

Departamento de Física Médica - Centro atómico Bariloche - IB

Entrenamiento de redes neuronales

Ariel Hernán Curiale
ariel.curiale@cab.cnea.gov.ar

La mayoría de los slides fueron adaptados de Fei Fei Li, J. Johnson y S. Yeung, cs231n, Stanford 2017.



UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO



Entrenamiento de redes neuronales

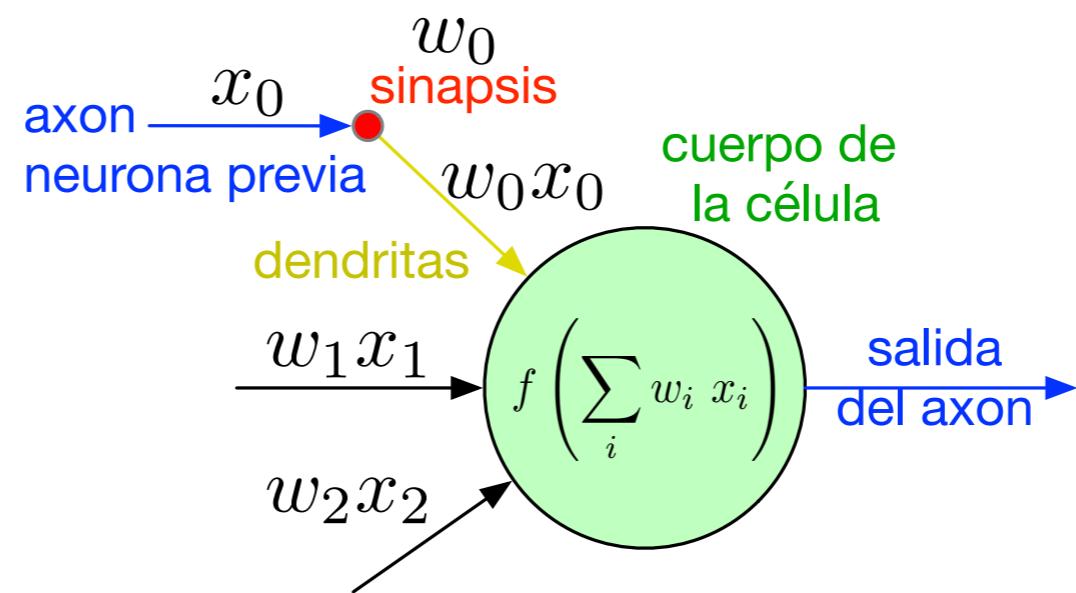
- ❖ Existen varios factores a considerar a la hora de entrenar las redes neuronales:
 - ❖ Algunos sólo se definen una vez, como la función de activación, la función de costo, el tipo de inicialización de los pesos, el tipo de regularización, etc.
 - ❖ Otros son dinámicos, como la optimización de los hiperparámetros
 - ❖ Por último tenemos la evaluación del modelo en términos de generalización

Funciones de Activación

Funciones de Activación

Comencemos estudiando las funciones de activación y su impacto en el entrenamiento de las redes neuronales.

- ❖ A cualquier capa la podemos ver como una versión vectorial del modelo de neurona de McCulloch-Pitts donde las neuronas de la capa pueden (CNN) o no compartir los pesos y tener asociado un campo receptivo

