

# Implementación de Redes Neuronales

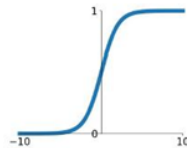
¿Qué función de activación se usa en la última capa cuando tengo una clasificación multi clase?

- Regresión → sigmoidea , lineal, ReLu, etc
- Clasificación de clases excluyentes (es clase A o B, no ambas) → sigmoidea
- Clasificación N clases simultaneas → ??

## Funciones de activación

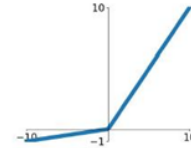
**Sigmoid**

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



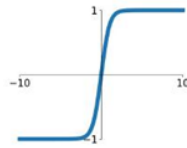
**Leaky ReLU**

$$\max(0.1x, x)$$



**tanh**

$$\tanh(x)$$

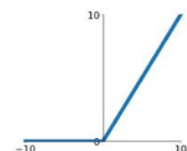


**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

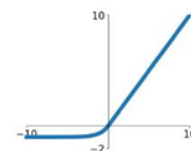
**ReLU**

$$\max(0, x)$$



**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



## Softmax:

Función exponencial normalizada. Se utiliza como función de activación de la capa de salida en modelos de clasificación, interpretándola como **scoring**, según el modelo, de pertenecer a dicha clase.

Es como un modelo de puntaje que nos da como una probabilidad de que algo pertenezca a tal o cual clase.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}} \quad \begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} \rightarrow \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \rightarrow \begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

Cuando tenemos un modelo de clasificación con una red neuronal, vamos a tener una neurona de salida por cada clase. Si se activa una neurona  $\Rightarrow$  indica de que clase es  
Si queremos que se activen varias a la vez, cada una con un grado de pertenencia



**Que pasa si tengo una Red Neuronal muy compleja?  $\Rightarrow$  OVERFITTING**

## Métodos de regularización

Son métodos que ayudan a una mejor generalización, es decir, que el modelo funcione en datos que nunca vió. Los más usados son:

- Regularización L1 y L2
- Dropout
- Early stopping
- Data augmentation

## Regularización L1 y L2

Penalizan el valor de los pesos de la red. Esto evita que se le dé más relevancia a una característica que a otra. Se le agrega un término en la función de costos proporcional a los pesos.

Si es proporcional al módulo de los pesos se llama **Regularización L1**

Si es proporcional al módulo cuadrado se le llama **Regularización L2**

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^N |w_i|$$

Loss function with L1 regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^N w_i^2$$

Loss function with L2 regularisation

Error → la diferencia entre lo que tengo y lo que esperaba

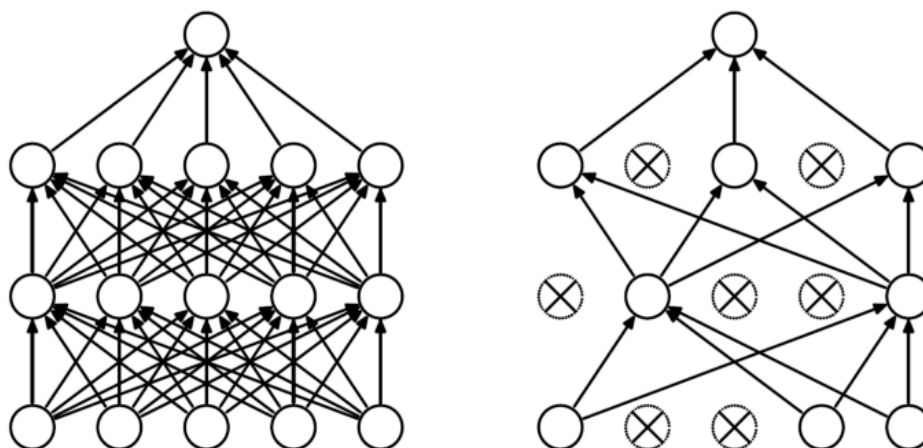
Loss es la funcion que voy a tratar de minimizar en el ciclo de entrenamiento

## Dropout

Método de regularización que evita codependencias en las conexiones de la red.

