



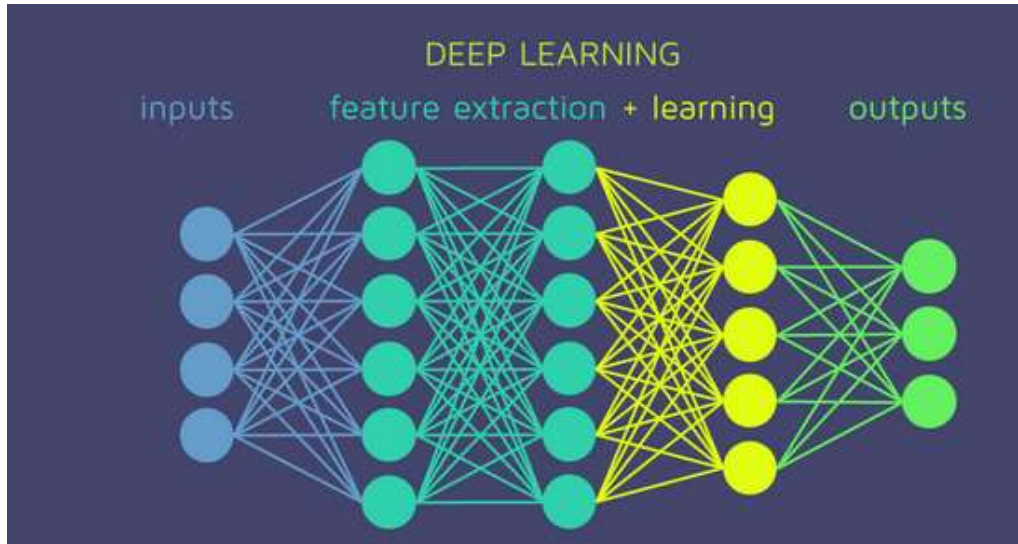
# UT5. Entrenamiento redes neuronales.

# Bloques de la unidad:



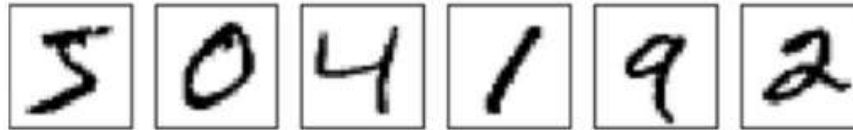
1. Funciones de coste.
2. Entrenamiento con gradient descent.
3. Stochastic Gradient Descent (SGD).
4. Laboratorio 1.
5. Backpropagation.
6. Ejercicios regla cadena.
7. Unidades de activación.
8. Inicialización de parámetros.
9. Batch normalization.
10. Optimización avanzada.
11. Regularización.
12. Laboratorio 2.

# 1. Funciones de coste:



# 1. Funciones de coste:

- 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.

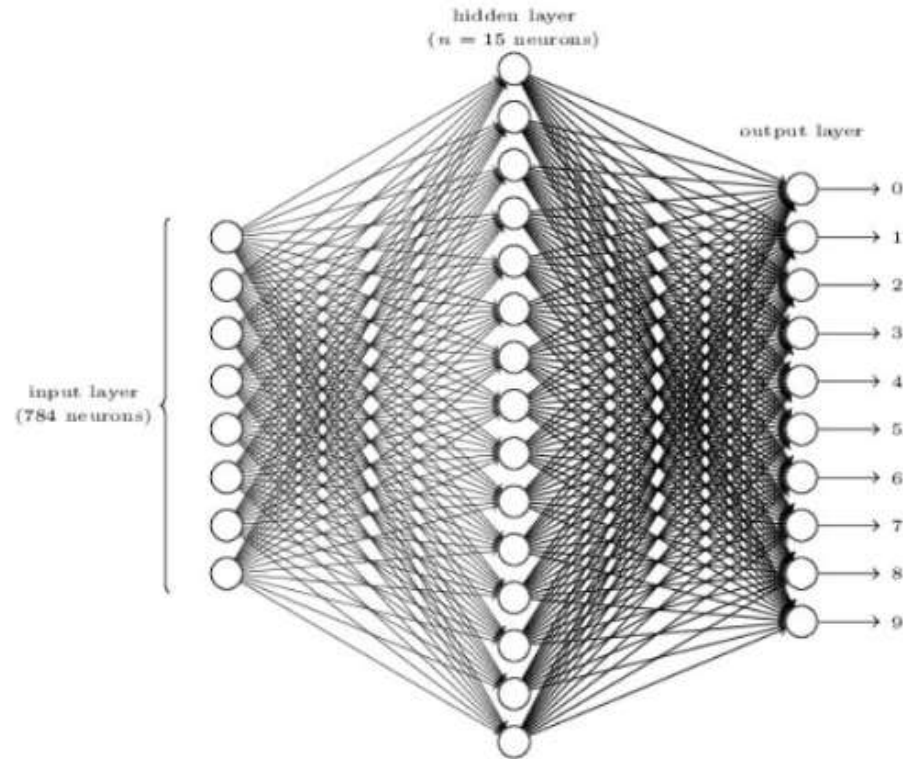


# 1. Funciones de coste:



- El tamaño del dataset es de 60.000 elementos de training data y 10.000 elementos de test data.
- Tenemos por tanto un problema de clasificación con 10 clases: los números del 0 al 9.

# 1. Funciones de coste:



# 1. Funciones de coste:



## 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.

# 1. Funciones de coste:



## Hidden layer:

- 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.



# 1. Funciones de coste:



## 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.

# 1. Funciones de coste:



Vamos a intentar intuir 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.

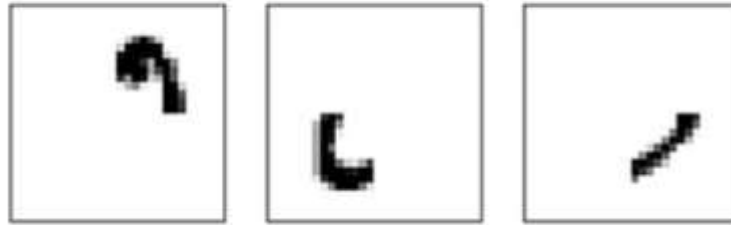
# 1. Funciones de coste:



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.