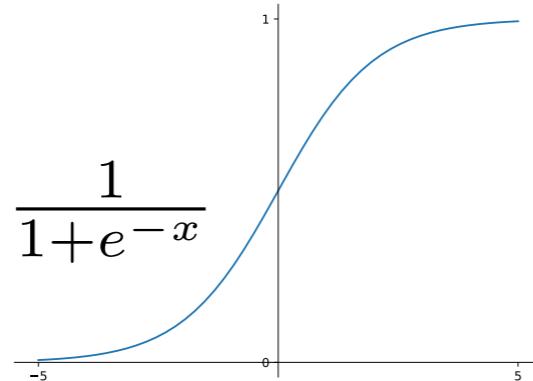


Funciones de activación

Recordemos las funciones de activación que vimos

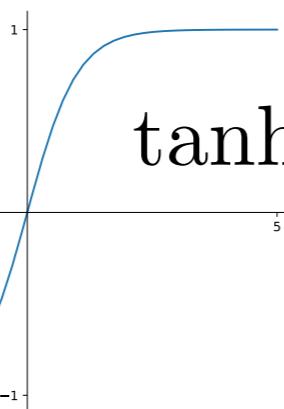
- ❖ Sigmoid

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



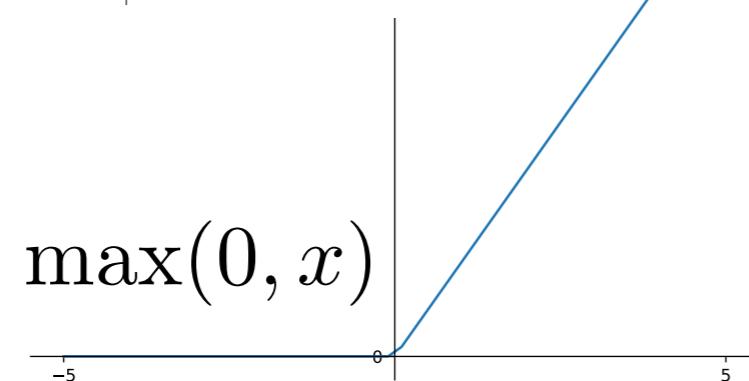
- ❖ tanh

$$\tanh(x)$$



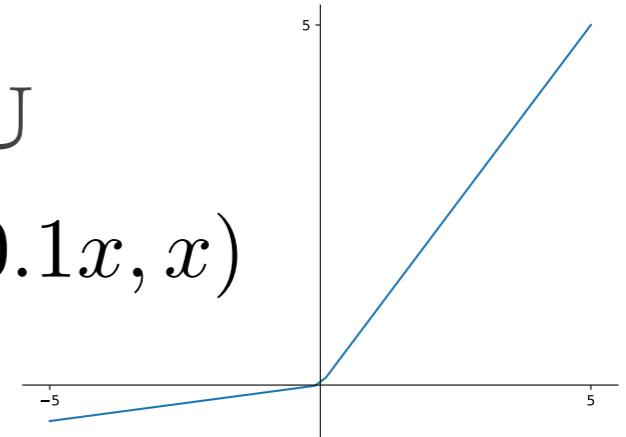
- ❖ ReLU

$$\max(0, x)$$



- ❖ Leaky ReLU

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

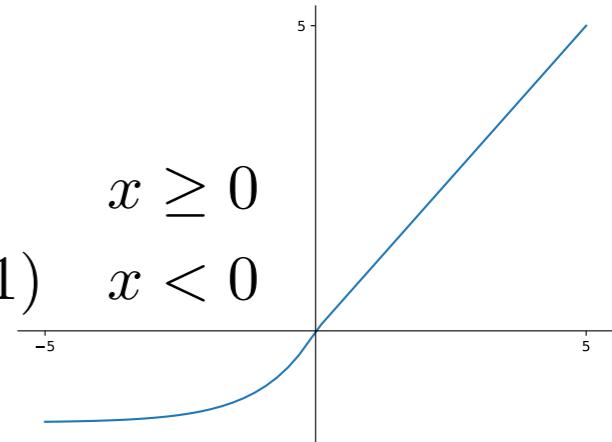


- ❖ Maxout

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

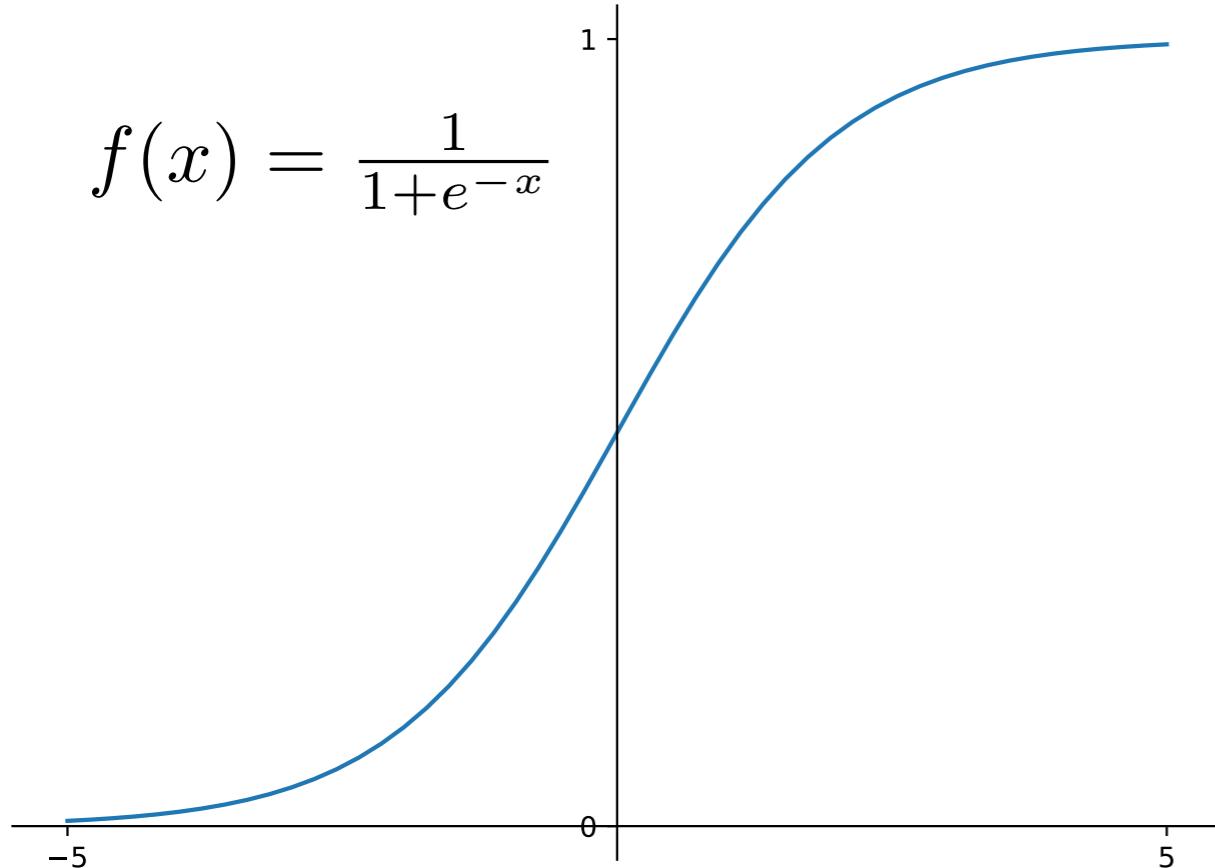
- ❖ ELU

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



Sigmoide

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

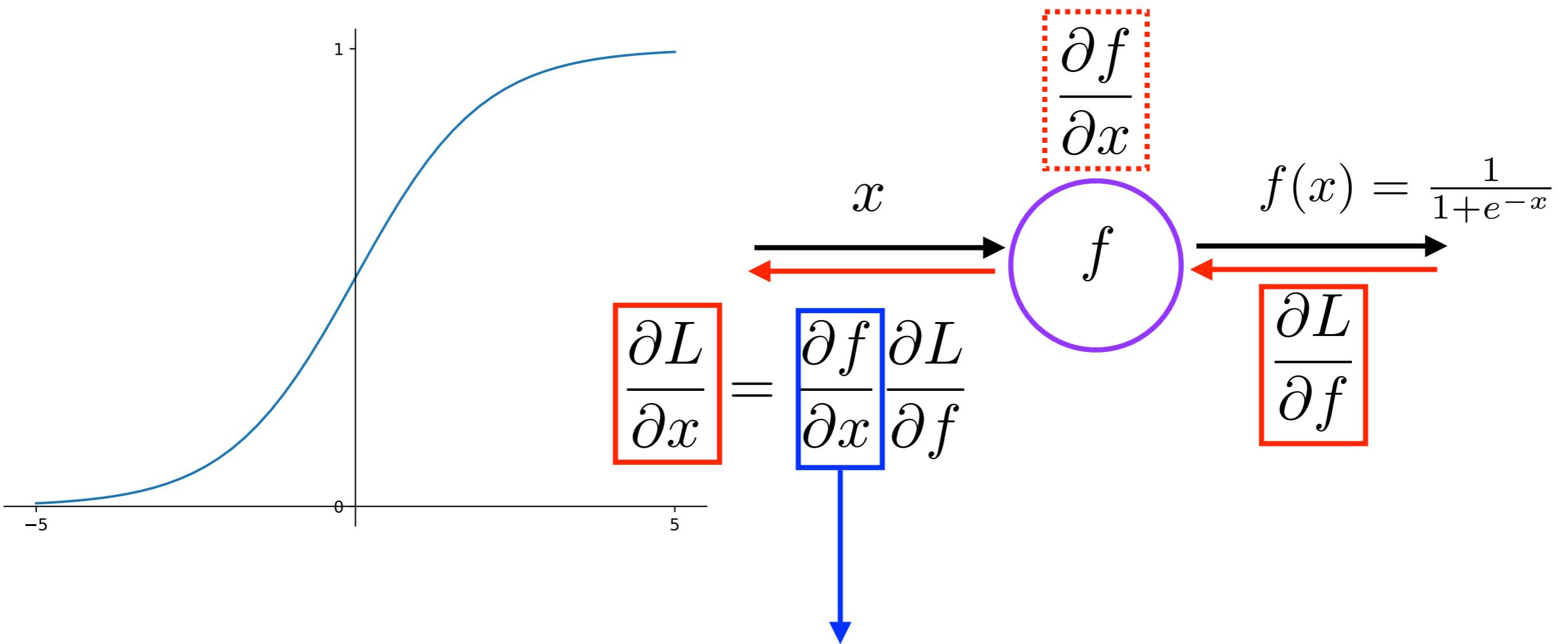


- ❖ El resultado se encuentra en el rango $[0, 1]$ dando una interpretación directa con la saturación de la tasa de disparo de las neuronas. Es por ello que es históricamente muy popular.
- ❖ Régimen lineal en valores cercanos al cero

Tiene 3 problemas principales:

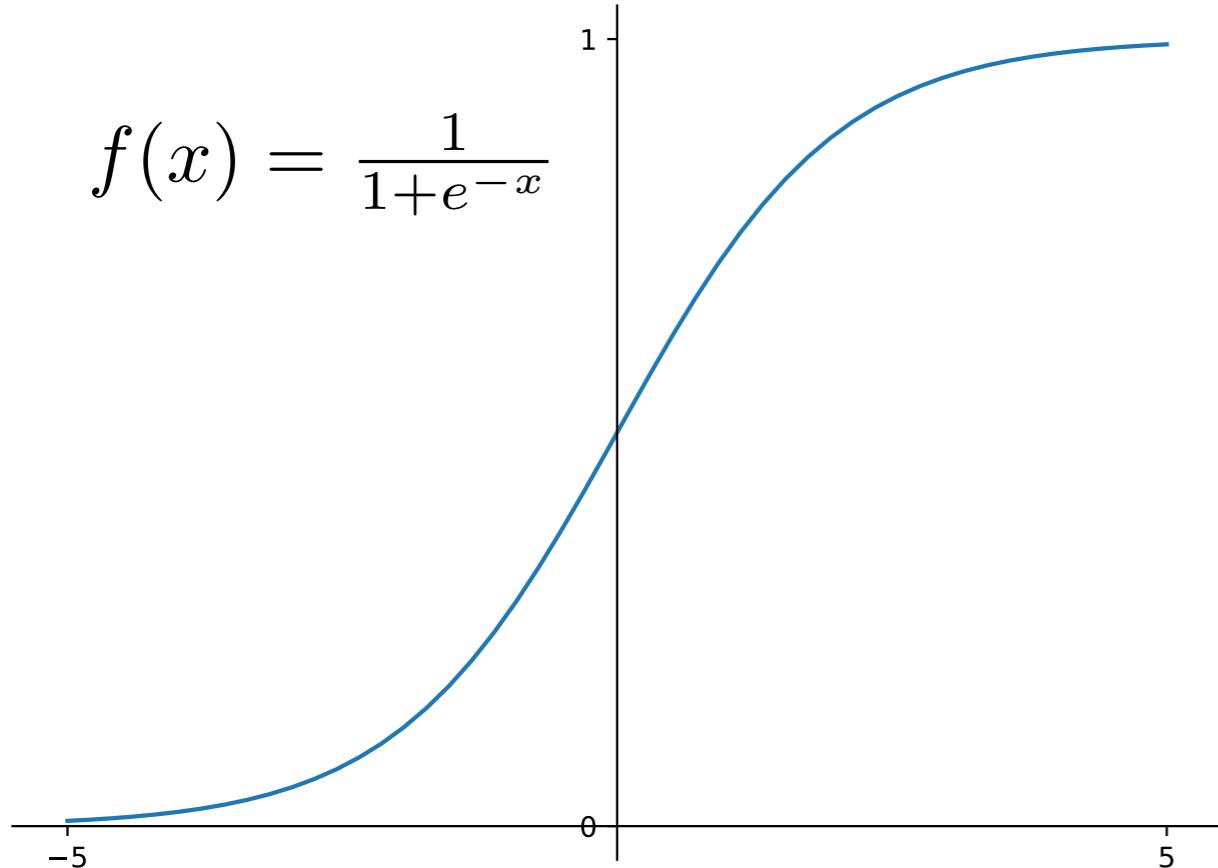
- ❖ Neuronas saturadas matan el gradiente: al saturar la neurona el gradiente es cero

Sigmoide



Sigmoide

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



- ❖ El resultado se encuentra en el rango $[0, 1]$ dando una interpretación directa con la saturación de la tasa de disparo de las neuronas. Es por ello que es históricamente muy popular.
- ❖ Régimen lineal en valores cercanos al cero

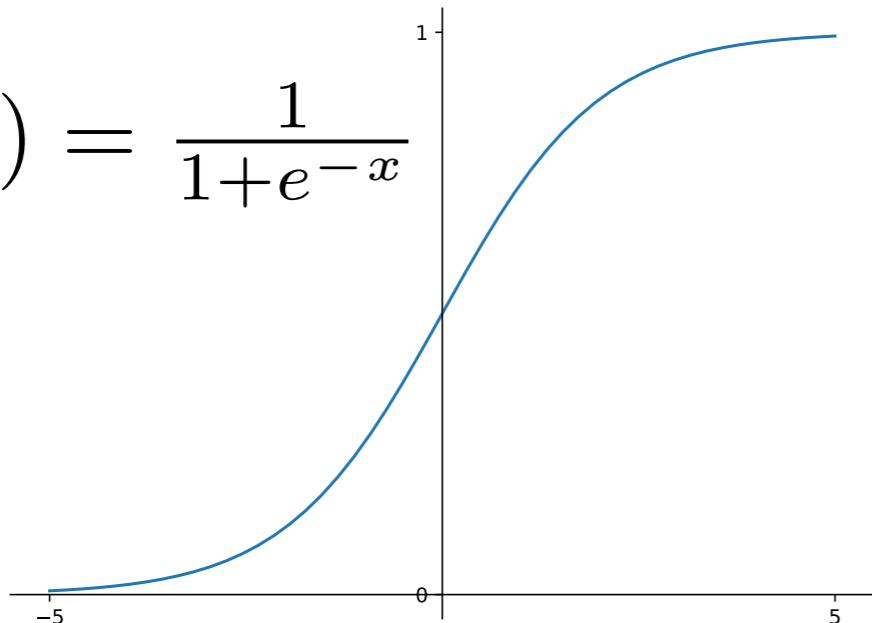
Tiene 3 problemas principales:

- ❖ Neuronas saturadas matan el gradiente: al saturar la neurona el gradiente es cero
- ❖ Su salida no esta centrada en cero (media cero)

Sigmoide

- ❖ Veamos qué pasa con el gradiente para el caso de x positivo.
 - ❖ El gradiente de la función es siempre positivo.