La idea es “apagar” activaciones aleatoriamente durante el entrenamiento. Esto hace que el buen funcionamiento de la red no dependa de unas pocas neuronas.

Cuando entrena la red, no lo vuelve a hacer por los mismos caminos porque estos se van apagando. Entonces, encuentra varios caminos para modelizar los datos y hace que sea mas robusta la red, que no dependa tanto de los datos de entrada.



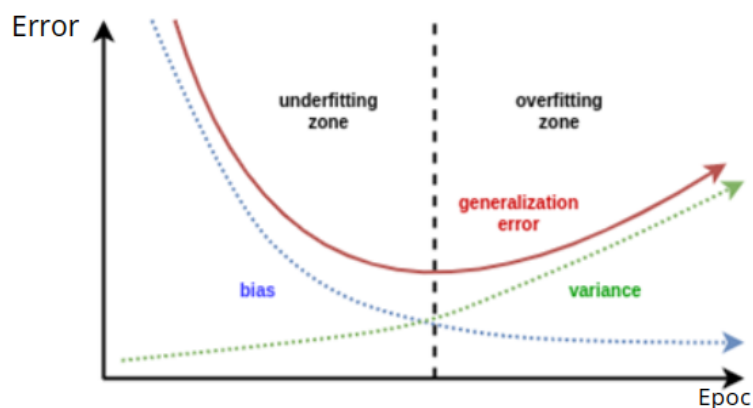
**Evita el sobreajuste (overfitting)**

## Early stopping

Dejar de entrenar cuando vemos que ya esta ajustando muy bien

La idea es evitar el sobreajuste parando el entrenamiento antes de que el error del set validación empiece a aumentar.

Este método busca entonces quedarse con los pesos en la instancia óptima.

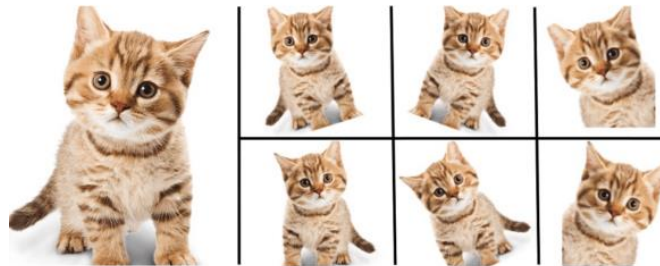


## Data augmentation

Aumentar la cantidad de datos, darle mas inputs, valores de entrada.

La idea es agregar datos usando los datos que se tienen y aplicarles transformaciones que los conviertan en nuevos datos, de manera que sean verosímiles.

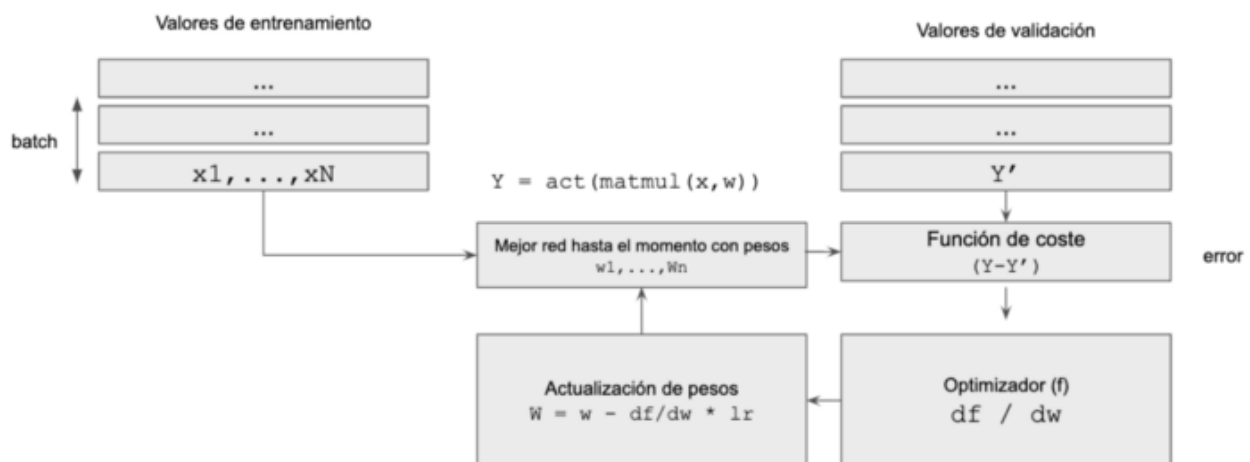
Ej: para generar mas datos de prueba se gira, invierte, se escala, se agrega ruido a una misma imagen



