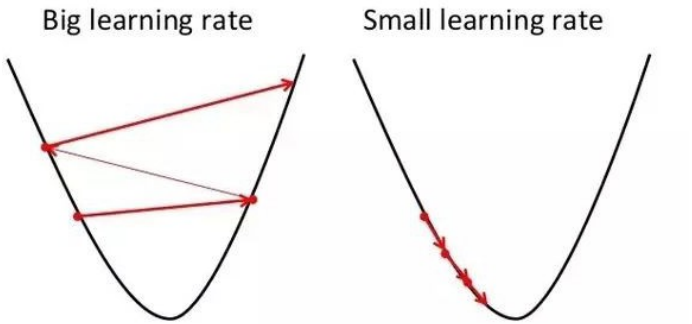


Learning rate

- El **learning rate** η, como su propio nombre indica, define en cierta medida la velocidad a la que gradient descent funciona. Tiene que ser lo suficientemente pequeño para que la **aproximación local** que define la derivada sea correcta y la derivación anterior sea cierta.

- Si η es demasiado grande, podríamos no encontrar el mínimo o incluso divergir a valores de *C* mayores.

- Por otro lado, un valor demasiado pequeño haría que el algoritmo avanzara de forma muy lenta.



Fuente: https://www.quora.com/What-is-the-meaning-of-changing-the-LR-learning-rate-in-neural-networks

Gradient descent para redes neuronales

- En nuestro caso, los parámetros de la función a minimizar son los pesos *w* y los biases *b* de la red.

- Por tanto, la *update rule* queda como:

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

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

para cada uno de los valores w_k y b_l

- La función de coste tiene que ser **diferenciable**, ya que necesitamos calcular derivadas.

Gradient descent para redes neuronales

- Sin embargo, hay un pequeño problema a la hora de calcular el gradiente. Nuestra función de coste tiene la forma:

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

donde

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

es el coste o pérdida de un *training example*. Para calcular el gradiente de C, ¡realmente tenemos que calcular el gradiente con respecto a cada punto x!

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

- Esto es **demasiado costoso**. Tenemos que calcular un gradiente por cada punto en el dataset, y sólo para un *step* del algoritmo.

Stochastic Gradient Descent (SGD)

- En la práctica, una red neuronal no se entrena calculando el gradiente completo, sino una **estimación** del mismo obtenido con una **muestra aleatoria** de *training examples*.

- En vez del gradiente completo, hacemos una media de gradientes a partir de una pequeña muestra de ejemplos, por ejemplo 128, lo cual es una gran diferencia para datasets con miles o millones de puntos. Podemos pensar en ello como una especie de encuesta electoral donde nos hacemos una idea del valor del gradiente.

- Este algoritmo se denomina **Stochastic Gradient Descent**, o **SGD**.

Stochastic Gradient Descent (SGD)

- Elegimos *m* examples del dataset: $X_1, X_2, \ldots, X_m$. Este conjunto de *m* puntos se denomina ***batch*** o ***mini-batch***.

- Tamaños comunes para una mini-batch: 16, 32, 64, 128, 256, 512… A valores mayores, mejor aproximación, pero más operaciones. El uso de *batches* ayuda también a hacer cálculos vectorizados.

- Estimación del gradiente: $\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}$

- Por tanto, la regla de aprendizaje con SGD es, con *j* de 1 a *m*:

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

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

Recordatorio: update de gradient descent

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

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

Stochastic Gradient Descent (SGD)

- El entrenamiento ocurre *batch* a *batch* (elegidas aleatoriamente) hasta completar todos los elementos del dataset. Cuando hemos utilizado todos los valores del dataset en batches, decimos que hemos entrenado una **epoch**.

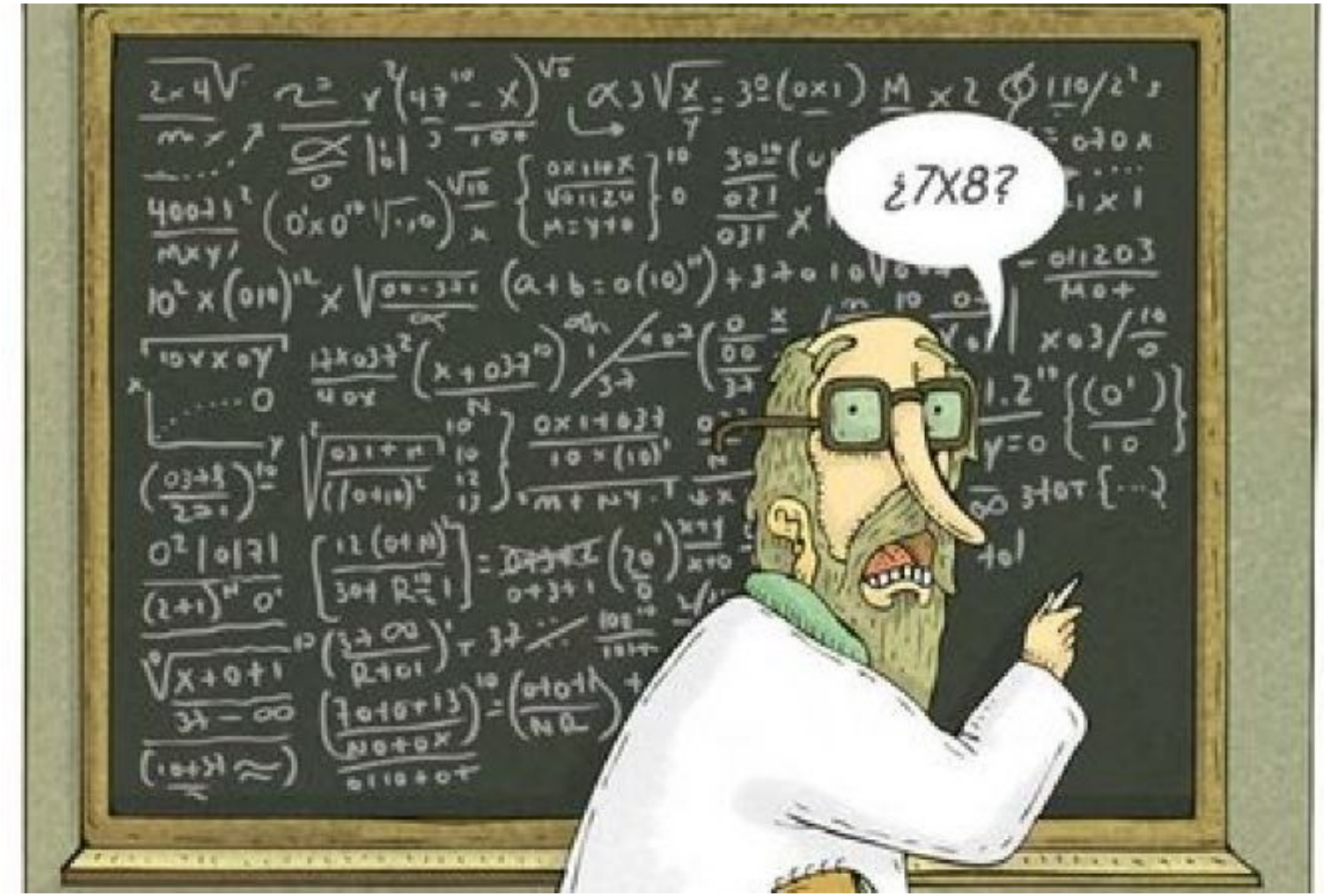
- **Proceso**:

```
for epoch=1 to num_epochs:
    mientras queden training examples por ver en la epoch:
        1.  elegir una batch de m elementos no utilizados en la epoch
        2.  calcular gradientes de los parámetros y aplicar SGD
```

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

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

¿Dudas?



Sistemas Cognitivos Artificiales

Roberto Casado Vara

Tema 2.3: Backpropagation

Universidad Internacional de La Rioja

Trabajo en grupo

- Apuntaros en el foro creando un nuevo hilo para vuestro grupo
- El resto de miembros del grupo que confirmen.

Semana 3, tema 2: Backpropagation

- Hemos visto la semana pasada SGD, un algoritmo que permite entrenar una red neuronal a partir de los gradientes respecto a cada parámetro de la red.

- En este apartado vamos a ver **cómo calcular de forma teórica estos gradientes**.

Backpropagation

- Una primera idea para calcular los gradientes sería obtener una solución analítica (derivar a mano la función de la red respecto a cada parámetro). Pero esto se hace extremadamente complejo para redes neuronales de gran tamaño.

- Otra opción sería calcular los gradientes de manera numérica mediante la fórmula de la derivada. Sin embargo, esto es prácticamente intratable ya que necesitaríamos hacer cálculos relativamente complejos para cada parámetro de la red.