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



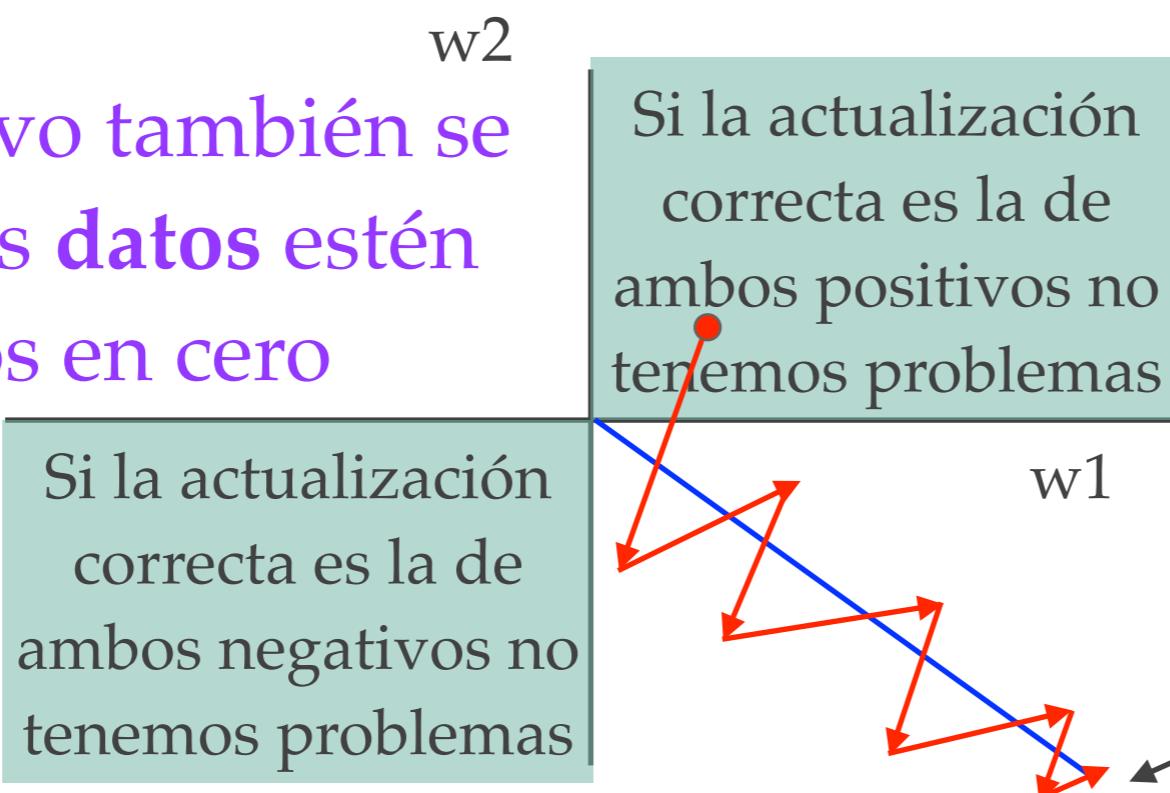
$$\frac{\partial f}{\partial x} = \frac{(1 - f(x))f(x)}{> 0 > 0} > 0$$

- ❖ El gradiente respecto a los pesos es para todos los pesos mayor que cero
- $$\frac{\partial L}{\partial w_i} = \frac{\partial f}{\partial x} x_i$$
- y siempre menor cuándo x es negativo

Sigmoide

- ❖ Supongamos tenemos sólo dos pesos en la red, w_1 y w_2

Por este motivo también se busca que los **datos** estén centrados en cero

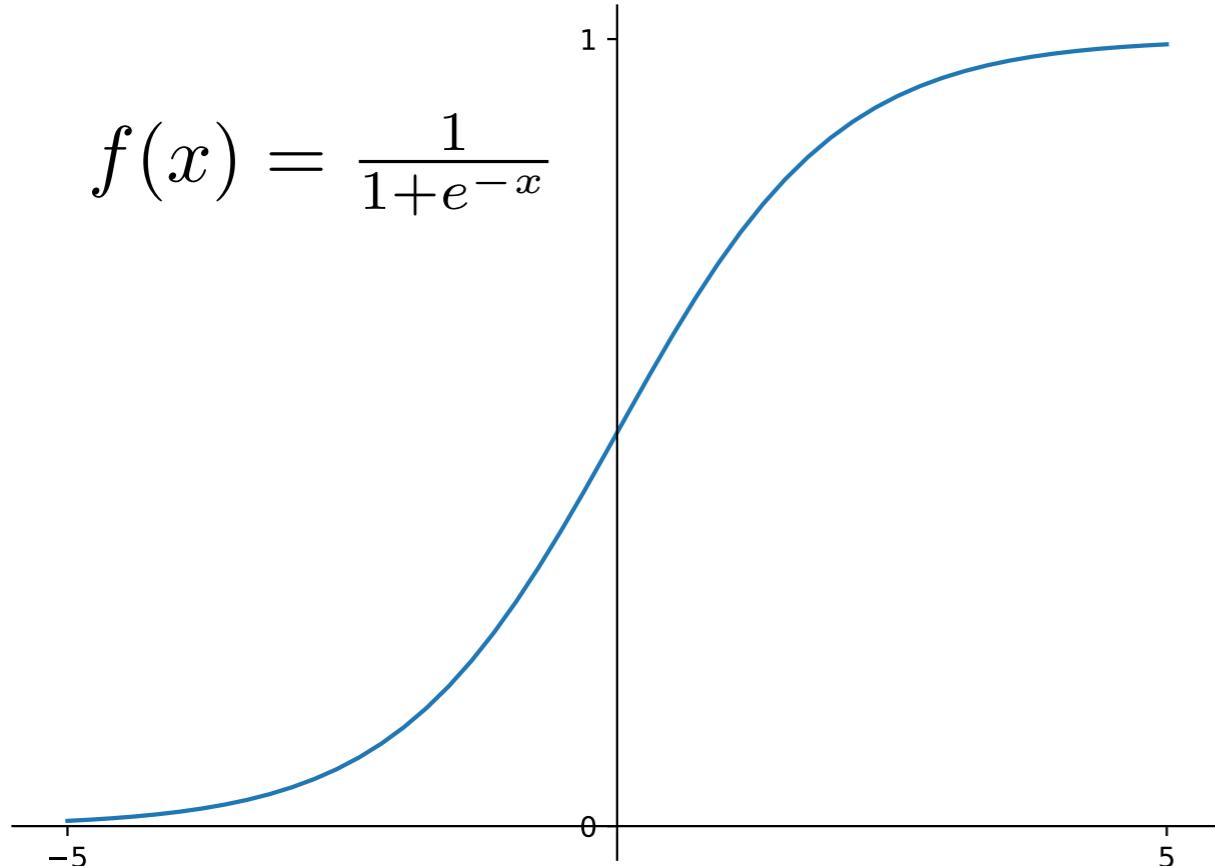


Supongamos que la correcta es w_1 positivo y w_2 negativo con la entrada positiva

Como la actualización de todos los pesos es siempre positiva o negativa, se genera un efecto zig-zag bastante ineficiente para el proceso de optimización

Sigmoide

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



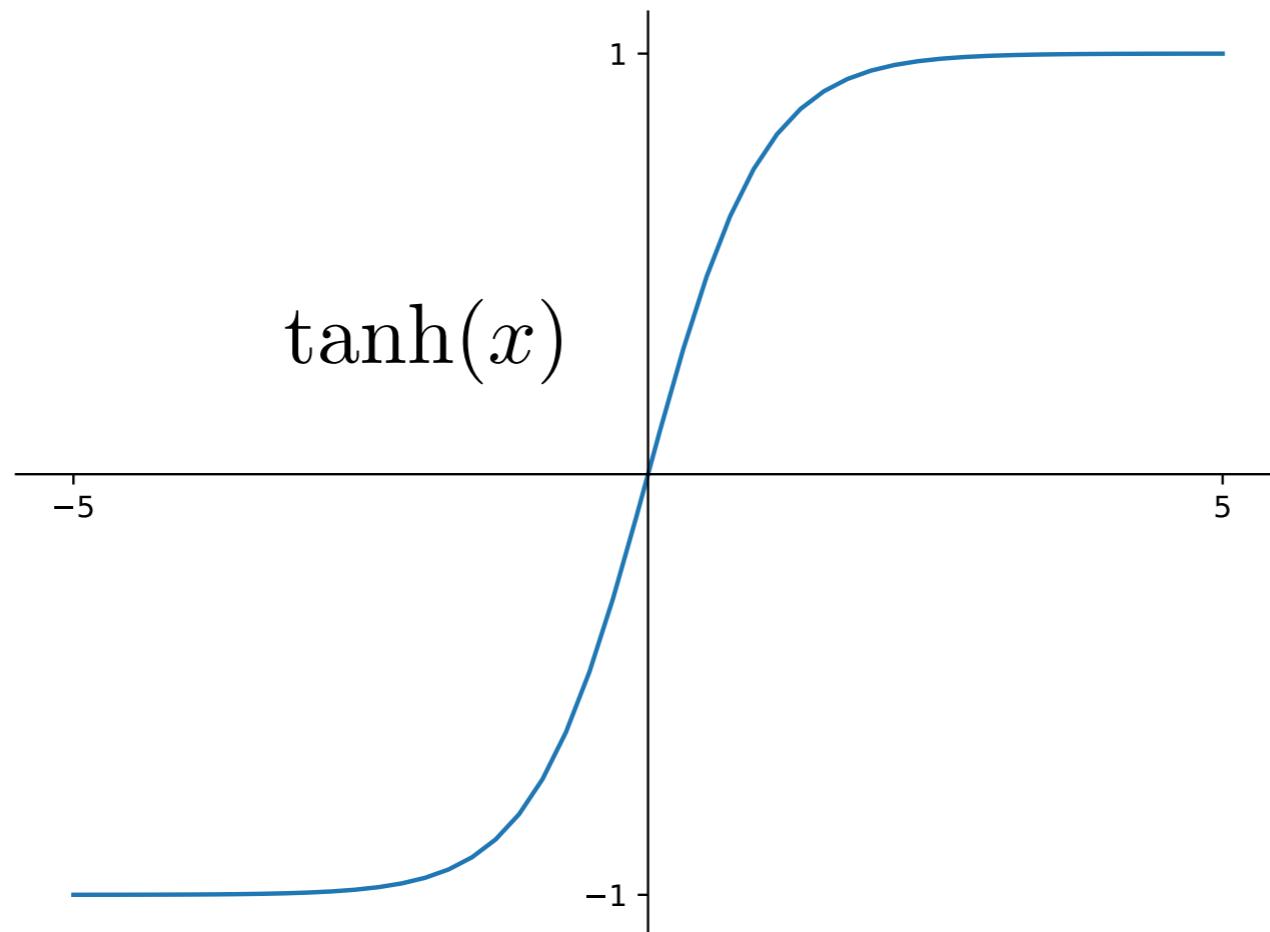
- ❖ El resultado se encuentra en el rango $[0, 1]$ dando una interpretación directa con la saturación de la tasa de disparo de las neuronas. Es por ello que es históricamente muy popular.
- ❖ Régimen lineal en valores cercanos al cero