Es especialmente útil cuando se trabaja con imágenes. En general, es difícil encontrar las transformaciones a aplicar.

## Optimizadores

Un optimizador es una implementación concreta del algoritmo de descenso por gradiente dentro backpropagation



Los optimizadores más utilizados son:

- SGD: *stochastic gradient descent*
  - Momentum
  - Nesterov
  - RMSprop
  - AdaGrad
  - Adam
  - Nadam
- ↓
- + sofisticados  
Utilizan otra información además del gradiente para modificar los pesos, como derivadas segundas.

## SGD

Backpropagation simple, sin ningún tipo de optimización, tal y como lo vimos en clase.

Para implementar SGD, en Keras, tenemos que usar el optimizador SGD.

```
optimizer = keras.optimizers.SGD(lr=0.001)
```

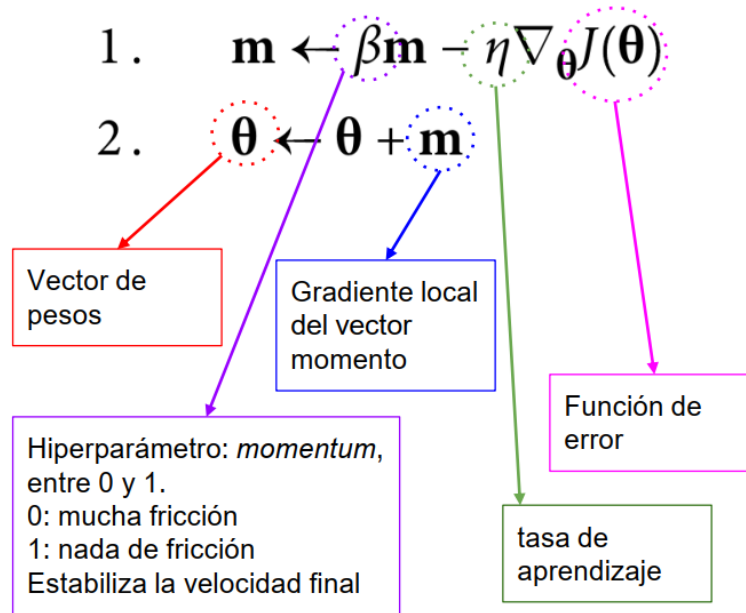
## Momentum

Idea principal: Imaginemos una pelota rodando por una colina...

Primero irá despacio, luego irá cada vez más rápido hasta alcanzar una velocidad final constante

A diferencia del backpropagation tradicional, en donde los pasos son regulares aquí son cada vez más rápidos

El gradiente se utiliza para la aceleración y no para la velocidad



Tenemos que usar el optimizador SGD, con el hiperparámetro: momentum distinto de cero

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Lo malo es que añade un nuevo hiperparámetro que ajustar.