- Fuentes en toda la presentación:
  - https://www.youtube.com/channel/UCy5znSnfMsDwaLlROnZ7Qbg
  - http://neuralnetworksanddeeplearning.com/

Repaso de derivadas

- Fórmula de la derivada:

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

- Nos da la velocidad de cambio de una función con respecto a una variable alrededor de una región infinitesimalmente pequeña de esa variable.

- Podemos hacer derivadas de funciones de varias variables, así como obtener su gradiente:

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

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

Repaso de derivadas

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

- 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 fija la variable *x*, *f* se vería incrementada en 4 veces ese pequeño cambio en *y*.

- Podemos entender la derivada como una medida de la **sensitividad** de una función respecto a cambios en una de sus variables.

- Otros ejemplos:

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

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

Errata en el tema

- El concepto básico sobre el que se asienta el algoritmo de *backpropagation* es el de la **regla de la cadena**.

- Esta nos permite derivar funciones complejas que se componen de otras funciones.

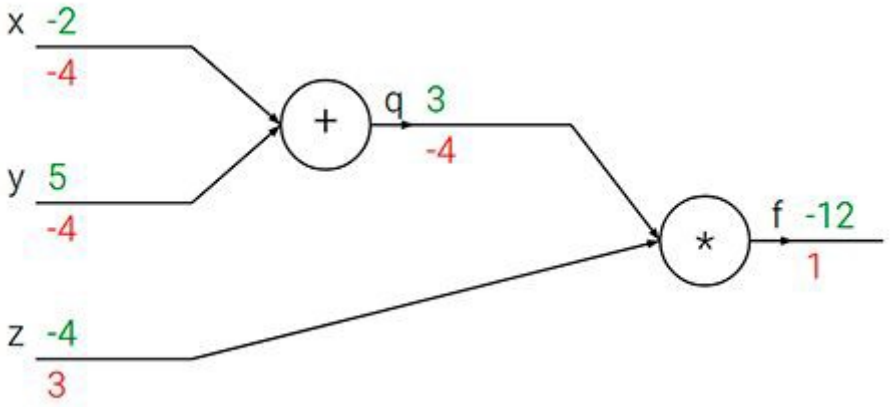
- Por ejemplo, $f(x, y, z) = (x + y)z$ podría ser escrita como $f = qz$ si definimos $q = x + y$. La regla de la cadena nos dice que:

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

- De este modo, obtenemos la derivada parcial de f respecto de x mediante el producto de dos derivadas más sencillas.

- Ejercicio: Calcular la derivada del ejemplo anterior directamente y mediante la regla de la cadena.

$$f(x, y, z) = (x + y)z$$



- Construimos un **grafo de computación** para entender las dependencias entre operaciones.
- Primero: ***forward pass***. Calculamos las salidas de cada nodo a partir de sus inputs.
- Segundo: ***backward pass***. Aplicamos recursivamente la regla de la cadena hacia atrás.

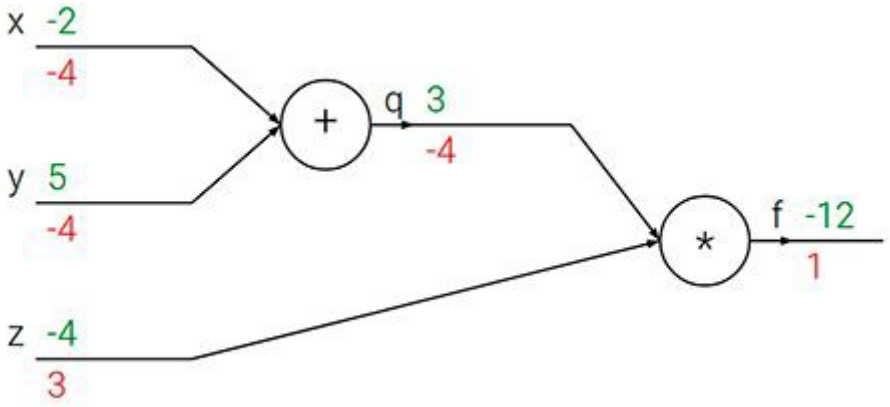
Recordatorio:

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/

$$f(x, y, z) = (x + y)z$$



- *Forward pass:*
  - Obtenemos las salidas de la operación intermedia q y de la salida final de f.

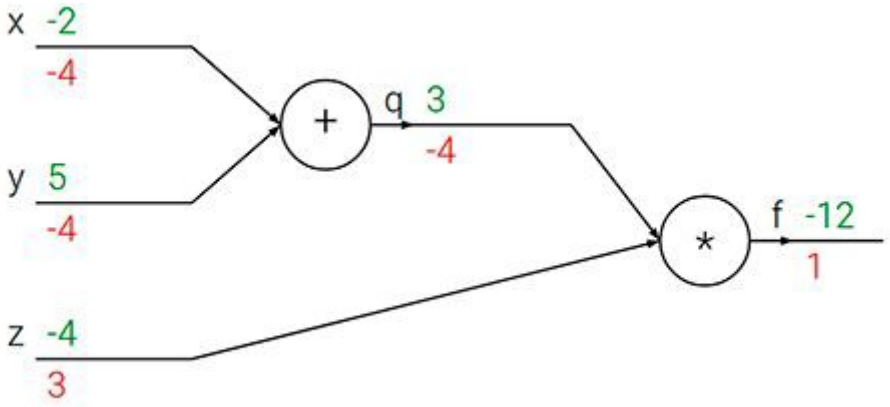
Recordatorio:

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/

$$f(x, y, z) = (x + y)z$$



- *Backward pass*:
  - El gradiente de f respecto a sí misma es 1.
  - Gradiente respecto a z:
    - f = qz, por tanto la derivada respecto a z es q. Sabemos que el valor de q es 3 por el forward pass, por tanto el gradiente en z es 3.
    - De manera similar, el gradiente en q es -4.

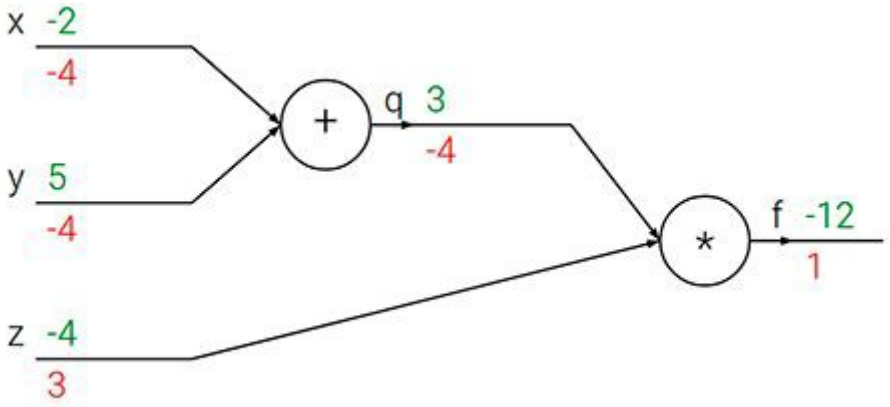
Recordatorio:

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/

$$f(x, y, z) = (x + y)z$$



- Gradiente de f respecto de x. Sabemos por la regla de la cadena 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 calcular: -4.
  - El gradiente de q respecto de x es 1.
  - Por tanto, el valor final del gradiente es -4.

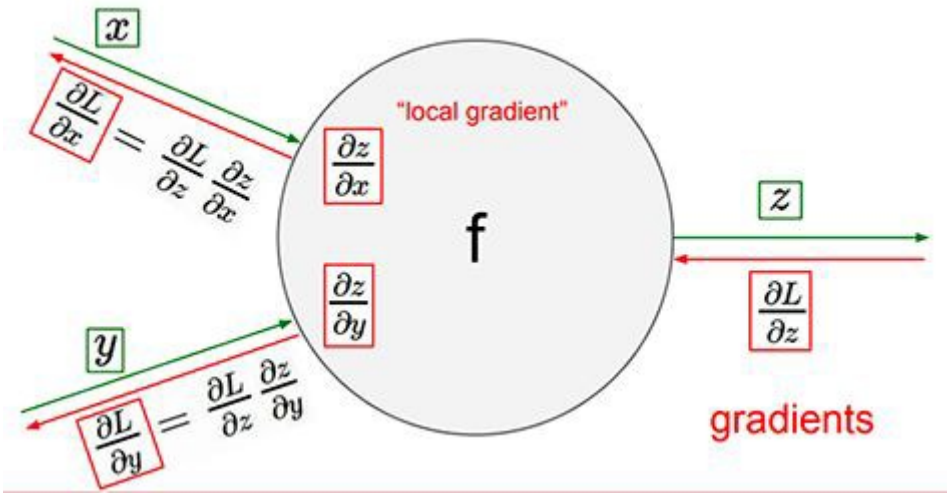
- De manera similar, el gradiente en y es -4 también.