# 1. Funciones de coste:

Ejemplo de representación de una imagen en la hidden layer.



# 1. Funciones de coste:



¿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 **hiperparámetro**

# 1. Funciones de coste:



**$\mathbf{x}$ :** input o training example (en nuestro caso, vector de 784 píxeles).

**$\mathbf{y}(\mathbf{x})$ :** valor deseado de salida de la red, representado como vector de 10 componentes.

Ej:  $\mathbf{y}(\mathbf{x}) = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$  representa la salida deseada de una imagen que es un 3.

# 1. Funciones de coste:



**$y(x)$**  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.

# 1. Funciones de coste:



**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.




# 1. Funciones de coste:



Necesitamos cuantificar de algún modo cómo de buena es la aproximación de  $\mathbf{a}(\mathbf{x}, \mathbf{w}, \mathbf{b})$  a  $\mathbf{y}(\mathbf{x})$ . Para ello definimos la función de coste o cost function

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

# 1. Funciones de coste:


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

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

# 1. Funciones de coste:



Estamos obteniendo el cuadrado de la distancia vectorial, un número siempre **positivo**, entre el **valor real de salida** y la **salida de nuestra red**.

- ¿ Qué es  $y(x)$  ?
- ¿ Qué es  $x$  ?

## 1. Funciones de coste:

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

Ejemplo

$a(x, w, b) = (0.9, 0.1, 0.1, 0.2, 0, 0, 0, 0, 0, 0)$


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

■ Para este ejemplo, el error sería:

$$(1 - 0.9)^2 + (0 - 0.1)^2 + (0 - 0.1)^2 + (0 - 0.2)^2 + \dots$$

Sumamos los errores de todos los datos=  $c(w,b)=2,27$

# 1. Funciones de coste:


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

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  $C(w, b)$ .

Ejemplo de aproximación correcta:

- $a(x, w, b) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$
- $c(w, b) = 0$

# 1. Funciones de coste:



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**.

# 1. Funciones de coste:



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.

# 1. Funciones de coste:

## REPASO

- 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.




# 1. Funciones de coste:

Entrenar la red neuronal se convierte en un problema de optimización: Queremos encontrar los valores de los parámetros  $w$  y  $b$  que minimizan  $C$ .



## 2. Entrenamiento con gradient descent:

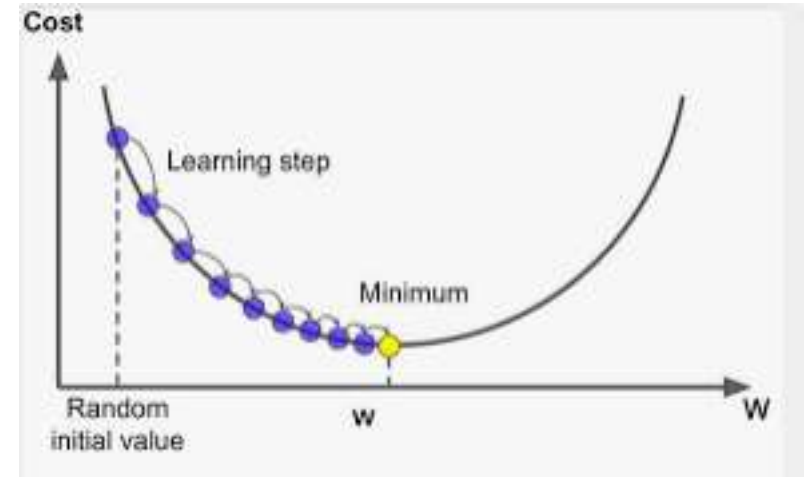

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

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 calcular 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.

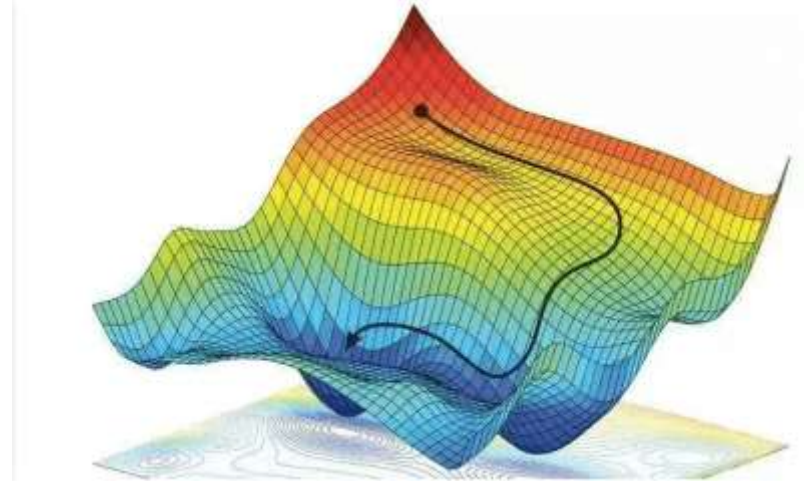
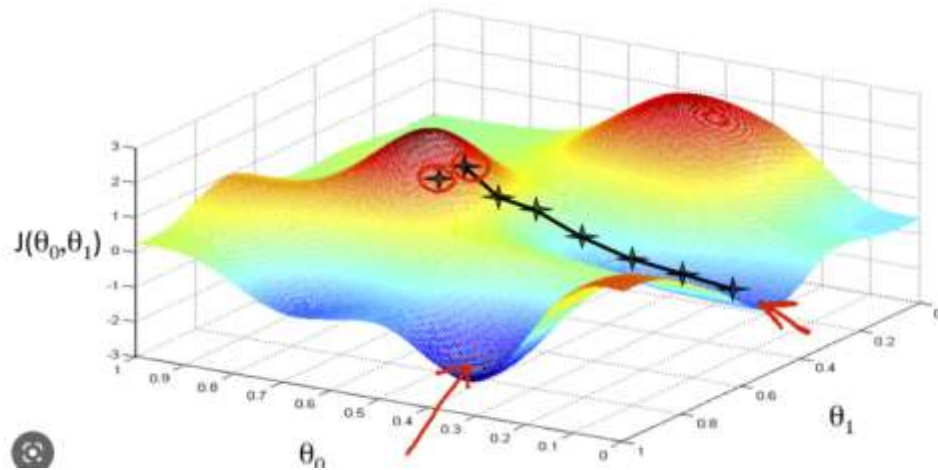
## 2. Entrenamiento con gradient descent:

- 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.