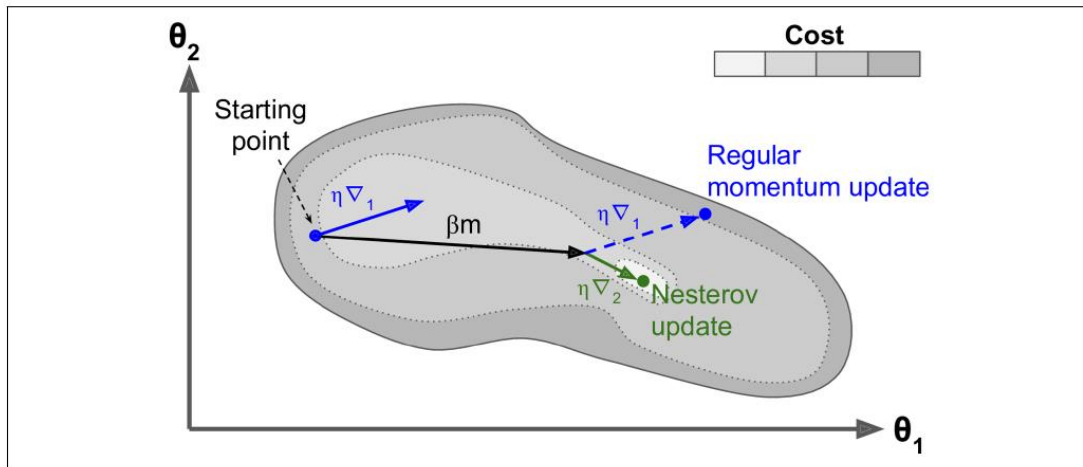
Lo bueno es que 0.9 suele funcionar bien en la mayoría de los casos, mejorando a Backpropagation tradicional

## Nesterov

Variante de Momentum, que en vez de calcular el gradiente del error en el punto actual, lo calcula un poco más adelante (en la dirección del momento):  $\theta + \beta \mathbf{m}$

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m})$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$