Tiene 3 problemas principales:

- ❖ Neuronas saturadas matan el gradiente: al saturar la neurona el gradiente es cero
- ❖ Su salida no esta centrada en cero (media cero)
- ❖ La función $\exp()$ es computacionalmente costosa

Tangente Hiperbólica (Tanh)

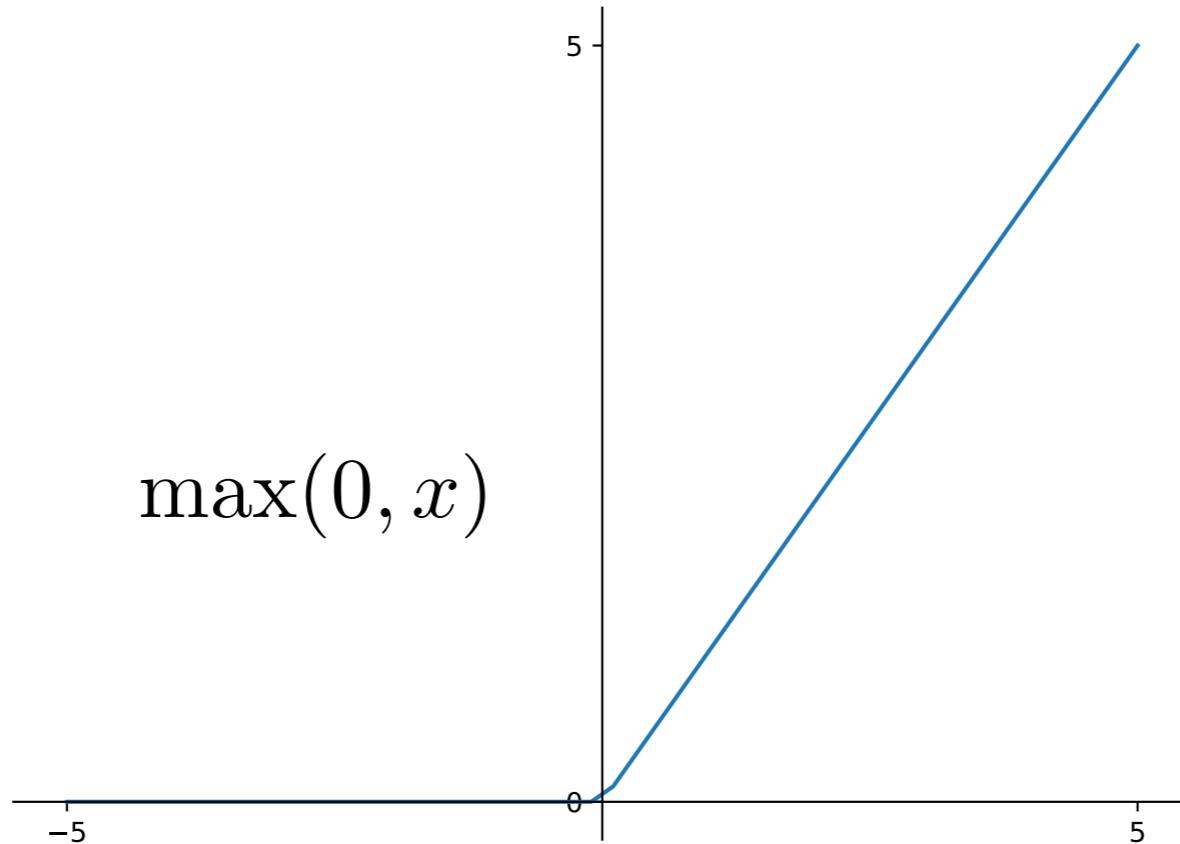
- ❖ Salida entre $[-1, 1]$
- ❖ Centrada en cero
- ❖ Cuando se satura mata el gradiente



Rectified Linear Unit (ReLU)

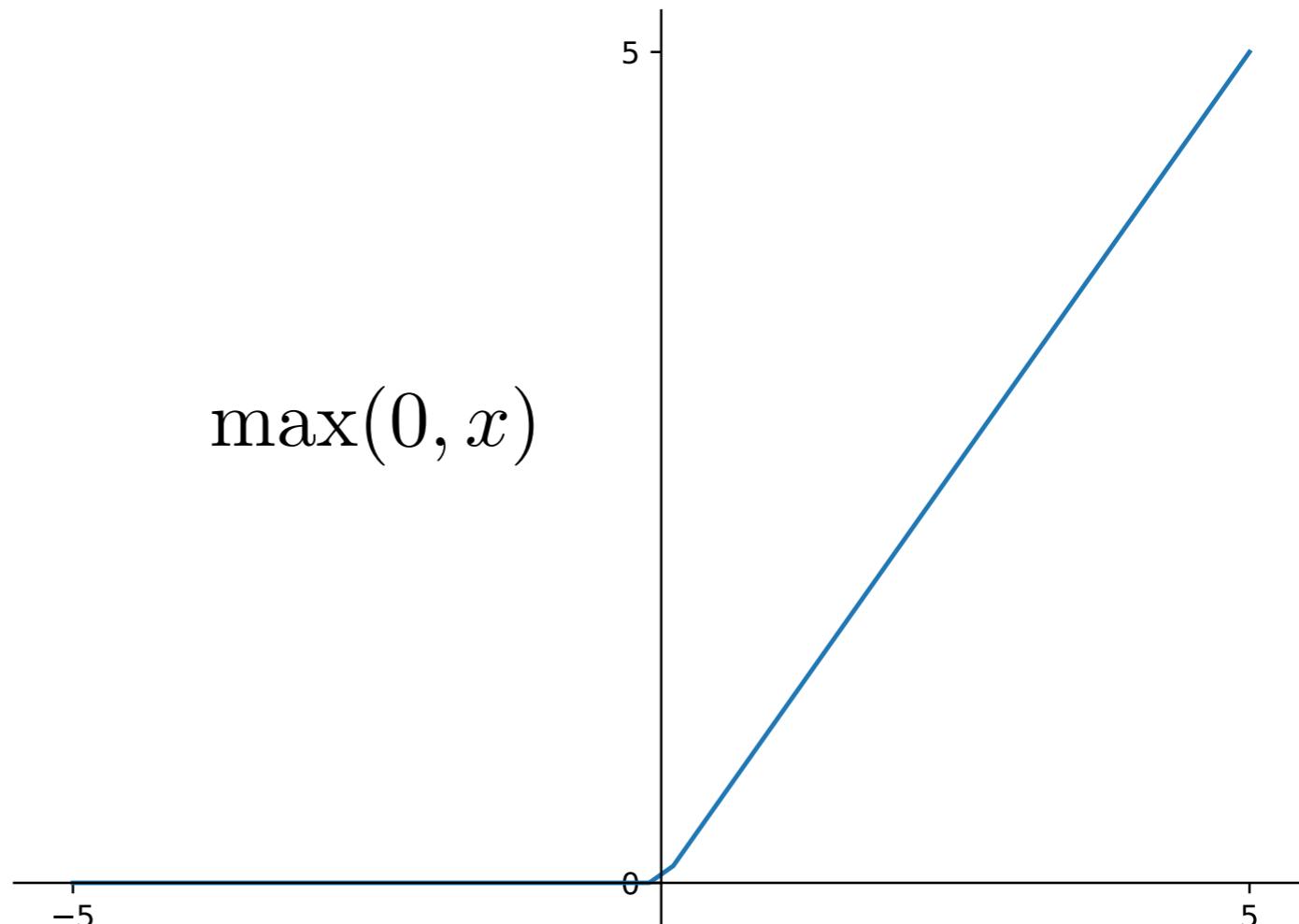
Alex Net [Krizhevsky et al., 2012]

- ❖ La salida no satura en la región + : Previene que muera el gradiente en la mitad del régimen
- ❖ Computacionalmente eficiente
- ❖ Converge mucho más rápido que sigmoide o tanh (ej. 6x)