## 2. Entrenamiento con gradient descent:

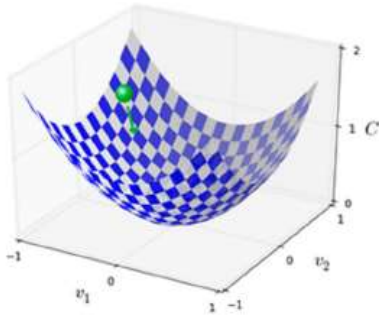
Se puede complicar la base matemática



## 2. Entrenamiento con gradient descent:

Se puede complicar la base matemática

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



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

## 2. Entrenamiento con gradient descent:



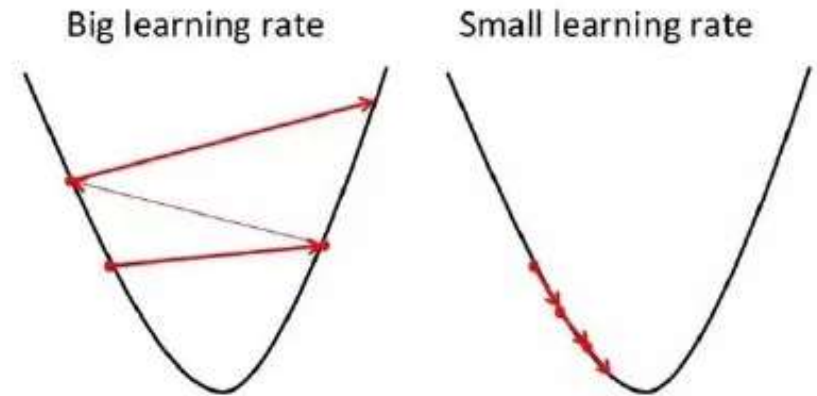
Pero a nosotros sólo nos va a preocupar para optimizar el **learning rate**.

El learning rate  $\eta$ , 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.

## 2. Entrenamiento con gradient descent:

Si  $\eta$  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.



## 2. Entrenamiento con gradient descent:



### Resumen de conceptos vistos hasta ahora.

Input layer.

Hidden layer.

Número de capas ocultas.

Output layer.

Función de coste.

Loss function.

Mean Squared Error (MSE)

Cross-entropy loss.

Gradient descent.

Learning rate



### 3. 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.

### 3. Stochastic Gradient Descent (SGD)



Elegimos **m** examples del dataset:  $X_1, X_2, \dots, X_m$ . 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. (Como de grandes son mis encuestas)

**¿ Por qué hacemos más operaciones ?**

### 3. 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
```

### 3. Stochastic Gradient Descent (SGD)



Ejemplo:

- a) Un dataset de 10.240 elementos con un mini-batch de 512. ¿Cuántos ajustes de gradientes realizaremos por cada época ?
- b) Y si no utilizáramos SGD. ¿Cuántos ajustes de gradientes realizaremos por cada época ?

### 3. Stochastic Gradient Descent (SGD)

¿Cuál es el máximo número de elementos en el conjunto de entrenamiento ?

```
[12] batch_size= 64
     epochs = 20
     history = model.fit(x_train, y_train,
                        batch_size = batch_size,
                        epochs = epochs,
                        verbose=1,
                        validation_data=(x_test, y_test))
     score = model.evaluate(x_test, y_test, verbose=0)
```