Tenemos que usar el optimizador SGD, pero con la opción nesterov en True:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9,
                                  nesterov=True)
```

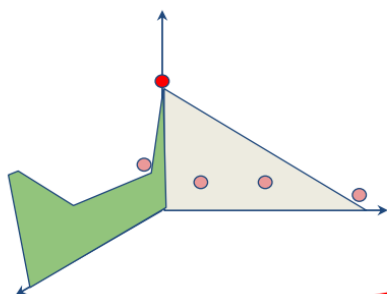
- Suele ser más rápido que Momentum

## AdaGrad

Uno de los problemas de los otros métodos es que en N dimensiones, el error descenderá por la dimensión con la pendiente más empinada, que no necesariamente será que conduzca al mínimo global.

Esto implica que hallará primero el mínimo local (más empinado) y luego irá lentamente hacia el mínimo global.





**AdaGrad** reduce el vector gradiente a lo largo de las dimensiones más empinadas.

$$1. \quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

Esto es Backpropagation normal

Pero el gradiente está dividido por un término:  $\sqrt{\mathbf{s} + \epsilon}$ , que dependerá de cuan empinado es. El símbolo  $\oslash$  indica división de vectores, cada elemento en el vector gradiente es dividido por el término

$\mathbf{s}$  es un vector, en donde se actualiza en cada paso el valor actual sumándole el cuadrado del gradiente.

**AdaGrad** reduce el vector gradiente a lo largo de las dimensiones más empinadas.

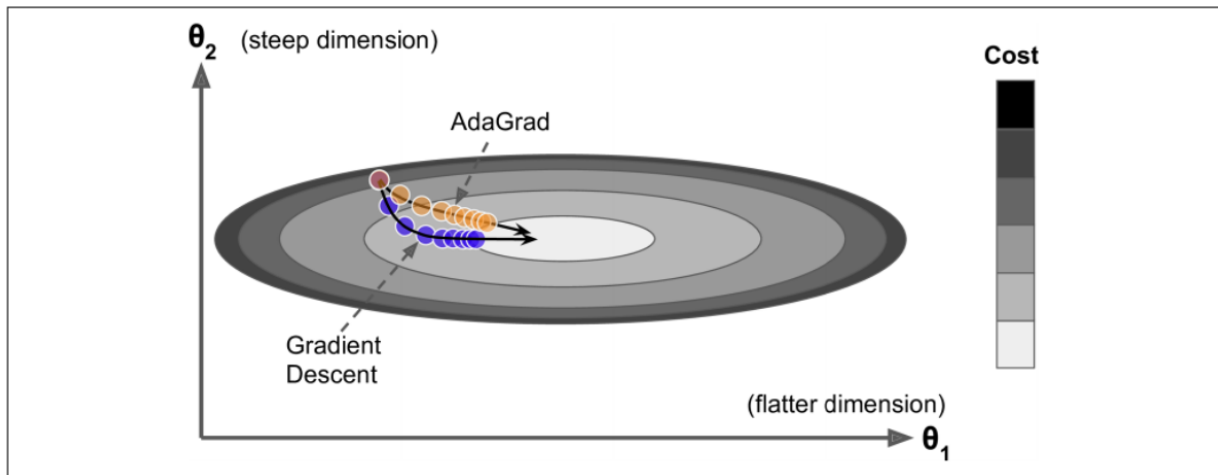
$\epsilon$ : es un término para atenuar este valor y evitar dividir por cero.

$$1. \quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

Cuanto más alto es el gradiente (la derivada), más empinada es la función de error en esa dimensión. Ya que el gradiente mide “la pendiente”.

Al dividir esta pendiente, por un valor proporcional a ella misma, se suaviza, y en todas las dimensiones el descenso por gradiente es similar, evitando caer en **valles locales** y **converjamos mas rapido al minimo global**.



Acá se ve AdaGrad vs Backpropagation, Ada no se desvía y corrige el rumbo hacia el mínimo global

### Lo bueno:

Con frecuencia AdaGrad tiene un buen desempeño para problemas cuadráticos simples.