Recordatorio:

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

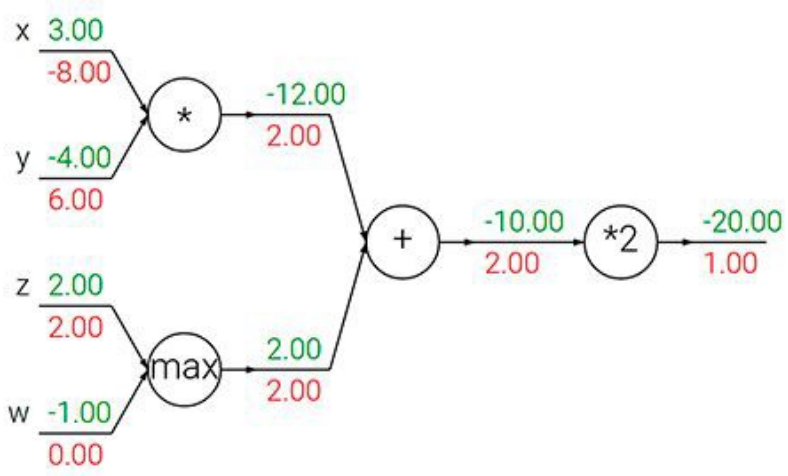
$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

Fuente de la imagen: http://cs231n.github.io/optimization-2/



- Backpropagation es un **proceso local**. Basta con saber el gradiente local de cada nodo con respecto a sus inputs y el valor del gradiente que se propaga hacia atrás desde el output.

- 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.



- **Suma:** Envía el gradiente de salida directamente hacia atrás a todos los *inputs*.
  - Consecuencia de que la derivada respecto a los inputs sea 1.

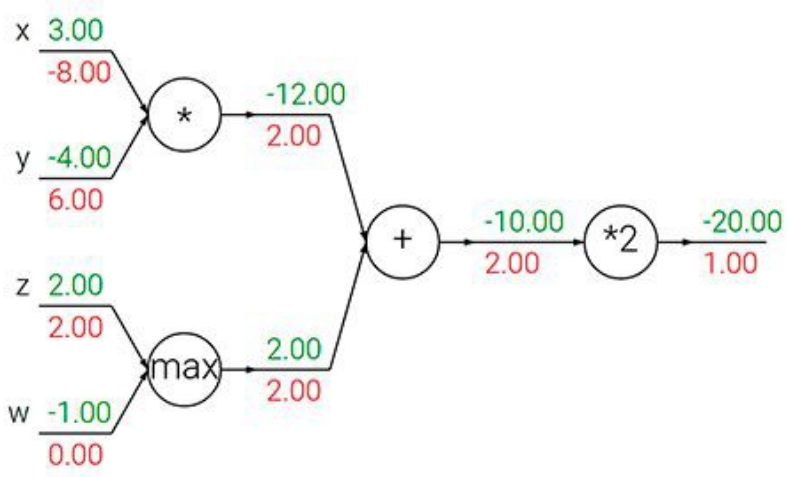
Recordatorio:

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

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/



- **Producto:** El gradiente de una variable de entrada viene dado por el producto del valor de la otra variable de entrada y el valor del gradiente de Salida. Es decir, para el nodo x el gradiente es -8. Esto es debido a que la entrada es -4 y la salida es 2.

Recordatorio:

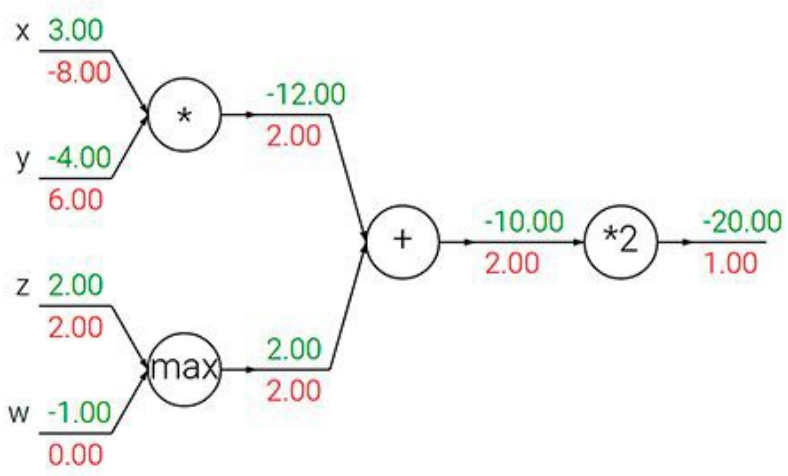
$$f(x,y) = \max(x,y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

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

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/



- **Máximo:** La variable con mayor valor se lleva todo el gradiente de salida, mientras que la variable de menor valor tiene gradiente 0 (el gradiente no fluye hacia atrás)

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

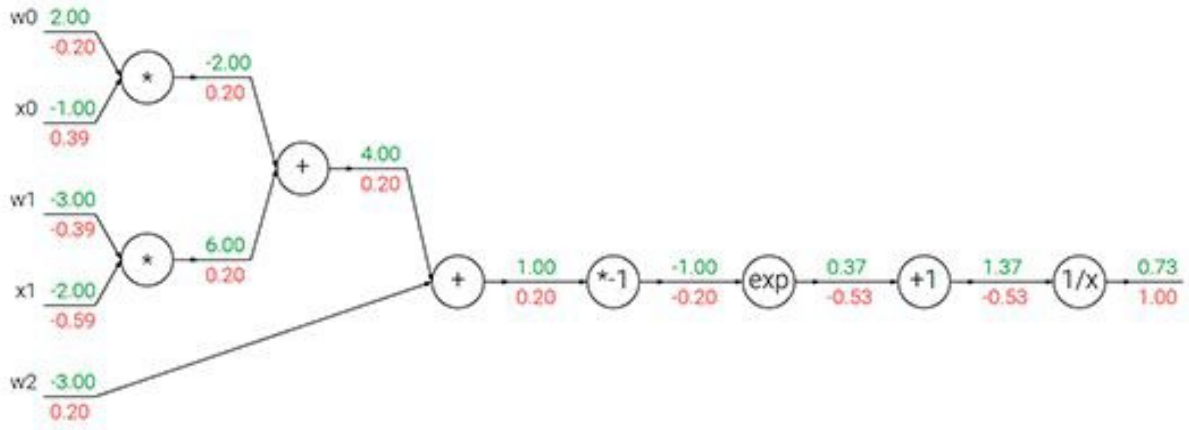
Recordatorio:

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

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

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/



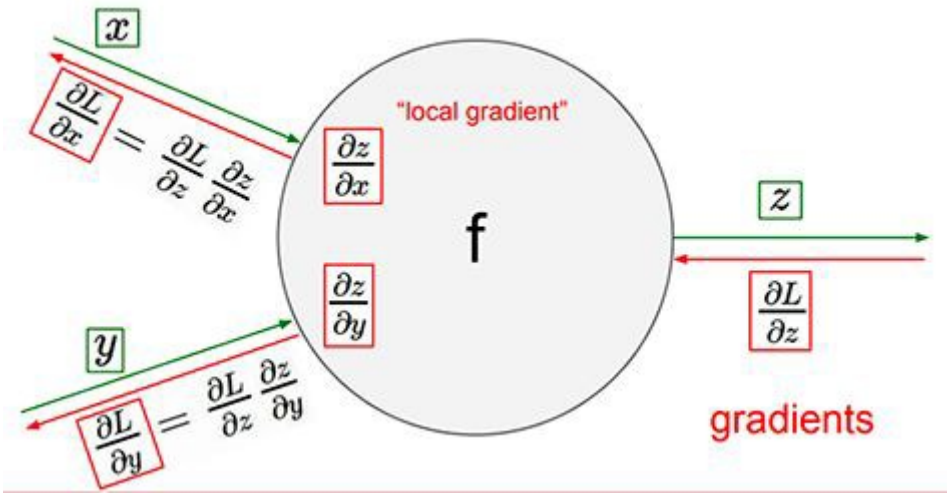
$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

- Así podríamos representar una neurona en un grafo de computación.
- No es necesario descomponer las funciones en sus mínimos componentes, muchas veces tenemos soluciones analíticas sencillas.

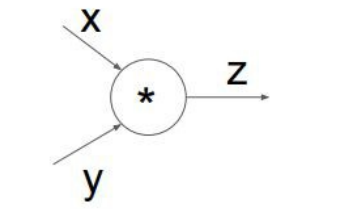
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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

Fuente de la imagen: http://cs231n.github.io/optimization-2/



- En definitiva, para aplicar el algoritmo de backpropagation, por cada operación o nodo a realizar necesitamos:
  - *Forward pass*: calcular el valor de salida del nodo.
  - *Backward pass*: obtener el cálculo de los gradientes de cada *input* a partir del gradiente propagado hacia atrás.