```
Epoch 1/20
938/938 [=====] - 16s 15ms/step - loss: 2.1860 - accuracy: 0.4158 - val_loss: 2.0265 - val_accuracy: 0.5128
Epoch 2/20
938/938 [=====] - 4s 5ms/step - loss: 1.7984 - accuracy: 0.5721 - val_loss: 1.5722 - val_accuracy: 0.6243
Epoch 3/20
938/938 [=====] - 5s 5ms/step - loss: 1.4006 - accuracy: 0.6170 - val_loss: 1.2662 - val_accuracy: 0.6489
_ _ _ _ _
```

## 4. Laboratorio



Normalmente al entrenar una red neuronal dividimos los datos en **training data** y **validation data**.

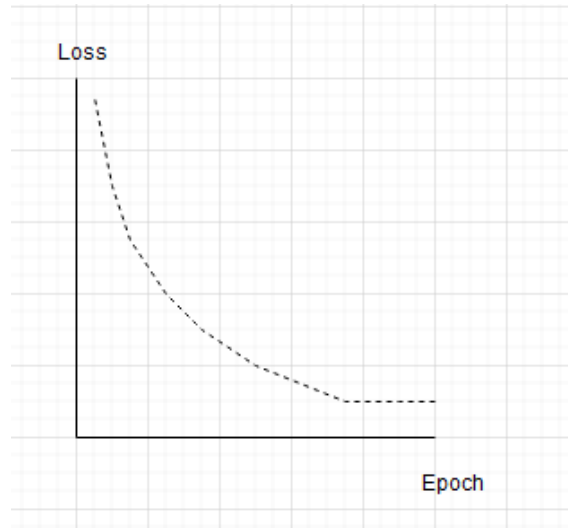
Los test data los utilizaremos para tunear los hiperparámetros a partir de la estimación final de las métricas de nuestro modelo.

Generalmente:

- 70 % Train.
- 20 % Validation.
- 10 % Test.

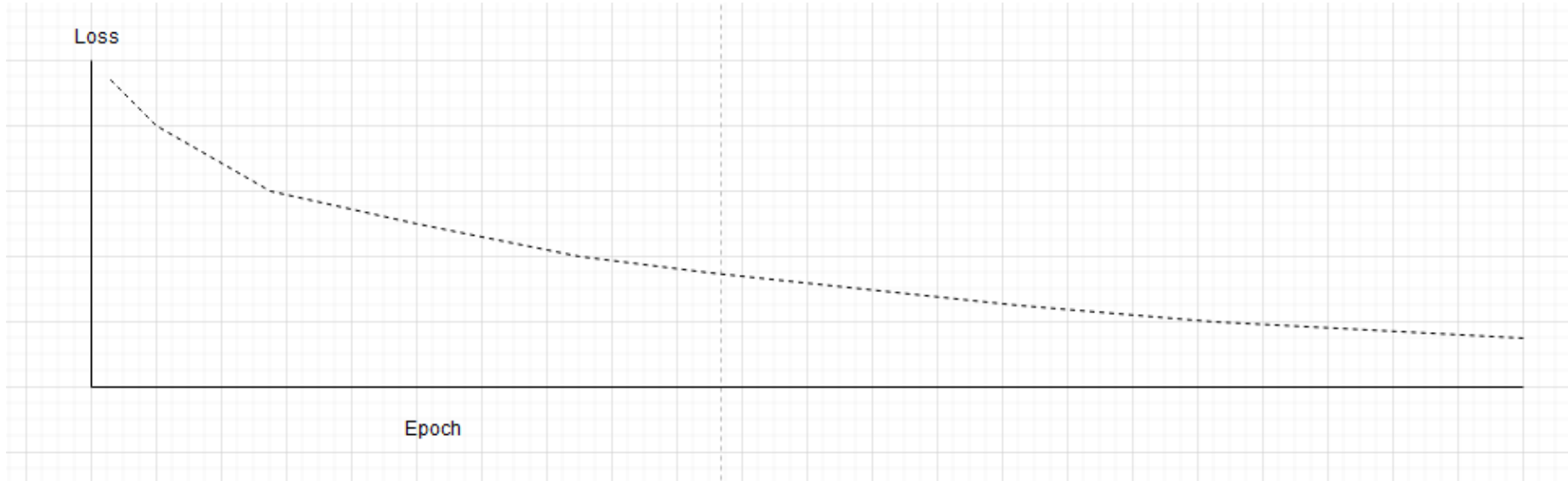
## 4. Laboratorio

Durante el entrenamiento es muy importante seguir las métricas, en especial la **loss**, para verificar que todo es correcto.



## 4. Laboratorio

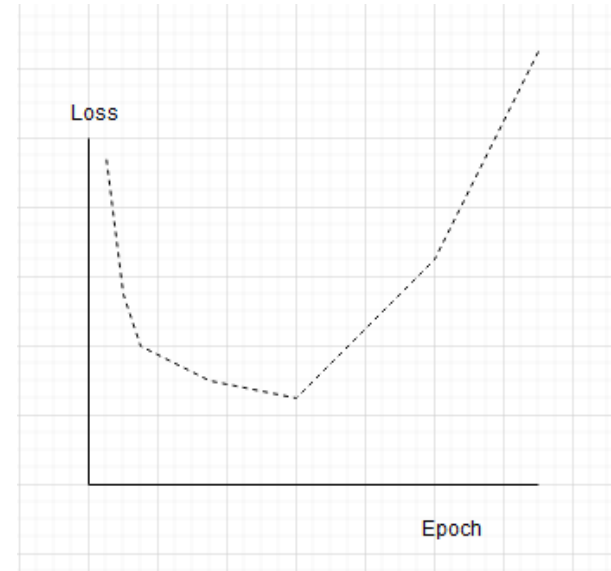
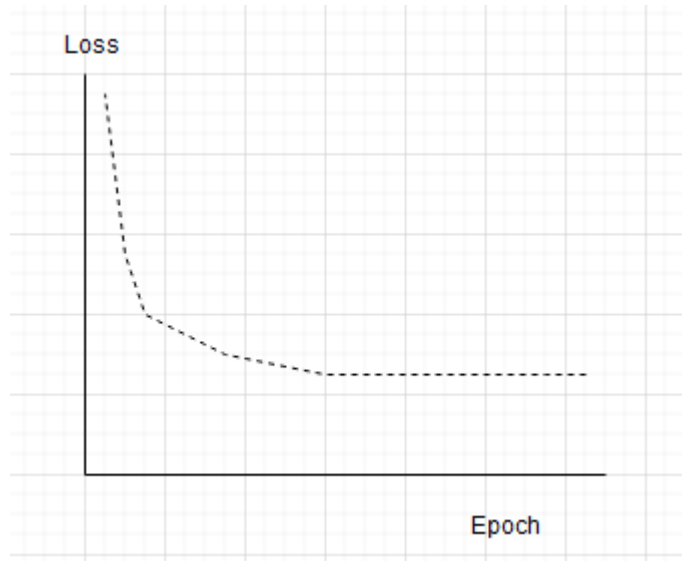
Si tenemos una learning rate baja. Necesita muchas Epoch (Bien escaso)





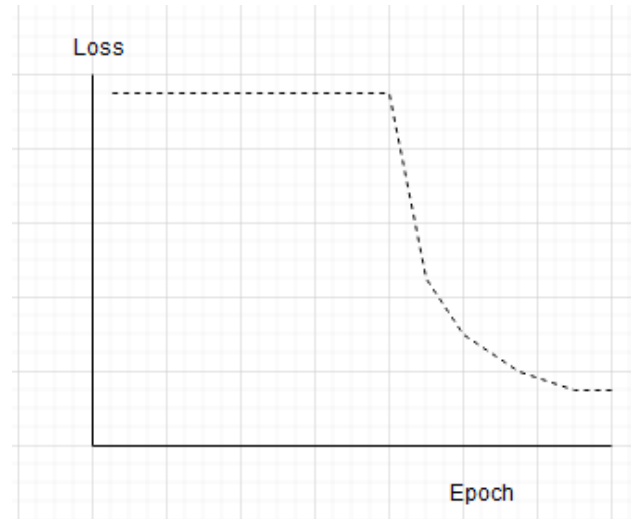
## 4. Laboratorio

Si tenemos una learning rate alta. No llegamos a una solución óptima.



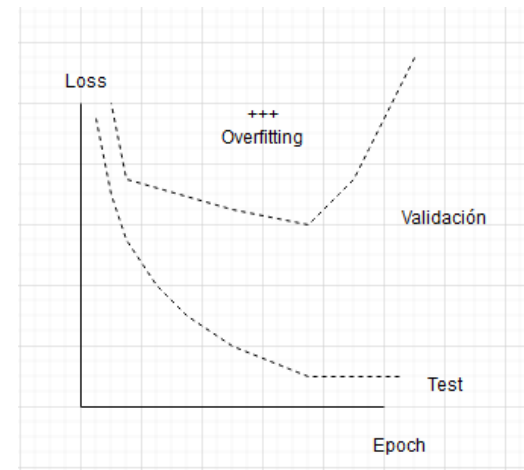
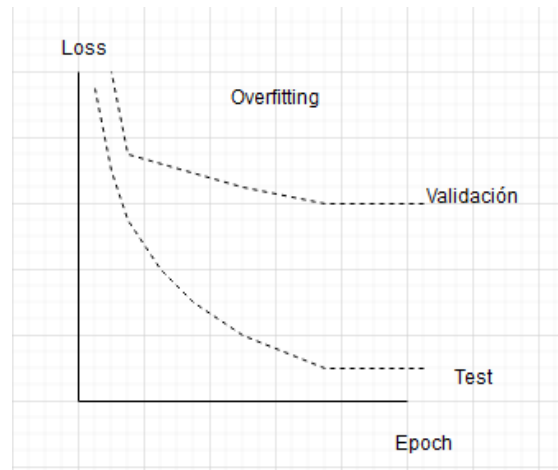