Es bueno para tareas sencillas como regresión lineal.

### Lo malo:

A menudo se detiene demasiado pronto cuando entrena redes neuronales. Muchas veces se detiene antes de alcanzar el mínimo global.

No debería usarse para entrenar redes profundas.



Sirve para entender cómo funcionan otros métodos más complejos.

## RMSPProp

Soluciona el principal problema de AdaGrad al ir “olvidando” las pendientes anteriores, a medida que sigue avanzando. Es decir solo acumula los gradientes de las iteraciones

más recientes.

$$\begin{aligned} 1. \quad \mathbf{s} &\leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ 2. \quad \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon} \end{aligned}$$

$\beta$  es la tasa de decaimiento, y suele configurarse como 0.9.

Es un nuevo hiperparámetro, pero suele funcionar bien con este valor predeterminado.

En Keras: `optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)` → rho es  $\beta$

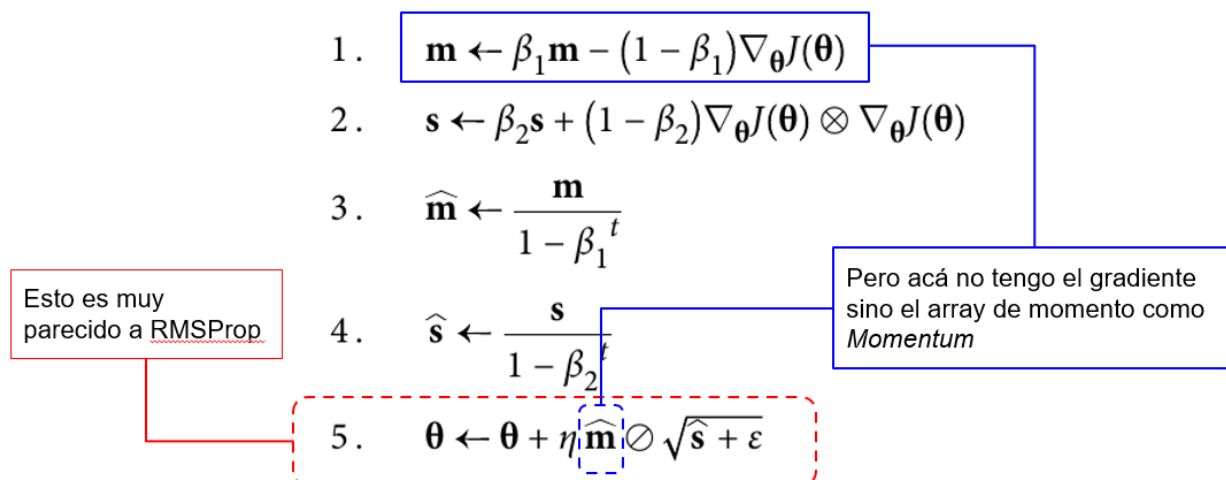
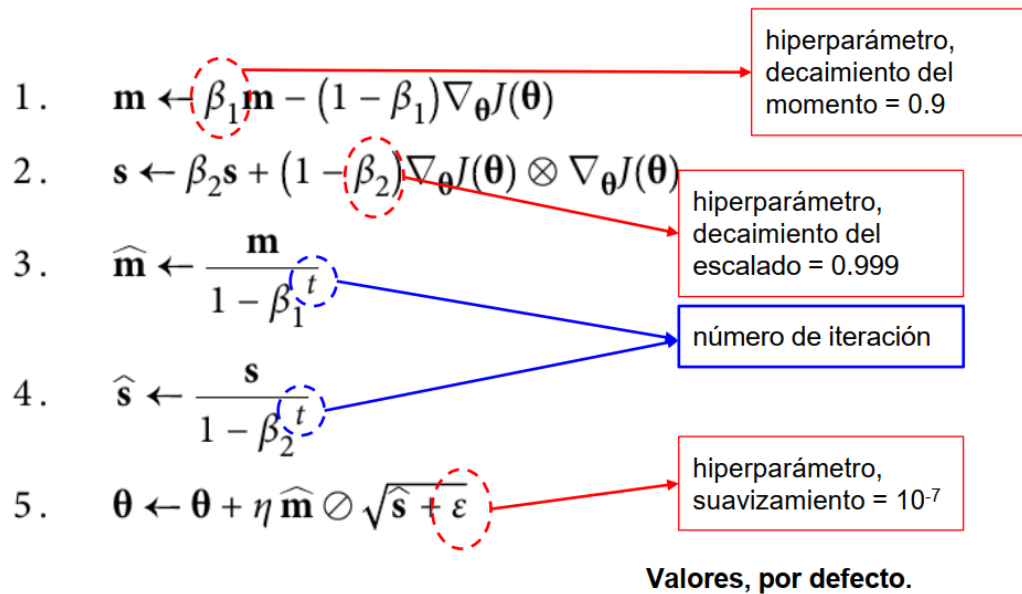
RMSProp suele ser mejor que AdaGrad y era el preferido antes de que apareciese Adam.

## Adam

Adam: Adaptive moment estimation

Combina las ideas de Momentum y RMSProp.

Hace un seguimiento de una media de decaimiento exponencial de gradientes pasados y de gradientes cuadrados pasados.



### ¿Qué son estos sombreritos?

$\mathbf{m}$  y  $\mathbf{s}$  se inicializan en 0  $\Rightarrow$  cuando  $t=1$ , por lo que  $\mathbf{m}$  y  $\mathbf{s}$  “sombrero” son también cero.

Luego a medida que  $t > 0$ , los hiperparámetros  $\beta_1$  y  $\beta_2$  tienden a cero, así que  $\mathbf{m}$  y  $\mathbf{s}$  quedan divididos por 1, y estas dos ecuaciones casi no tienen impacto.

Solo ayudan en las primeras iteraciones, para darle más fuerza a  $\mathbf{s}$  y  $\mathbf{m}$

en Keras, tenemos que usar el optimizador Adam

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9,  
                                   beta_2=0.999)
```