(x,y,z are scalars)

```python
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Local gradient

Upstream gradient variable

- Para aplicar el algoritmo de backpropagation, por cada operación o nodo a realizar necesitamos:
  - *Forward pass*: calcular el valor de salida del nodo.
  - *Backward pass*: obtener el cálculo de los gradientes de cada *input* a partir del gradiente propagado hacia atrás.

Entrenamiento de una red neuronal

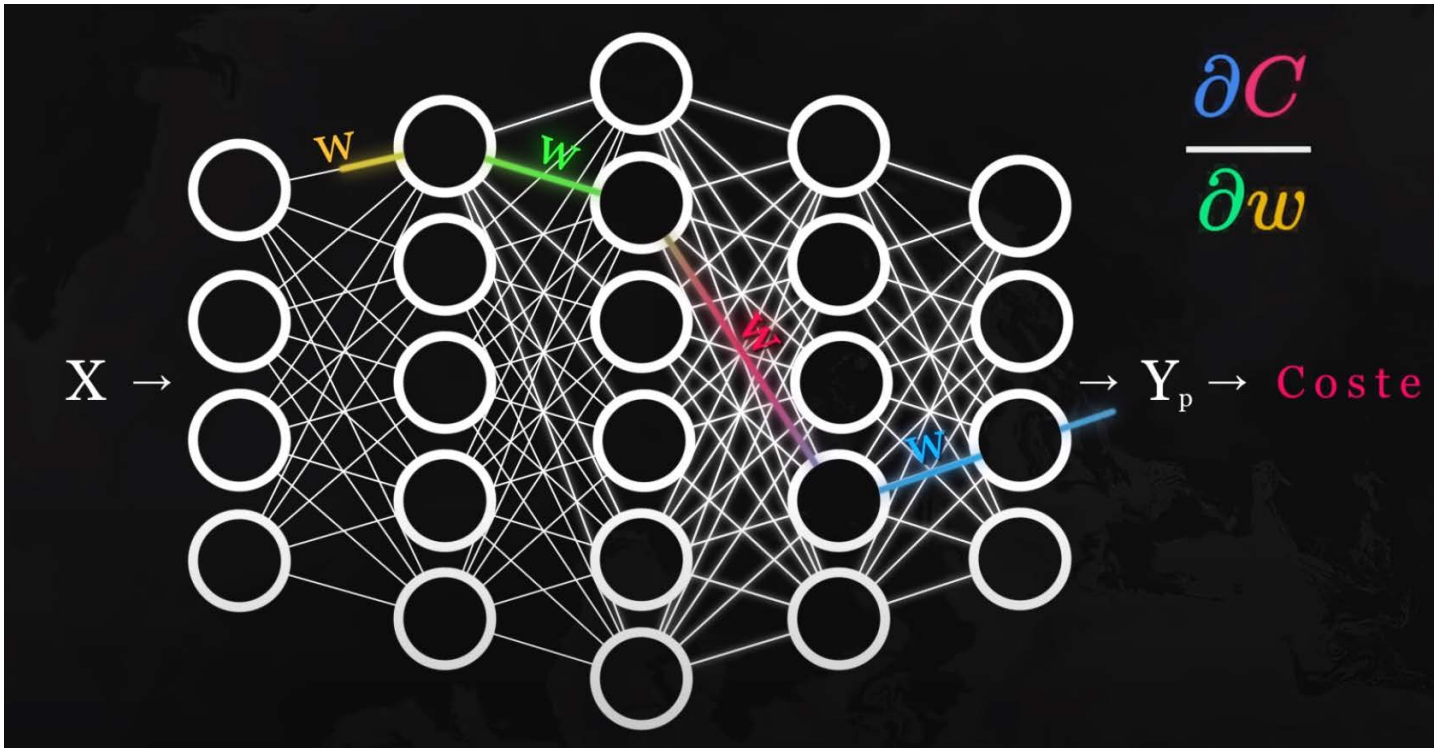
- Para entrenar la una red neuronal se usa el descenso del gradiente.
- Solo hay que contestar como varia la función de coste respecto de los parámetros (bias y pesos)

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

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

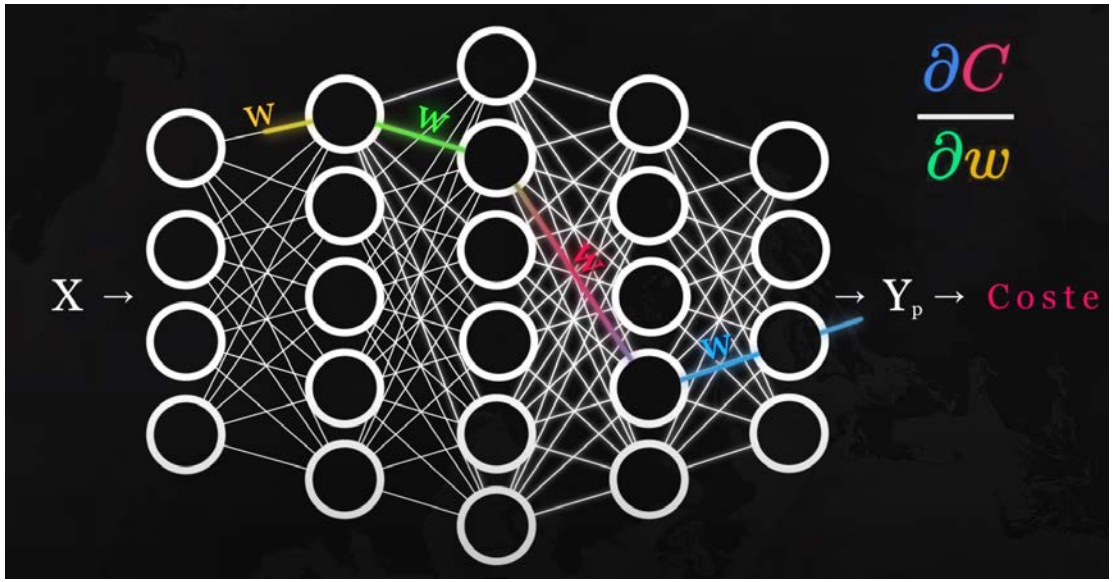
Entrenamiento de una red neuronal

- Minimizar la función de coste, si, pero… ¿como de complejos son los cálculos?



Entrenamiento de una red neuronal

- Reducir la función de coste, si, pero… ¿como de complejos son los cálculos?
- Como podéis imaginar, cada una de estas derivadas va a depender de los pesos en cada una de las capas complicando la situación bastante.