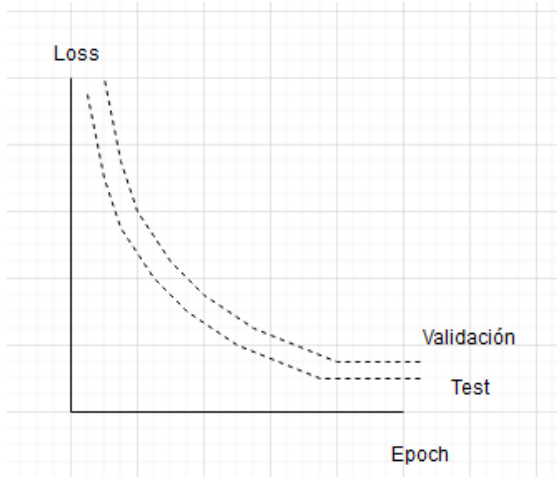
## 4. Laboratorio

Caso especial: Es muy probable que esto se deba a la inicialización de los parámetros. Lo veremos en la segunda parte del tema.



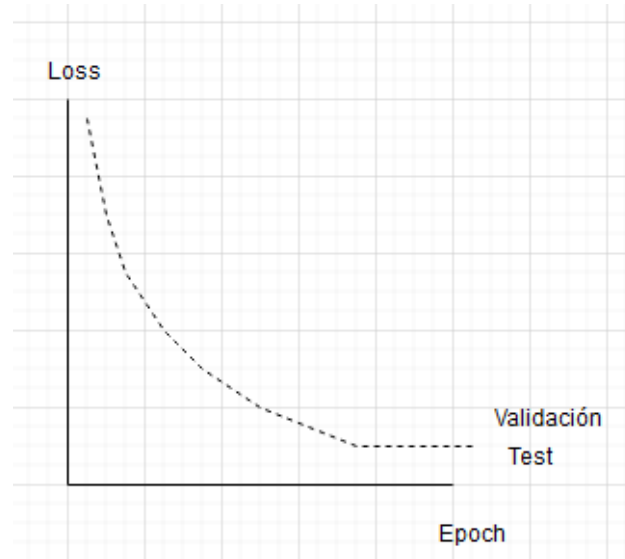
## 4. Laboratorio

### La función de pérdida ayuda al overfitting



## 4. Laboratorio

Por último si se solapan nos quiere decir que hacen falta más capas.



## 4. Laboratorio



Es muy importante que relacionemos los vistos en clase.

Input layer.

Hidden layer.

Número de capas ocultas.

Output layer.

Función de coste.

Loss function.

Mean Squared Error (MSE)

Cross-entropy loss.

Gradient descent.

Learning rate

Batch / Epoch

SGD

## 4. Laboratorio



Ahora vamos a realizar el Colab: Entrenamiento\_Redes\_Neuronales.ipynb

## 5. Backpropagation



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 estos gradientes.

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.

## 5. Backpropagation



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.



## 5. Backpropagation

Resumen hasta ahora:

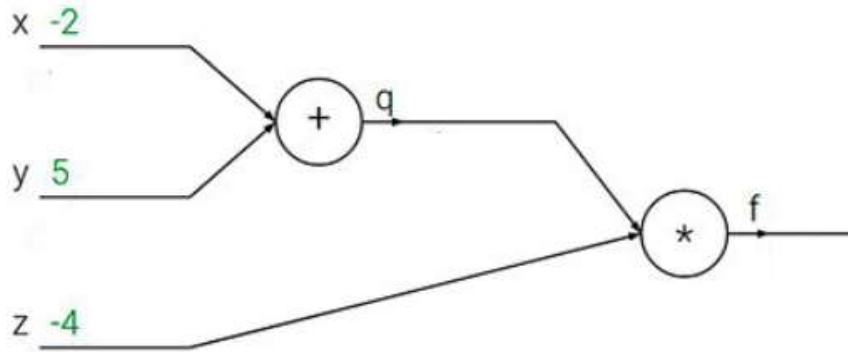
- Calcular los gradientes es una tarea muy compleja incluso en los tiempos que vivimos donde disponemos infinidad de GPU.
- Recordar que utilizamos GPU en vez de CPU porque básicamente están compuestas por ALU's en paralelo.