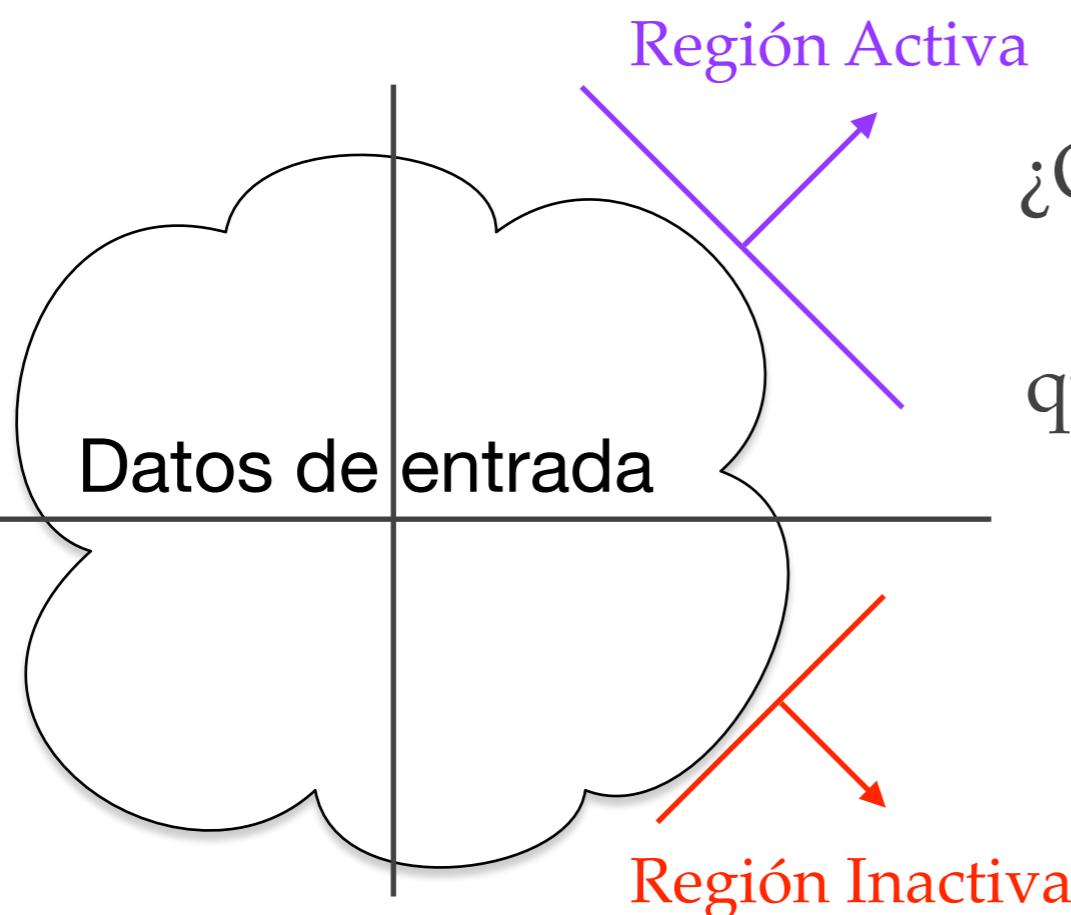
Rectified Linear Unit (ReLU)

- ❖ No esta centrada en cero
- ❖ Cuando se satura mata el gradiente (región -)
- ❖ Al no saturar en la región + se vuelve numéricamente inestable ($x \gg 0$)



Rectified Linear Unit (ReLU)

- ❖ Tiene una zona muerta donde nunca se activa ($x < 0$).



¿Qué pasa si inicializamos los pesos y juntos con nuestros datos produce que la ReLu se encuentre en la región inactiva ?

nunca actualiza los pesos

En la práctica la inicialización de los pesos tiene un sesgo positivo para evitar esto (ej.0.01). Aunque **NO** siempre funciona

Leaky ReLU

[Mass et al., 2013] [He et al., 2015]

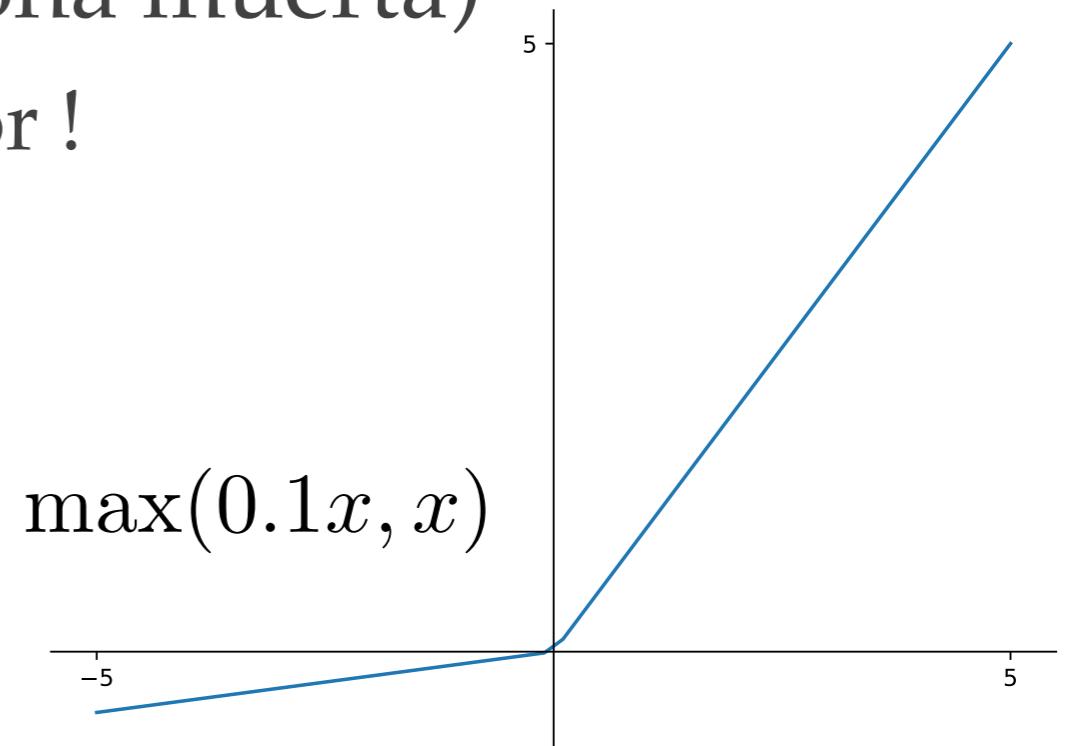
- ❖ No satura
- ❖ Computacionalmente eficiente
- ❖ Converge rápido como la ReLU
- ❖ No mueren neuronas (no tiene zona muerta)

Versión paramétrica (PReLU) mejor !

$$f(x) = \max(\alpha x, x)$$

¿Cómo elegimos alfa?

El parámetros se aprende en el
método de backpropagation

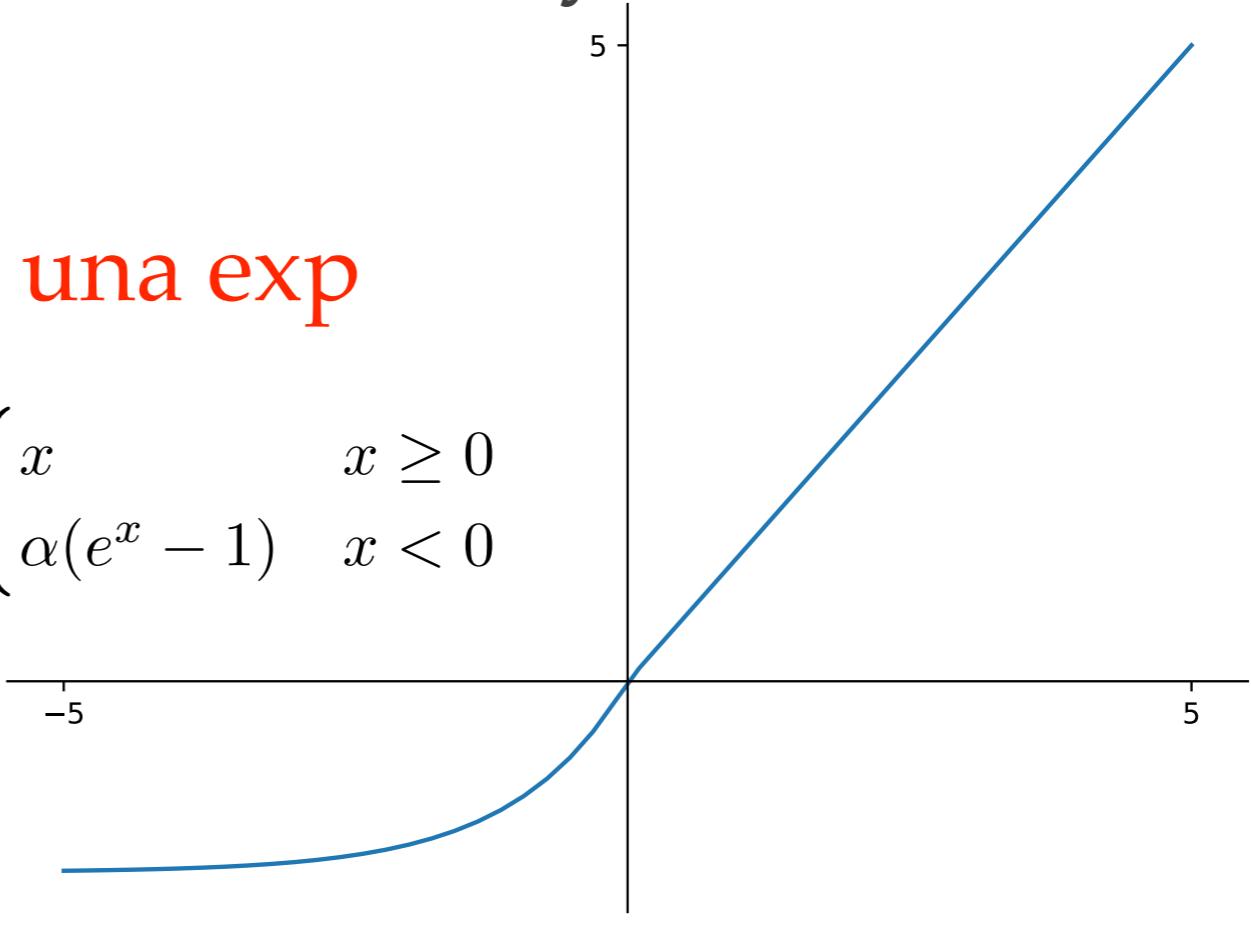


Exponential Linear Unit (ELU)