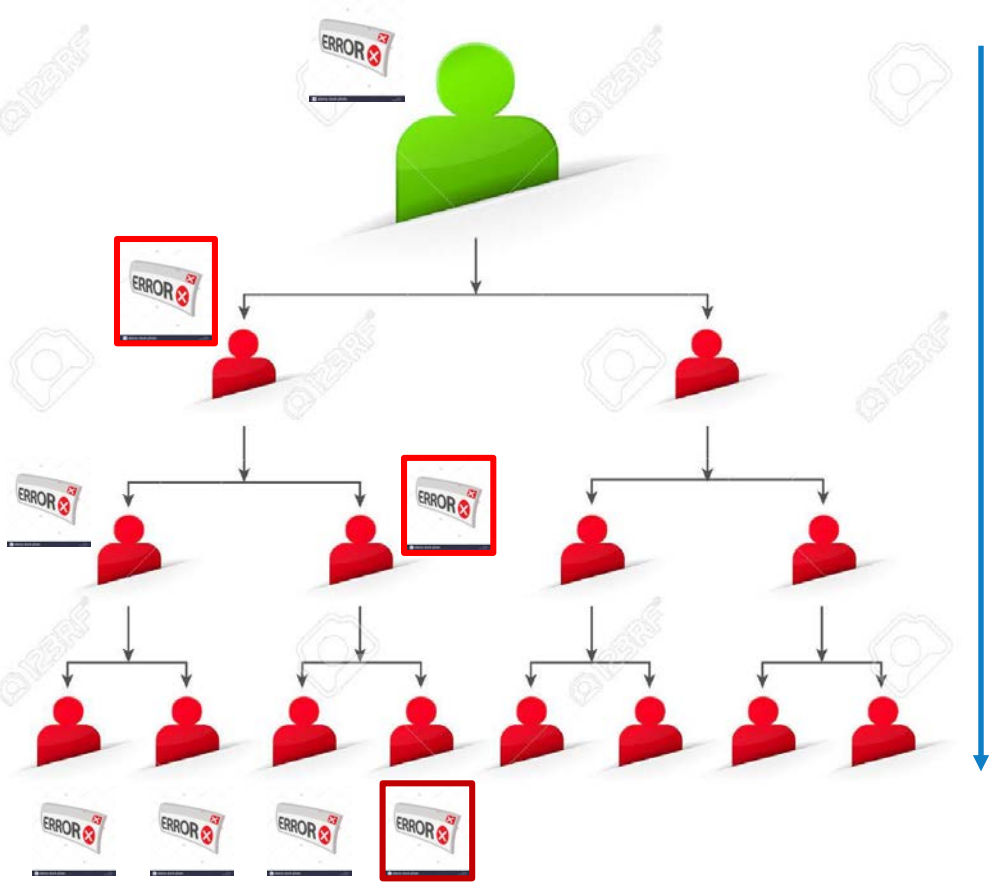


Entrenamiento de una red neuronal

- Entonces…¿Cuál es la solución a todos los problemas que tenemos respecto a como calcular el gradiente de una red neuronal teniendo en cuenta todos los parámetros que influyen?

Idea backpropagation

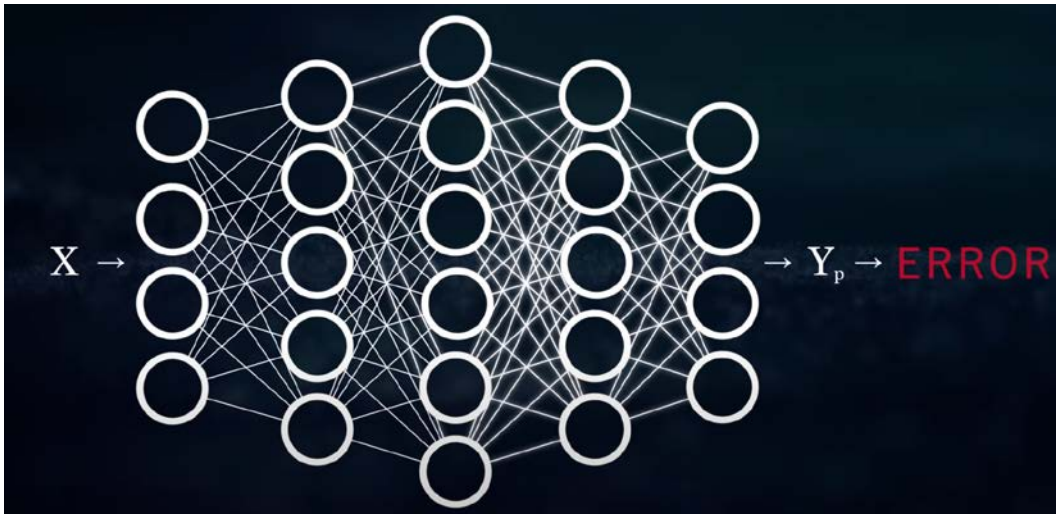
- Pensado en un esquema del personal de una empresa



Búsqueda de la responsabilidad en el error

Matemáticas del backpropagation

- Cuando entrenamos una red neuronal por primera vez, el 100% de las veces tendremos que el resultado es un error ya que el primer valor que vamos a obtener es aleatorio
- Es decir, función de coste con un error muy elevado



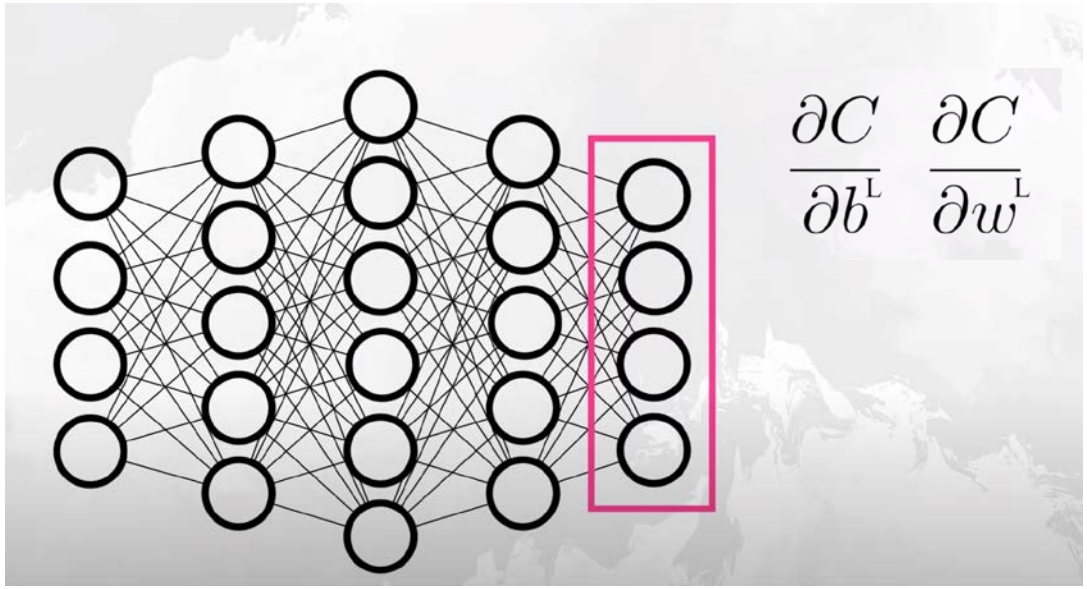
Matemáticas del backpropagation

- Recordad que para el calculo del gradiente teníamos dos tipos de derivadas:

$$\frac{\partial C}{\partial w} \qquad \frac{\partial C}{\partial b}$$

Matemáticas del backpropagation

- Vamos a empezar a calcular las derivadas de la ultima capa e iremos hacia atrás calculando esas derivadas



L = número de capas