- Para solucionar este problema utilizamos la regla de cadena.



## 5. Backpropagation

Nuestras redes neuronales van a ser grafos de computación, Tensorflow por dentro está formado por estos.

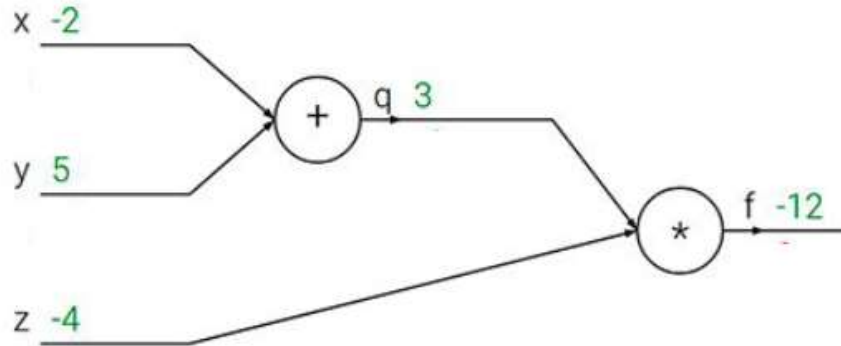
Ejemplo de backpropagation  $f(x, y, z) = (x + y)z$



## 5. Backpropagation

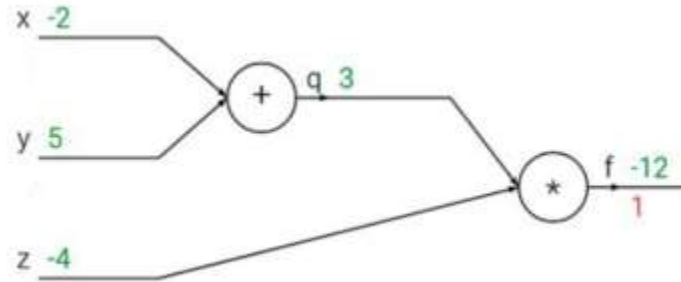
Primero: **Forward pass**. Calculamos las salidas de cada nodo a partir de sus inputs.

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



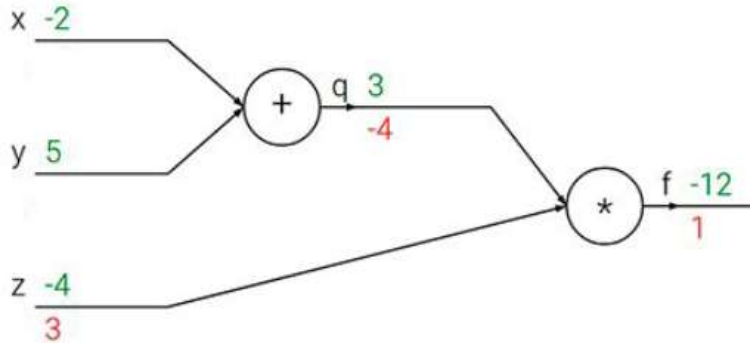
## 5. Backpropagation

El gradiente de  $f$  respecto a sí misma es 1.



## 5. Backpropagation

**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.

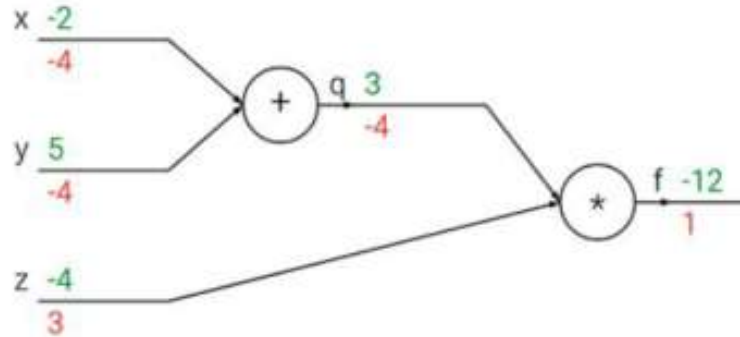


$$1 * (-4) = -4$$

$$1 * 3 = 3$$

## 5. Backpropagation

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



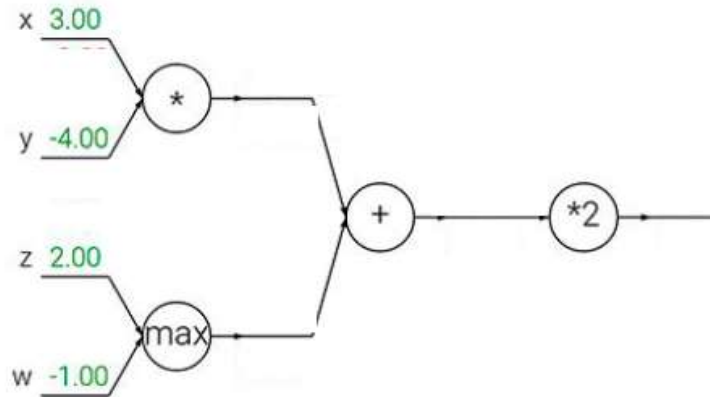
## 5. Backpropagation



Otro patrón que se encuentra es el **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).

## 5. Backpropagation

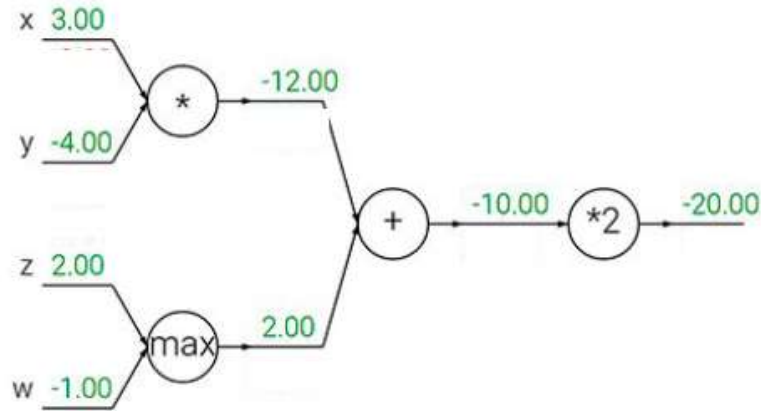
Ejemplo:





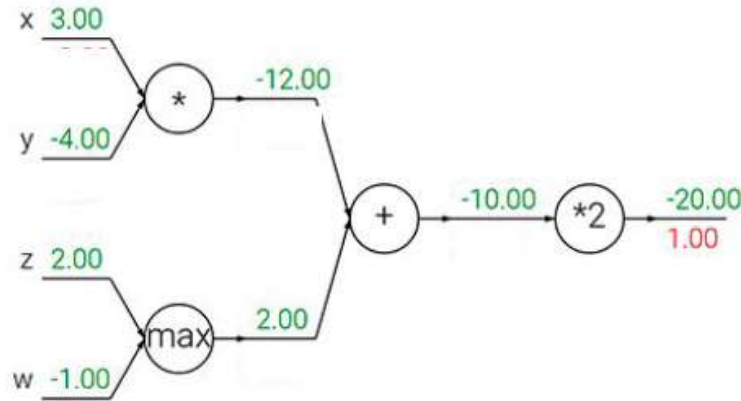
## 5. Backpropagation

Ejemplo:



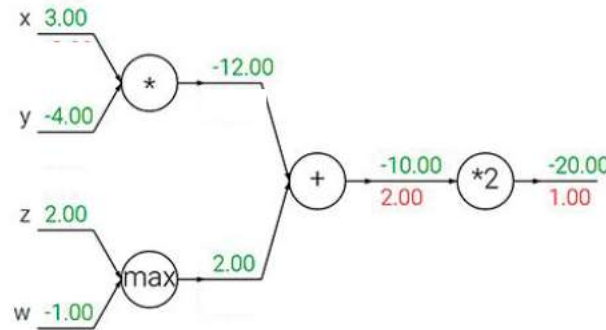
## 5. Backpropagation

El gradiente de  $f$  respecto a sí misma es 1.