[Clevert et al., 2015]

- ❖ Mismas ventajas que ReLU
- ❖ En un entorno cercano al cero tiene media cero
- ❖ Satura en la región “-“ dando una mayor robustez frente al ruido
 - ❖ Requiere el cálculo de una exp

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



Maxout

[Goodfellow et al., 2013]

- ❖ Generaliza ReLU y Leaky ReLu
- ❖ Tiene un régimen lineal, No satura y por tanto nunca mueren

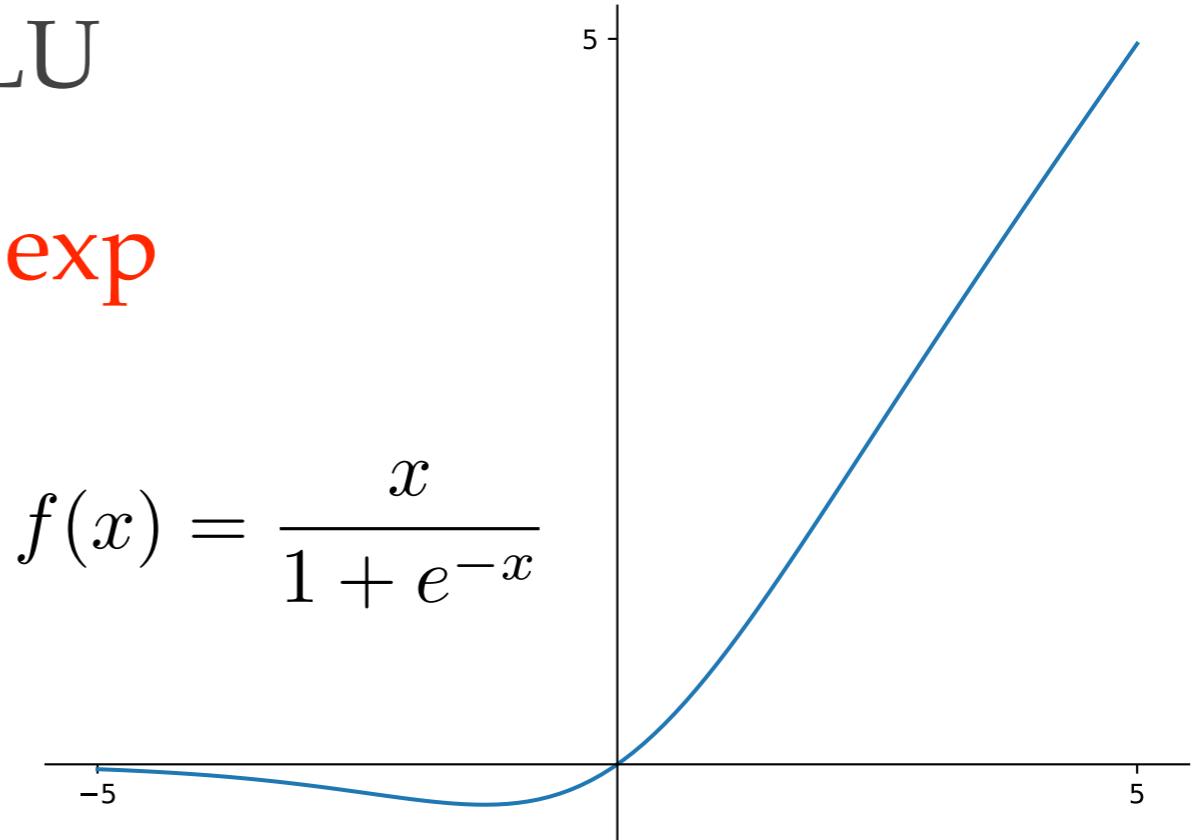
$$\max(w_1^T x + b1, w_2^T x + b2)$$

- ❖ Duplica el número de parámetros por neurona

Swish

[Ramachandran et al., 2017]

- ❖ Parece tener mejor rendimiento que ReLU
- ❖ Tiene mejor propiedad de media cero que ReLU
- ❖ Mismos beneficios que ReLU
- ❖ Requiere el cálculo de una exp



Funciones de Activación

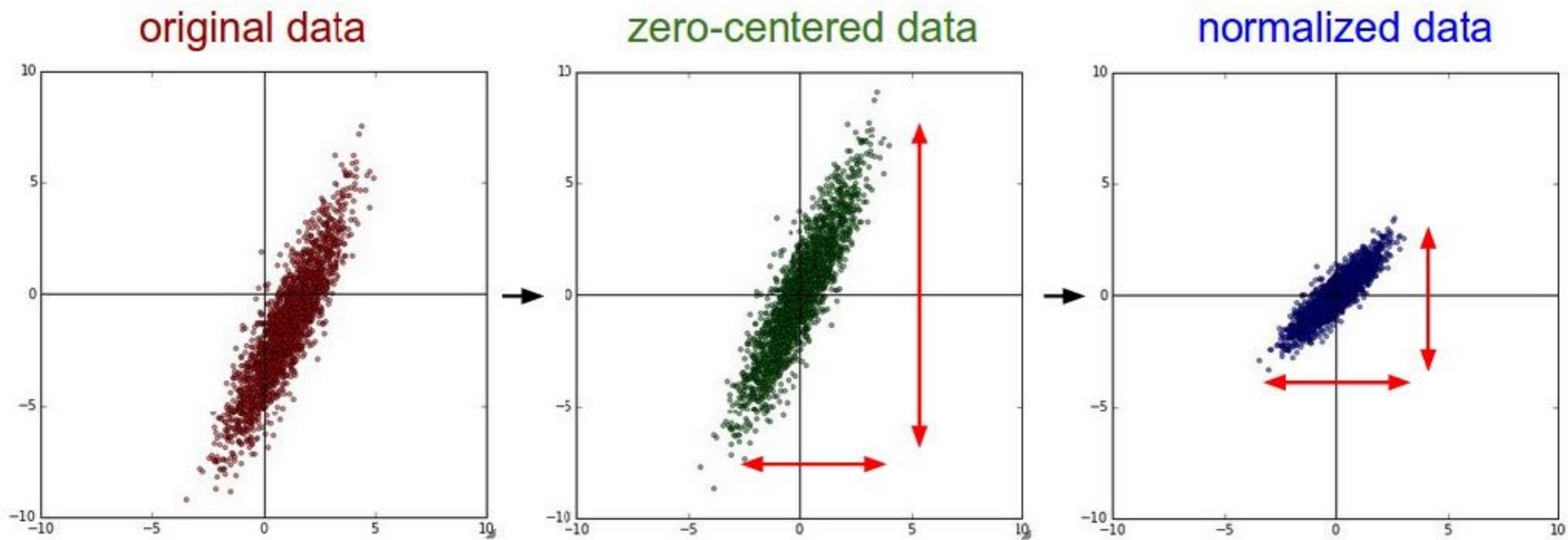
Consideraciones prácticas, sobre todo en Deep networks

- ❖ Usar **ReLU** pero tener cuidado con la tasa de aprendizaje para evitar inestabilidad (no satura).
- ❖ Probar **Leaky ReLU/Parametric Leaky ReLU/Maxout/ELU/Swish.**
- ❖ Probar **Tanh** pero no esperar mucho.
- ❖ **No utilizar sigmoid.**

Preprocesado de los datos

Preprocesado

Recordar: Este preprocesado reduce el efecto zig-zag en la optimización



X es una matriz de NxD.

N = #ejemplos

D = #características

$$X - \mu$$

`X -= np.mean(X, axis=0)`

$$\frac{X - \mu}{\sigma}$$

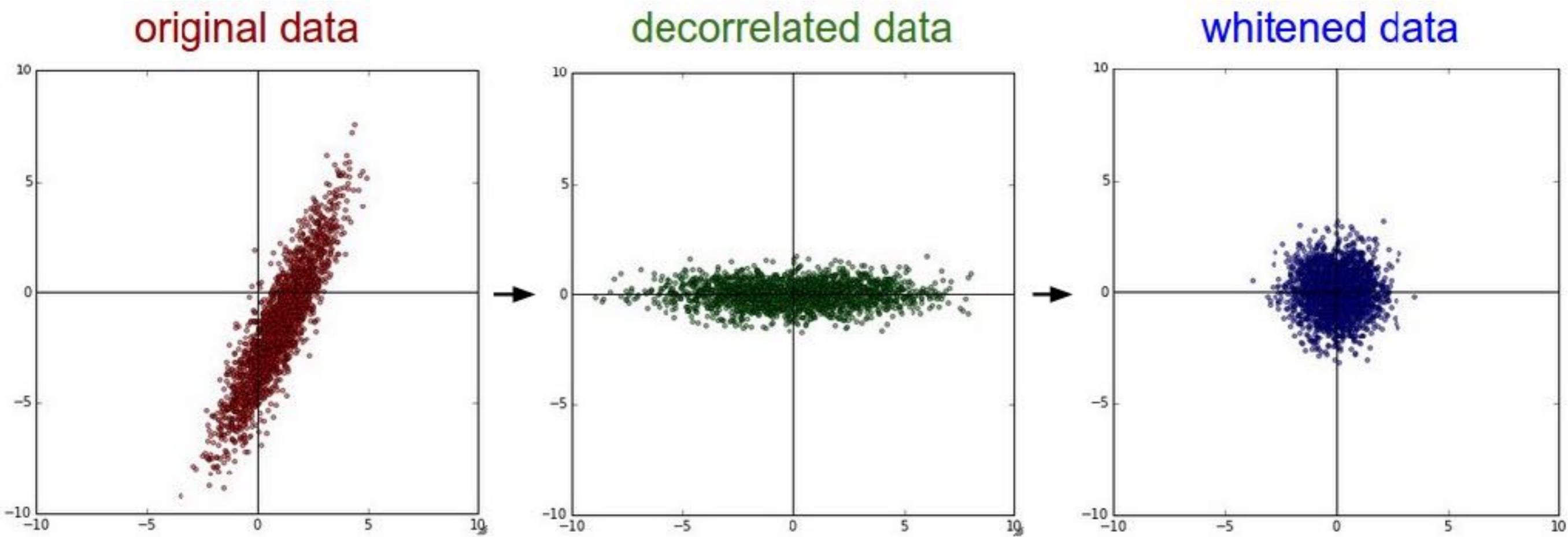
`X /= np.std(X, axis=0)`

Preprocesado

- ❖ En imágenes clásicas, normalmente no se normalizan los datos ya que las img tienen el mismo rango de intensidad
- ❖ Tiene más sentido en imágenes médicas donde el rango de intensidades no es fijo, como por ejemplo en MR
- ❖ En ML en general tiene aún más sentido ya que las características de entrada pueden provenir de diferentes sensores o representar diversas cosas, obviamente con escalas variadas (ej. temperatura, edad, peso, ph, velocidad, etc..)