Matemáticas del backpropagation

- Derivadas de la última capa:
  - Primer de las neuronas (suma ponderada)

$$Z^L = W^L X + b^L$$

  - Luego se pasa por la función de activación

$$a(Z^L)$$

  - Función de coste

$$C(a(Z^L)) = ERROR$$

- De esta forma llegamos a calcular el error de la red si solo tenemos en cuenta la ultima capa.

Matemáticas del backpropagation

- Por tanto las derivadas de los pesos y de las bias es:

Regla de la cadena!!

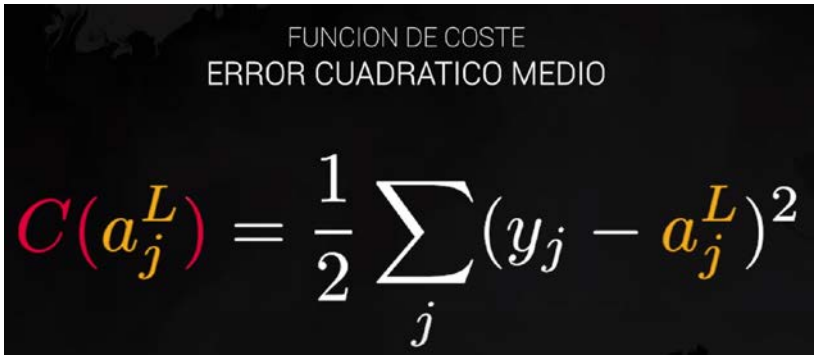
$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}$$

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L}$$

$$Z^L = W^L a^{L-1} + b^L \qquad C(a^L(Z^L))$$

Matemáticas del backpropagation

- Derivada del coste respecto de la activación

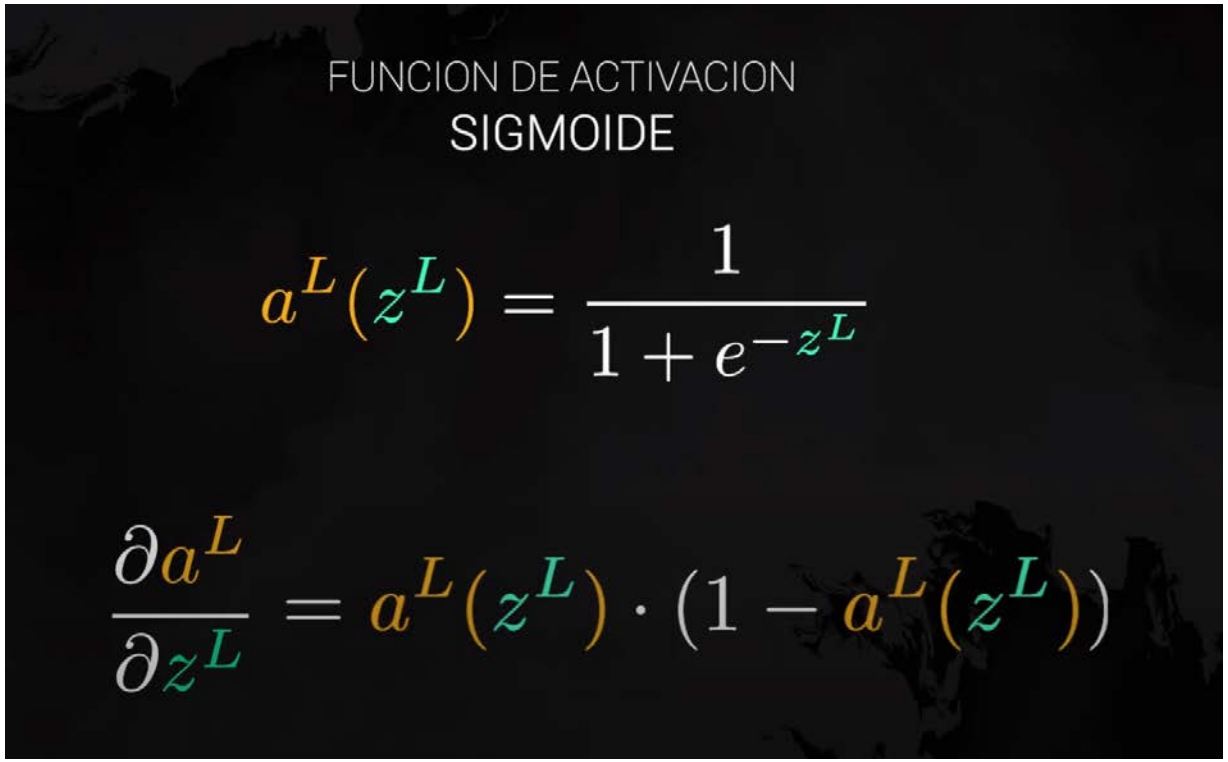


- Derivando:

$$\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$$

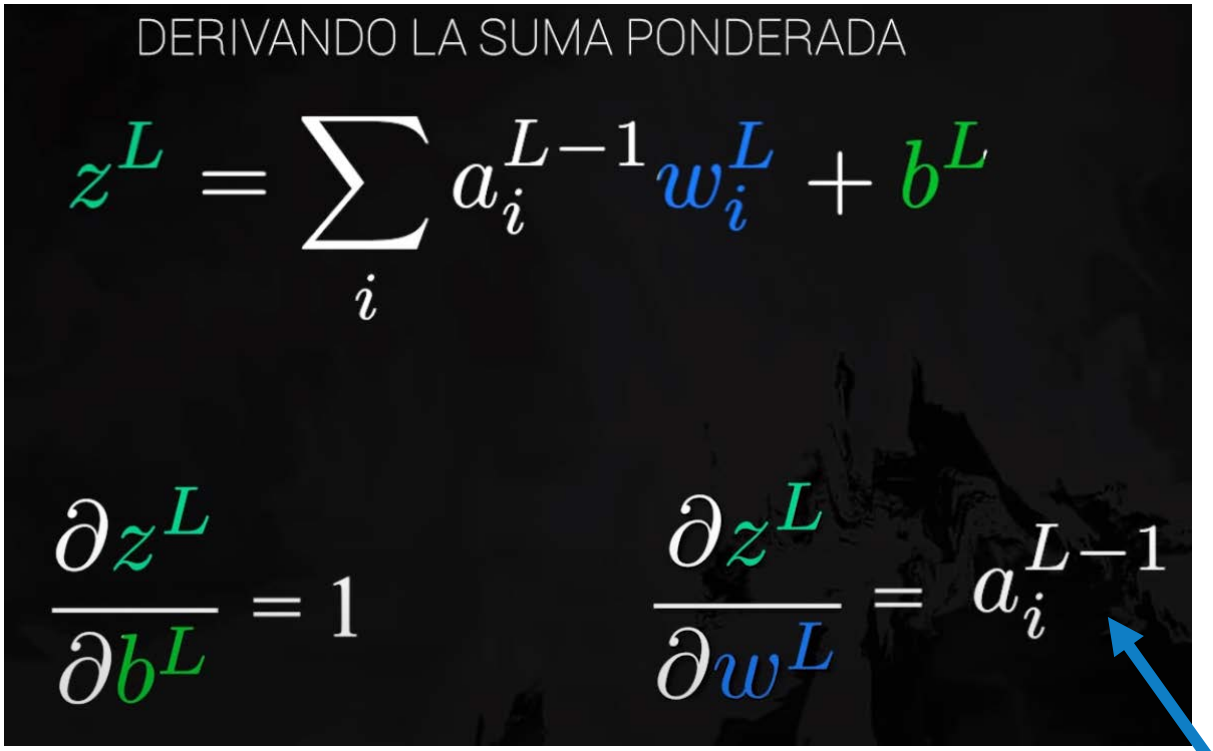
Matemáticas del backpropagation

- Derivada de la salida de la neurona respecto de la función de activación (sigmoide para el ejemplo)



Matemáticas del backpropagation

- Derivada de la salida de la neurona respecto de los pesos y de las bias