## AdaMax: Modificación de Adam

En general Adam da mejores resultados, pero depende del conjunto de datos.

$$\begin{aligned} 1. \quad \mathbf{m} &\leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta) \\ 2. \quad \mathbf{s} &\leftarrow \text{Max}(\tilde{\beta}_2 \mathbf{s}, \nabla_{\theta} J(\theta)) \\ 3. \quad \widehat{\mathbf{m}} &\leftarrow \frac{\mathbf{m}}{1 - \beta_1^t} \end{aligned}$$

$$5. \quad \theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \oslash \mathbf{s}$$

## Nadam

Es Adam + Nesterov, así que a menudo converge más rápido que Adam

## Adadelta

Es una variación de AdaGrad en la que en vez de calcular el escalado del factor de entrenamiento de cada dimensión, teniendo en cuenta el gradiente acumulado desde el principio de la ejecución, se restringe a una ventana de tamaño fijo de los últimos  $n$  gradientes.

Similar a RMSProp que va olvidando los gradientes

# Numero de capas

Para muchos problemas 1 capa oculta será suficiente.

En teoría un PMC (perceptron multicapa) con una sola capa oculta puede modelizar funciones complejas. Tendrá que tener neuronas suficientes.

Pero si estamos ante problemas más complejos, las redes profundas tendrán mejor desempeño, ya que pueden modelizar mejor con menos neuronas totales.

Ejemplo: MNIST:



1 capa oculta (cientos de neuronas) = 97 %

2 capas ocultas (mismo número de neuronas total) = 98%

Otros problemas como reconocimiento de imágenes o del discurso requieren decenas o cientos de capas, pero todas ellas conectadas como en PMC

El número de neuronas de la capa de entrada y de salida está determinado por el problema a resolver:

cada número es una imagen de 28x28 píxeles = 784 neuronas de entrada.

Los dígitos a reconocer, son los del sistema decimal tradicional. Así que son 10, del 0 al 9. 10 neuronas de salida.

Lo habitual es hacer una pirámide. Poniendo cada vez menos neuronas. Por ejemplo para MNIST, 3 capas ocultas podrían tener: 300, 200 y 100 neuronas cada una.

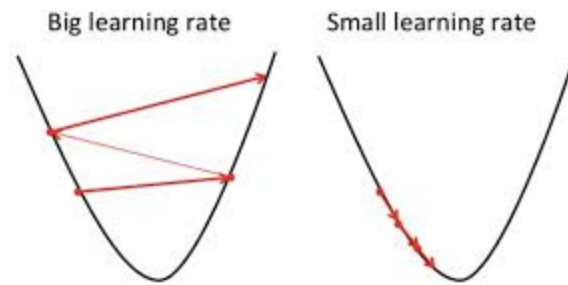
Sin embargo últimamente se ha cuestionado esta técnica, ya que a veces poner la misma cantidad de neuronas en todas las capas da el mismo resultado o a veces mejor.

## Hiperparametros

**Learning rate o tasa de aprendizaje:** es el más importante, indica qué tan rápido se va descediendo en la función de costo, qué tan rápido va acercandose al minimo local o global.

Que tanto voy a actualizar los pesos en cada ciclo.

Valores usuales: E-01 a E-04



Si avanza muy rapido puede no llegar a encontrarlo, porque da saltos muy grandes

Si avanza muy chiquito voy a tener que dar muchos pasos para llegar al minimo

## Entrenamiento de la red

Una vez que tenemos los ingredientes anteriores, podemos entrenar la red.

**(Arquitectura + Hiperparámetros + optimizador + Función de pérdida + funciones de activación)**