Preprocesado

- ❖ En ML es común hacer **PCA** y **whitening** a los datos, aunque poco común en imágenes



Se busca que los datos tengan una matriz de covarianza diagonal

La matriz de covarianza es la identidad en este caso

Preprocesado en imágenes

En la práctica solamente se centran los datos, y en img médicas puede ser útil normalizarlos, existen diversas estrategias

- ❖ Substraer la img. media (ej. [32,32,3] en CIFAR-10)
- ❖ Substraer la media por canal para cada img.
- ❖ Substraer la media de todo el dataset a cada pixel.

En imágenes médicas, con diferentes rangos de intensidades por pacientes puede ser útil normalizar cada paciente de forma individual → ¿Prob. de hacer esto?

En imágenes no se suele utilizar PCA y whitening

Preprocesado en imágenes

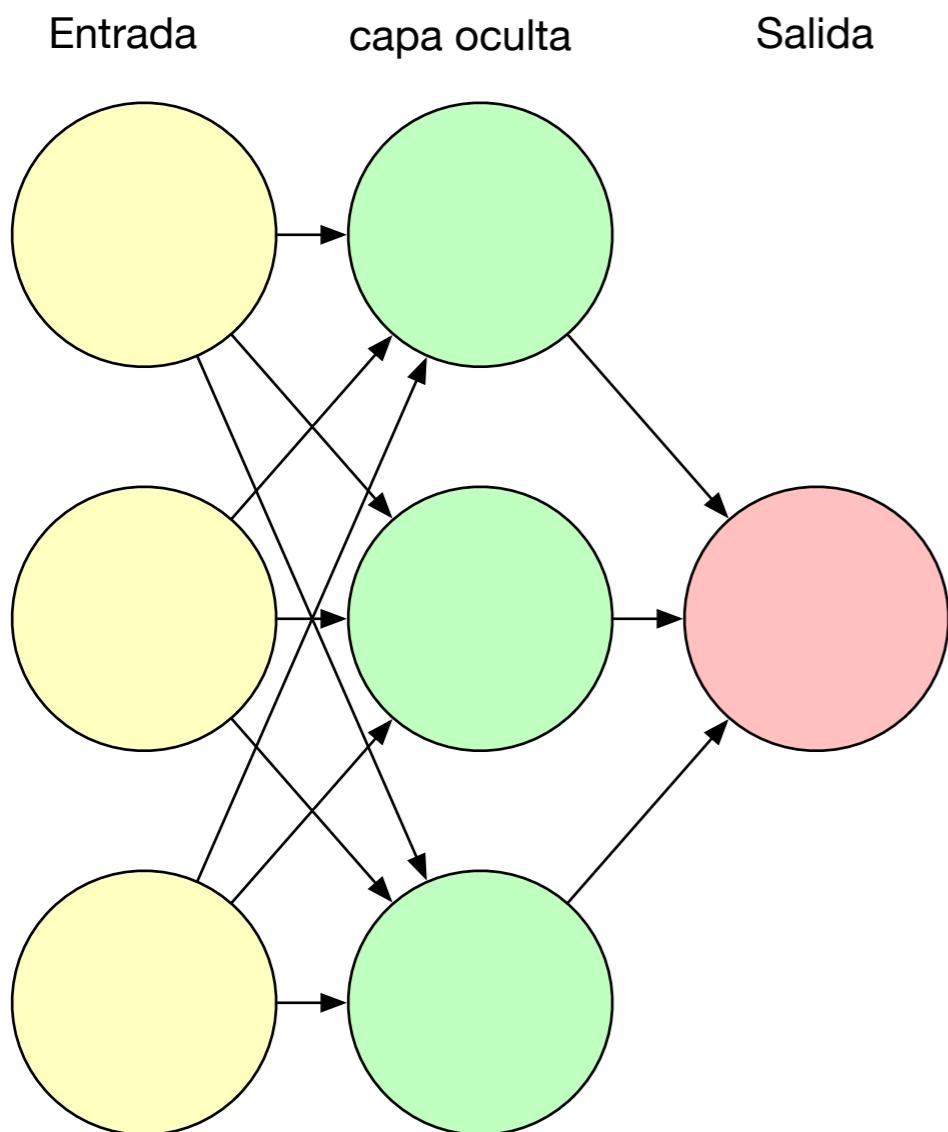
- ❖ **Importante:** Este proceso de normalización hay que hacerlo en los datos de entrenamiento como también en los de testing. Pero los **parámetros estadísticos** solo se tienen que **calcular** sobre los datos de **entrenamiento**.

Por ejemplo, la media y std solo se calcula en los datos de entrenamiento y luego se aplica tanto a todos los datos de entrada de la red (tanto entrenamiento como test).

El problema de la Inicialización de los pesos de una red neuronal

Inicialización de los pesos

- ❖ Supongamos inicializamos los pesos en 0. ¿Todas las neuronas están muertas?



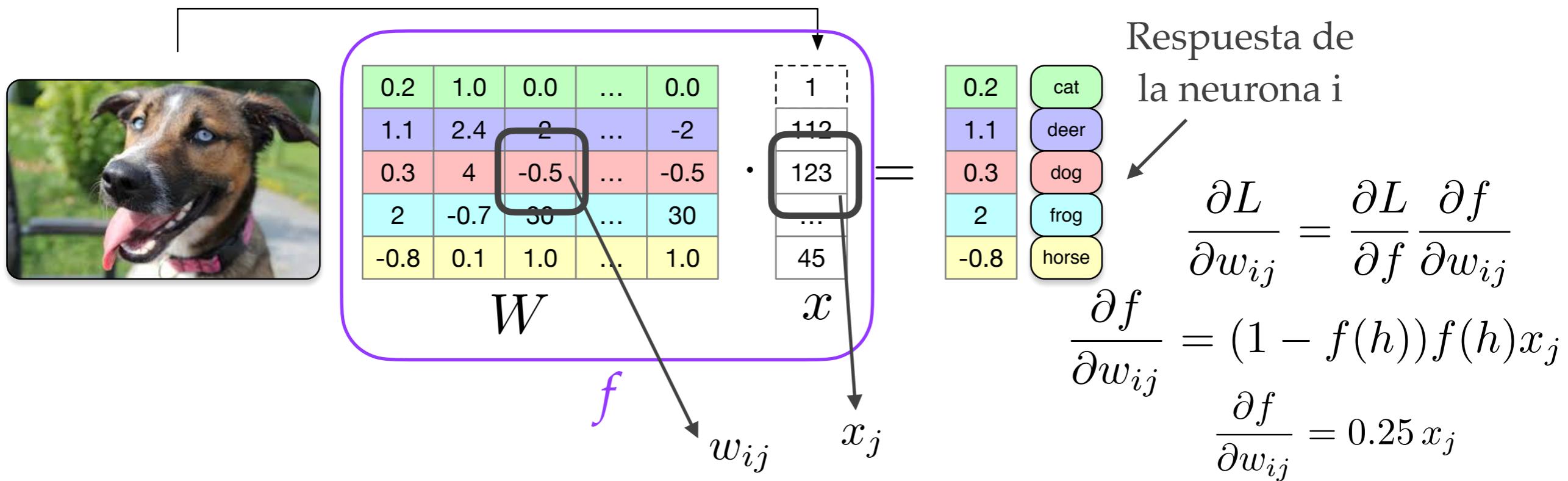
No, dependiendo de la función de activación y la entrada de la neurona puede estar operando en cualquier régimen

¿Cuál sería el problema?