Salida de la neurona L-1

Matemáticas del backpropagation

- Resumen:

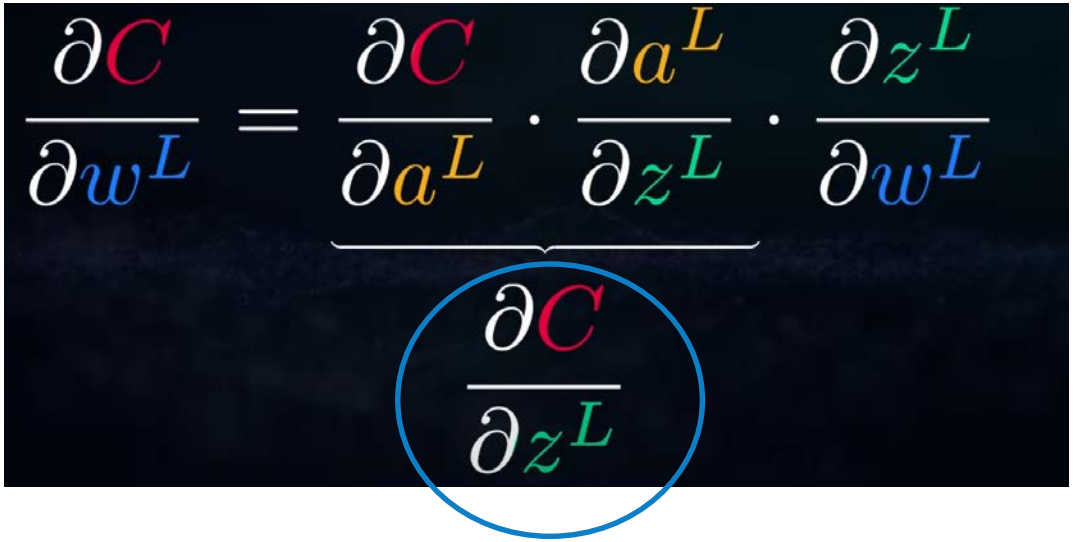
$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}$$

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L}$$

$$\frac{\partial a^L}{\partial z^L} = a^L(z^L) \cdot (1 - a^L(z^L)) \qquad \frac{\partial C}{\partial a_j^L} = (a_j^L - y_j) \qquad \frac{\partial z^L}{\partial b^L} = 1 \qquad \frac{\partial z^L}{\partial w^L} = a_i^{L-1}$$

Matemáticas del backpropagation

- Operaciones matemáticas:



  - Si esta derivada es grande, quiere decir que el error afecta mucho a la función de coste, si es pequeña afecta poco a la función de coste.
  - Si usamos el ejemplo de la empresa:
    - La responsabilidad de la neurona en el error final!

Matemáticas del backpropagation

- Por tanto, el error de la capa L (ultima capa):

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L}$$

$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}$$

Matemáticas del backpropagation

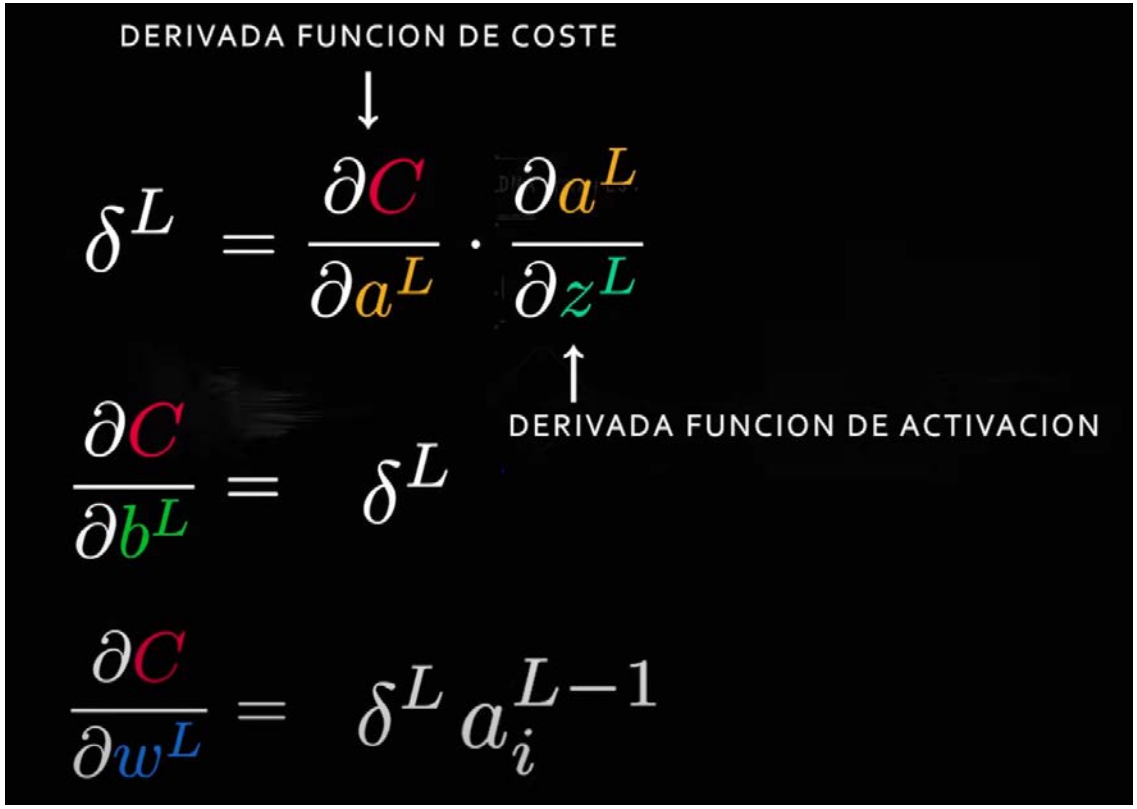
- Para simplificar los cálculos, este derivada se representa con este símbolo