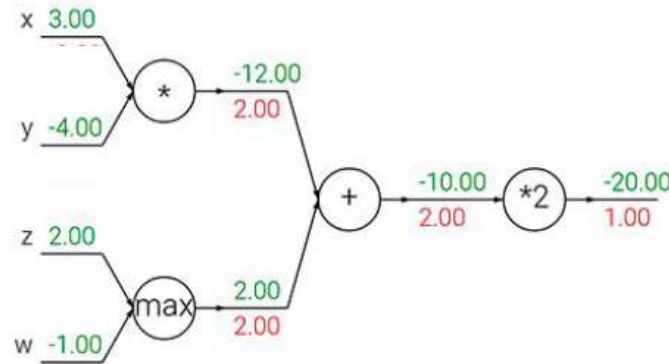
## 5. Backpropagation

**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.



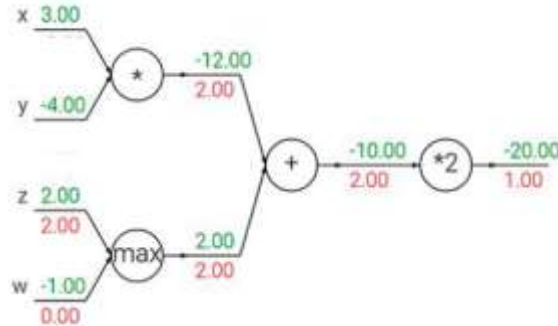
## 5. Backpropagation

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



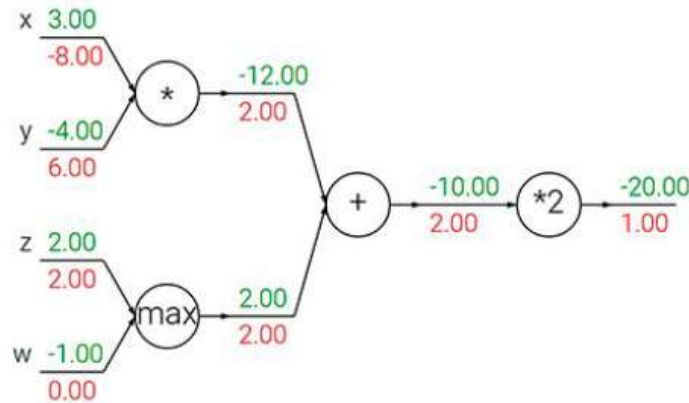
## 5. Backpropagation

**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).



## 5. Backpropagation

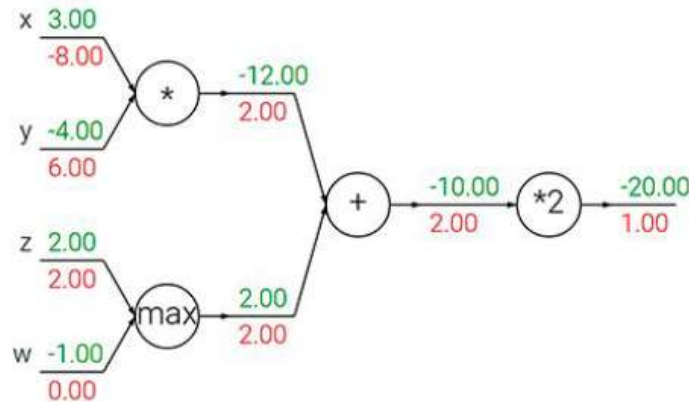
**Producto:** El gradiente de una variable de entrada viene dado por el producto del valor de la otra variable de entrada y el valor del gradiente de salida.



$$2 * (-4) = -8$$
$$2 * 3 = 6$$

## 5. Backpropagation

**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.



$$2 * (-4) = -8$$
$$2 * 3 = 6$$

## 5. Backpropagation

Hemos conseguido que operaciones muy costosas a nivel de cálculo, se conviertan en operaciones aritméticas básicas.

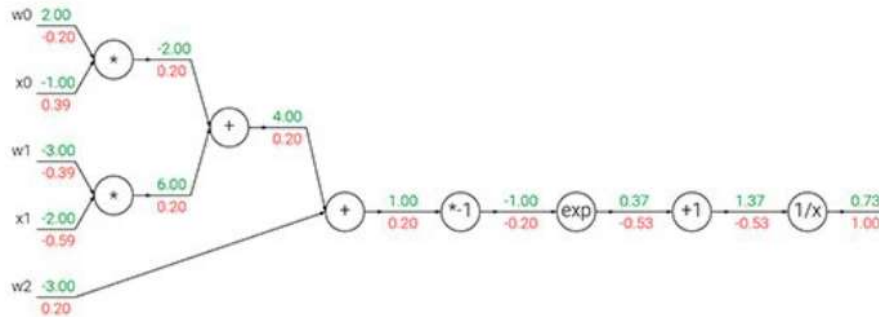
Gracias a la regla de la cadena.





## 5. Backpropagation

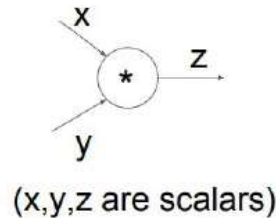
Así podríamos representar una neurona en un grafo de computación. Para una función de activación Sigmoid tenemos el siguiente grafo.



$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

## 5. Backpropagation

Fragmento de código de TensorFlow.