$$\frac{\partial C}{\partial b^L} = \delta^L \cdot \frac{\partial z^L}{\partial b^L}$$

$$\frac{\partial C}{\partial w^L} = \delta^L \cdot \frac{\partial z^L}{\partial w^L}$$

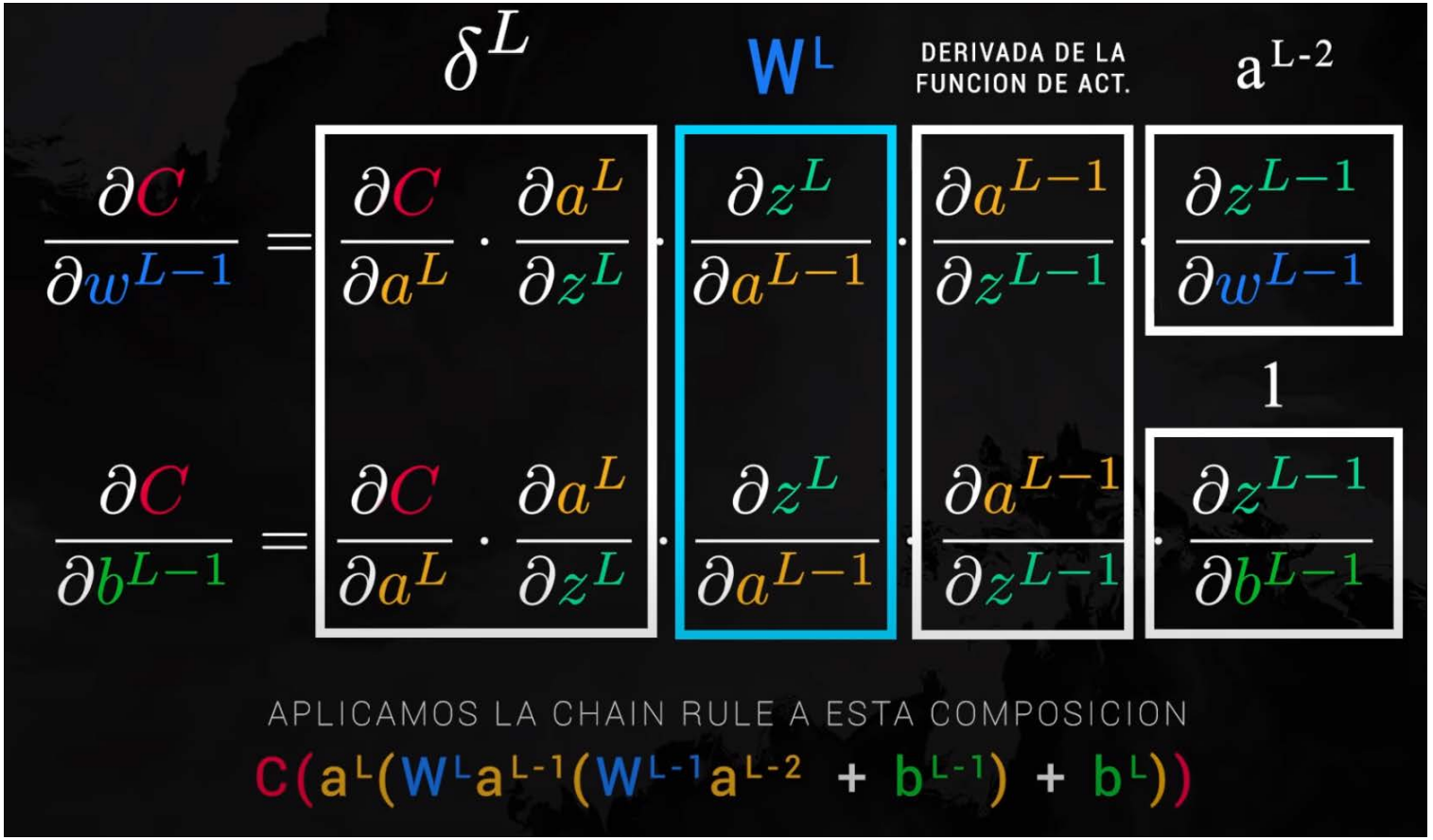
Matemáticas del backpropagation

- Por tanto, las derivadas de la función de coste respecto de las bias y de los pesos, es:



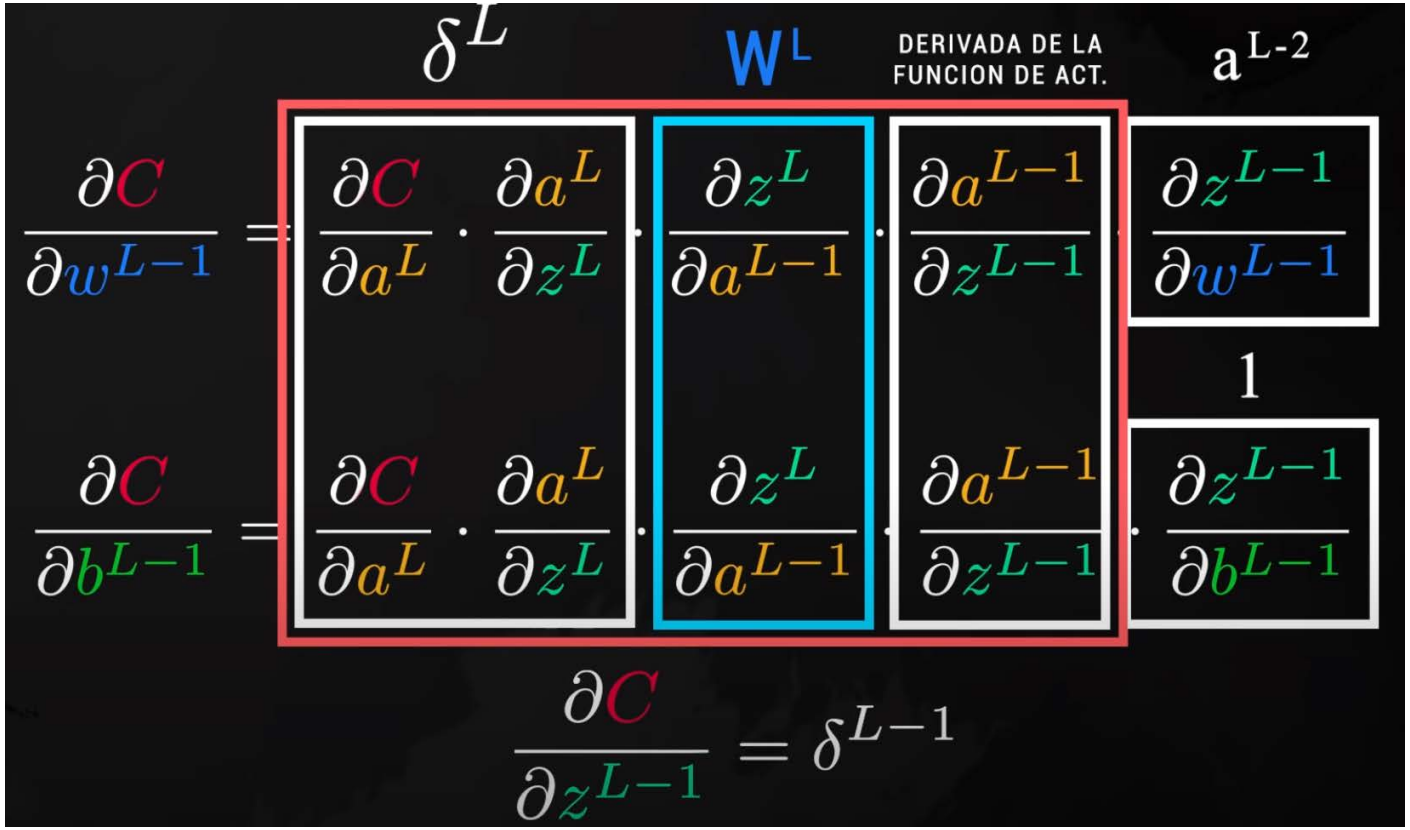
Matemáticas del backpropagation

- Y el resto de capas, ¿Cómo se calcula?



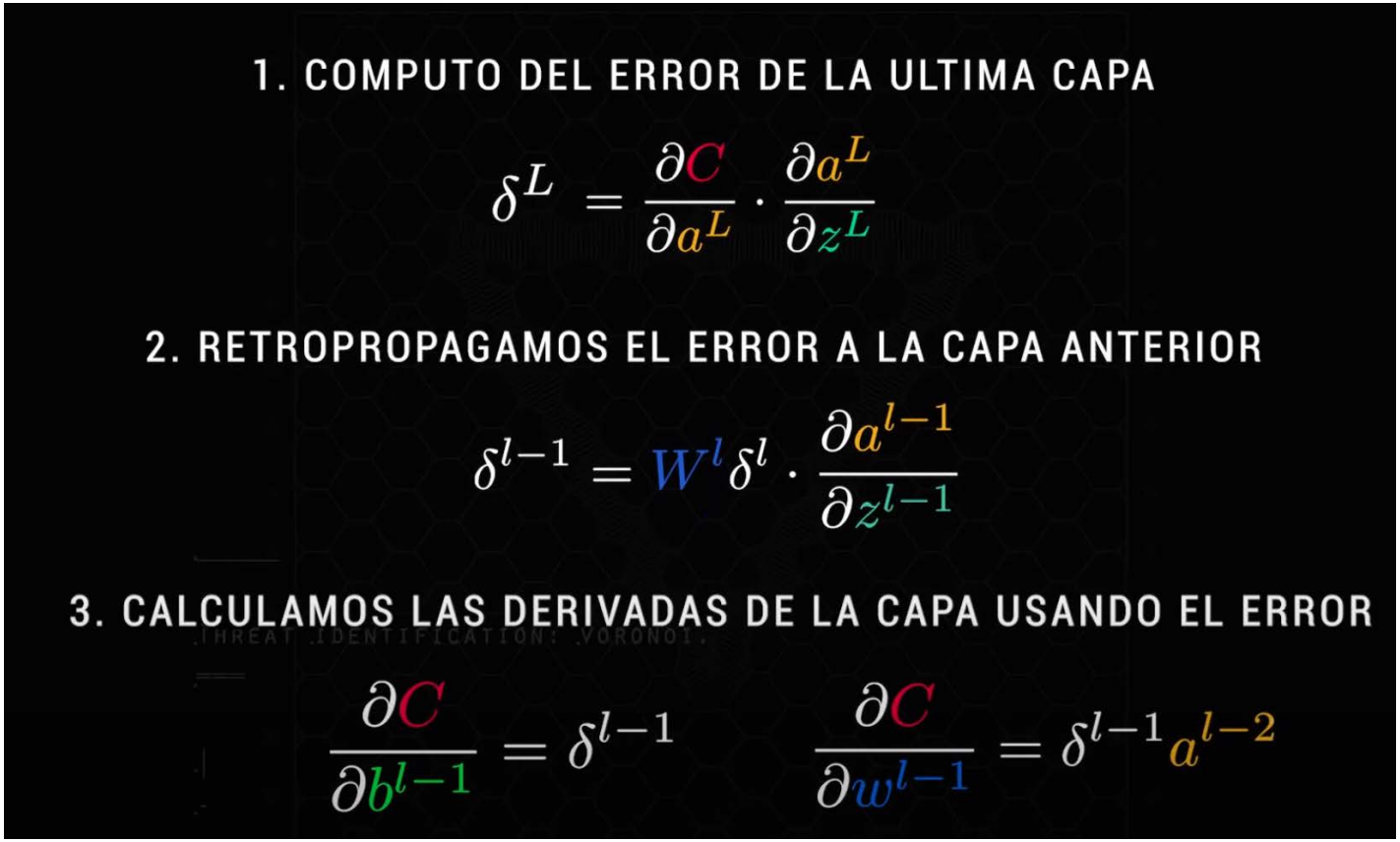
Matemáticas del backpropagation

- Entonces, por recurrencia:



Matemáticas del backpropagation

- En resumen este proceso de recurrencia sigue este esquema:



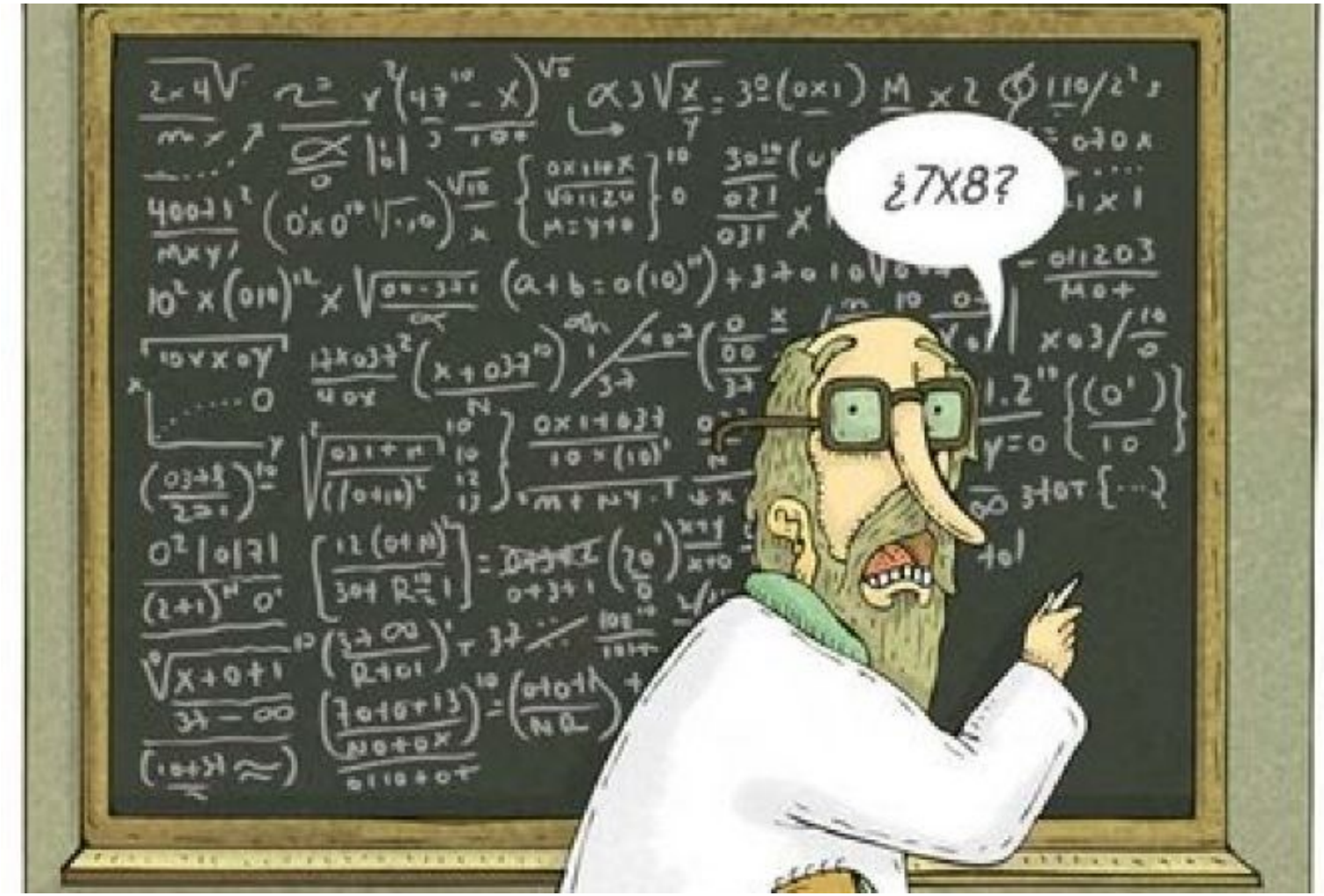
Recordatorio tema 2

- Para entrenar una neurona o una red neuronal, hemos visto los siguientes e IMPORTANTES métodos:
  - Función de coste
  - Algoritmo del gradiente descendente
  - Algoritmo de gradiente descendente estocástico
  - Retropropagación

Actividad

- Vamos a revisar la actividad por encima, las dudas si no las contesto ni en clase ni en el foro es porque no os voy a resolver la actividad.

¿Dudas?



www.unir.net