```
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

## 5. Backpropagation



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.

## 5. Backpropagation

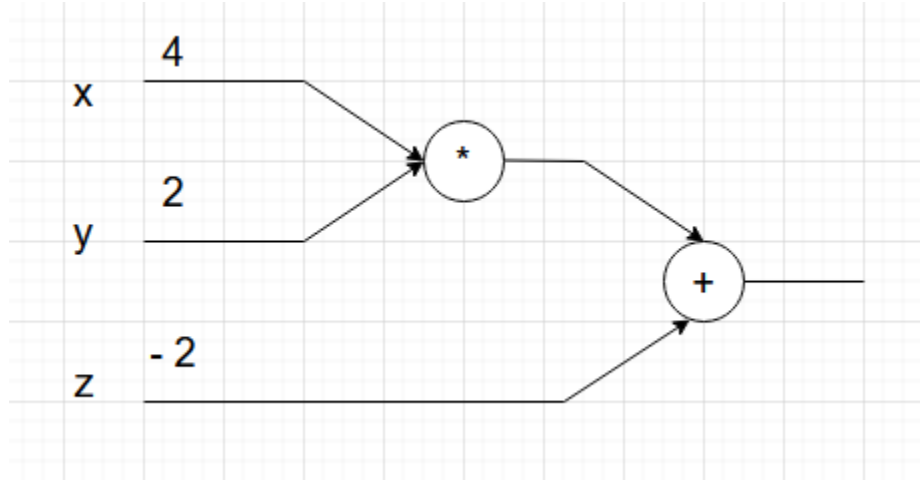


Conociendo las derivadas de cada neurona y de la función de coste, empezamos calculando el gradiente del coste final y lo propagamos hacia atrás hasta obtener los gradientes respecto a todos los parámetros.

Este proceso es general y funcionará para cualquier tipo de arquitectura de red siempre y cuando esté constituida de operaciones diferenciables.

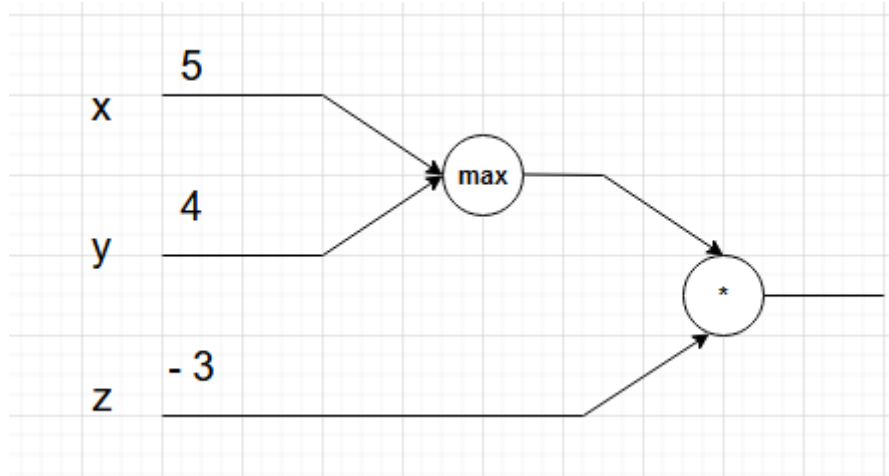
## 6. Ejercicios de regla de la cadena.

### Ejercicio 1



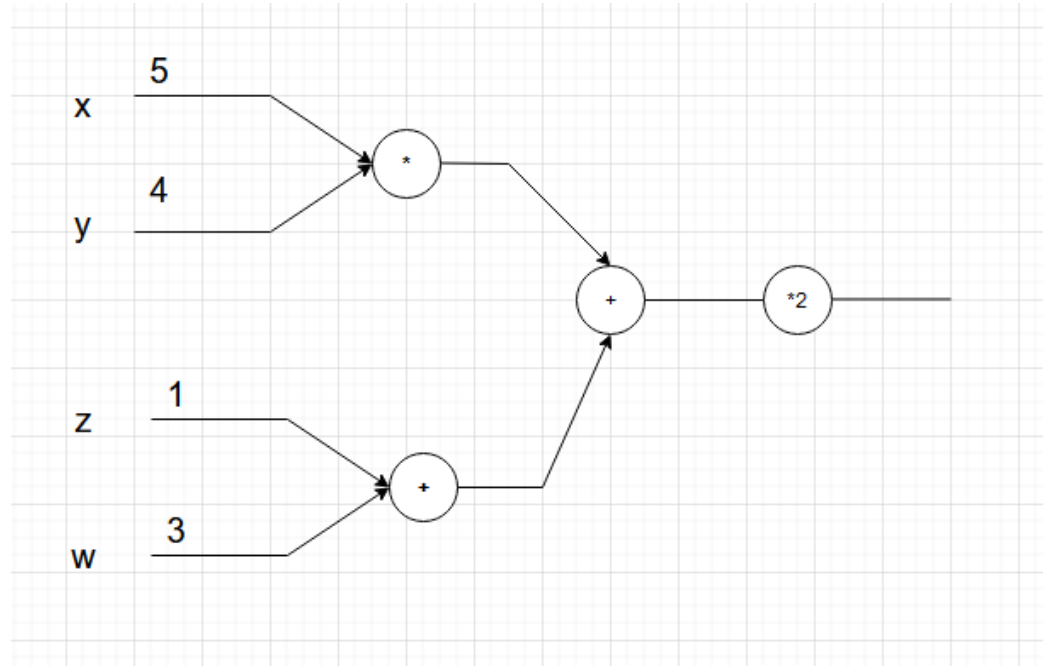
## 6. Ejercicios de regla de la cadena.

### Ejercicio 2



## 6. Ejercicios de regla de la cadena.

### Ejercicio 3



## 6. Ejercicios de regla de la cadena.

### Ejercicio 4