Inicialización de los pesos

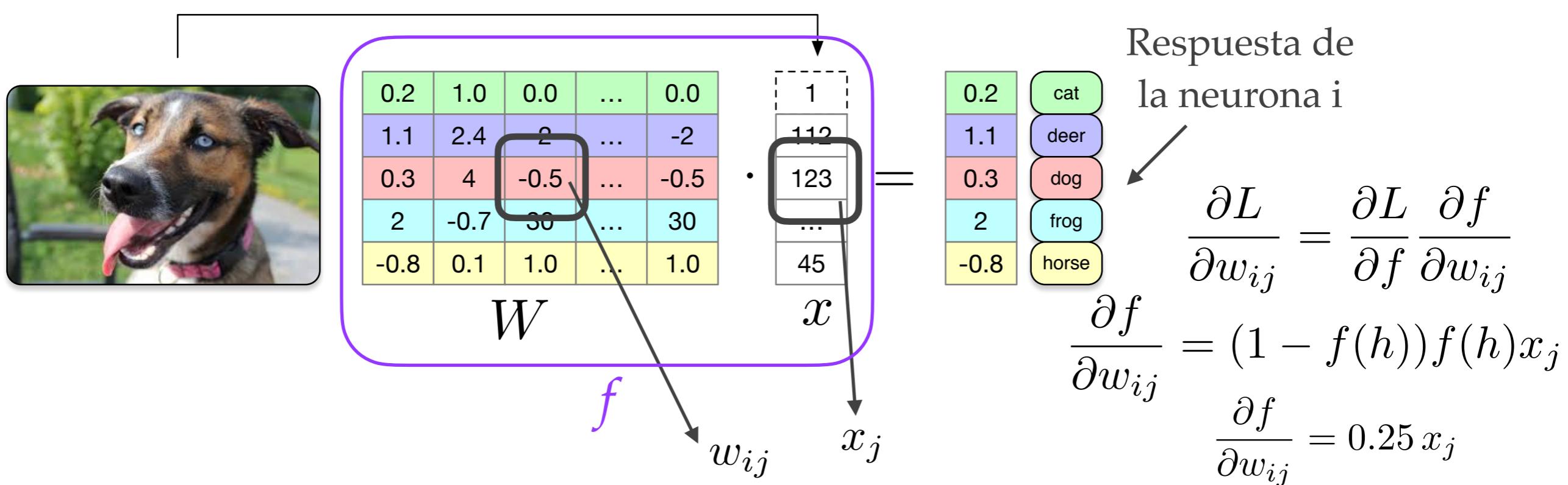
- ❖ Supongamos inicializamos los pesos en 0. ¿Todas las neuronas están muertas?

Ej. supongamos $f(x)$ es una sigmoide. Inicialmente los pesos están en 0. Todas las neuronas van a recibir la misma actualización para el input j , ya que el gradiente es igual.



Inicialización de los pesos

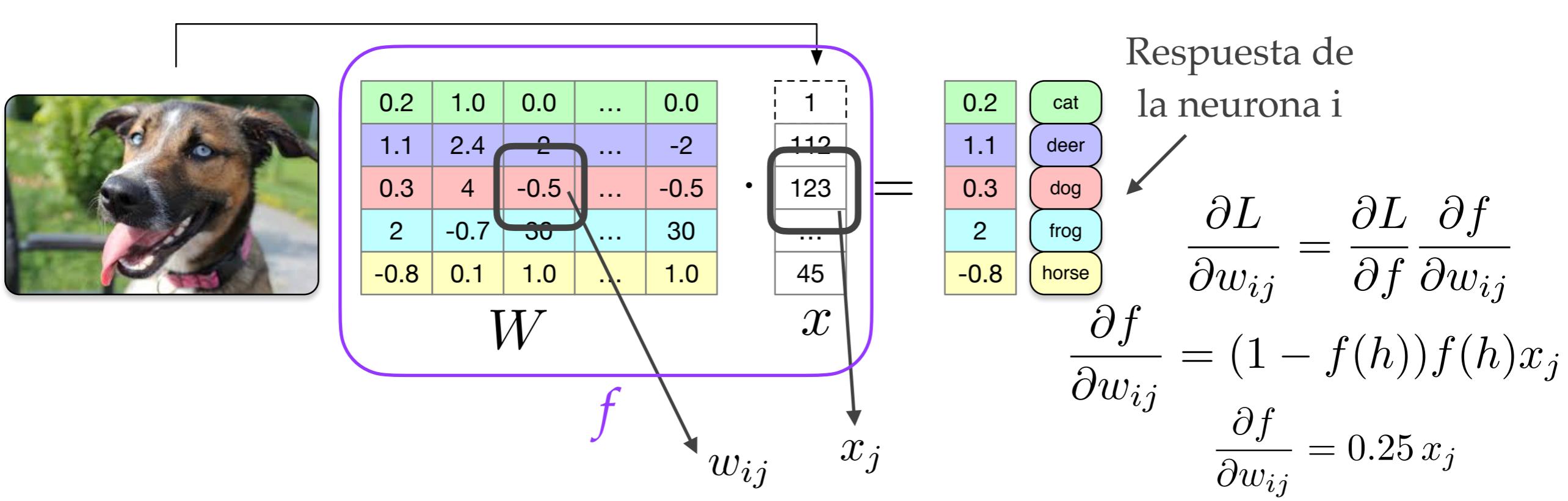
- ❖ Supongamos inicializamos los pesos en 0. ¿Todas las neuronas están muertas?
¿Qué es lo que realmente pasa?
La red NO aprende
- Todas las neuronas se comportan igual. Tienen las mismas salida y las mismas actualizaciones



Inicialización de los pesos

- ❖ ¿Qué pasa si inicializamos con los mismos peso toda la red?

Lo mismo de antes, la red NO aprende



Inicialización de los pesos

- ❖ Inicializamos los pesos con valores pequeños ~ 0 de forma aleatoria. Por ejemplo con una desviación estándar pequeña ($1e-3$ o $1e-2$) y media cero.
- ❖ Funciona \sim bien en redes pequeñas. Bastante mal para deep networks ya que mata las neuronas

Inicialización de los pesos

- ❖ 10 capas
 - ❖ 500 neuronas
 - ❖ Tanh
- Implementación no OO para ver la activación de las neuronas cuando la inicialización es pequeña

Ojo acá no hay actualización de pesos

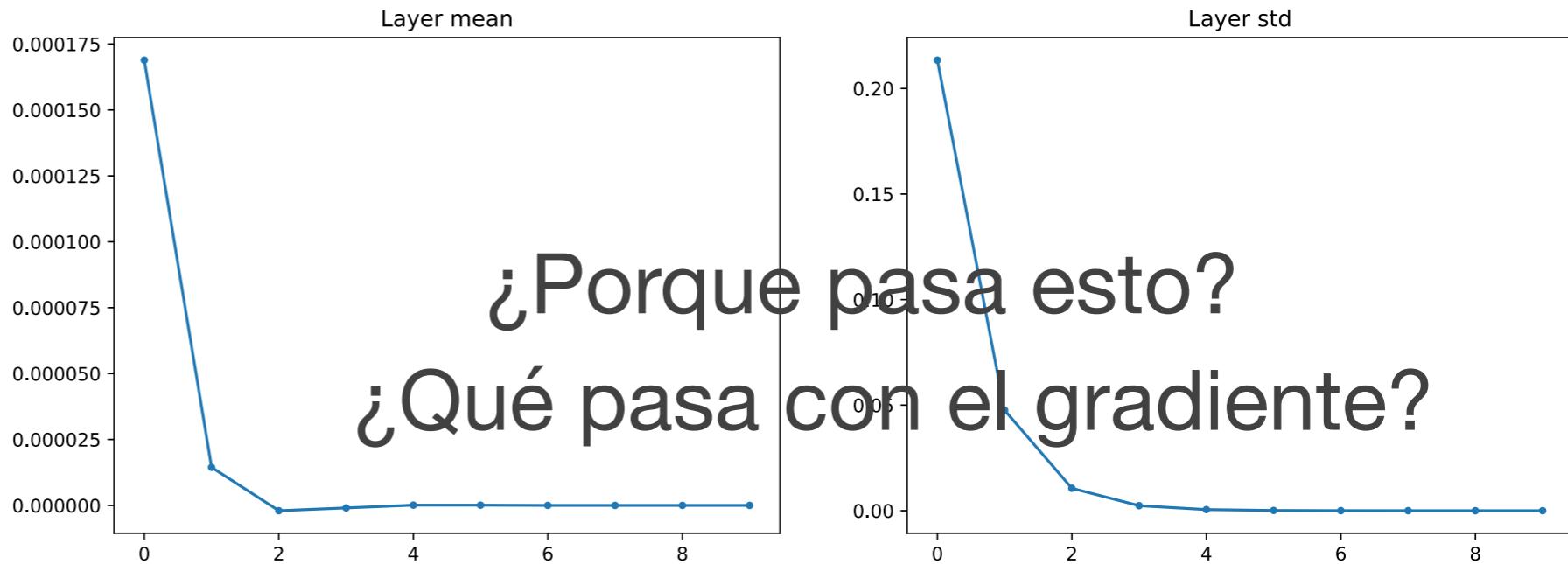
```
1 n_layers = 10
2 n_neurons=500
3 n_samples = 10000
4 D = np.random.randn(n_samples, n_neurons)
5 n_neurons_by_layer = [n_neurons] * n_layers
6 act = {'relu': lambda x:np.maximum(0,x), 'tanh': lambda x:np.tanh(x)}
7 # Como todas las capas tienen la misma cantidad de neuronas puedo agrupar todo
8 # en una gran matriz
9 Hs = np.zeros((n_samples, n_neurons, n_layers))
10 for i in range(n_layers):
11     X = D if i==0 else Hs[..., i-1]
12     fan_in = X.shape[-1] # neuronas de la capa anterior
13     fan_out = n_neurons_by_layer[i] # neuronas de la capa i
14     # Inicializamos con media cero y std 1e-2
15     W = np.random.randn(fan_in, fan_out) * 1e-2
16     H = X.dot(W) # producto interno
17     H = act['tanh'](H)
18     Hs[..., i] = H
19
20 layer_means = Hs.mean(axis=(0,1))
21 layer_stds = Hs.std(axis=(0,1))
22 print('Datos mean {:.6f} std {:.6f}'.format(D.mean(), D.std()))
23 for i in range(n_layers):
24     print('Layer {} \t mean {:.6f} \t std {:.6f}'.format(i, layer_means[i],
25             layer_stds[i]))
26 plt.figure()
27 plt.plot(layer_means, '-.')
28 plt.title('Layer mean')
29 plt.figure()
30 plt.plot(layer_stds, '-.')
31 plt.title('Layer std')
32 plt.figure()
33 for i in range(n_layers):
34     plt.subplot(1, n_layers, i+1)
35     plt.hist(Hs[..., i].ravel(), bins=30, range=(-1,1))
36     plt.title('Layer {}'.format(i))
```

Inicialización de los pesos

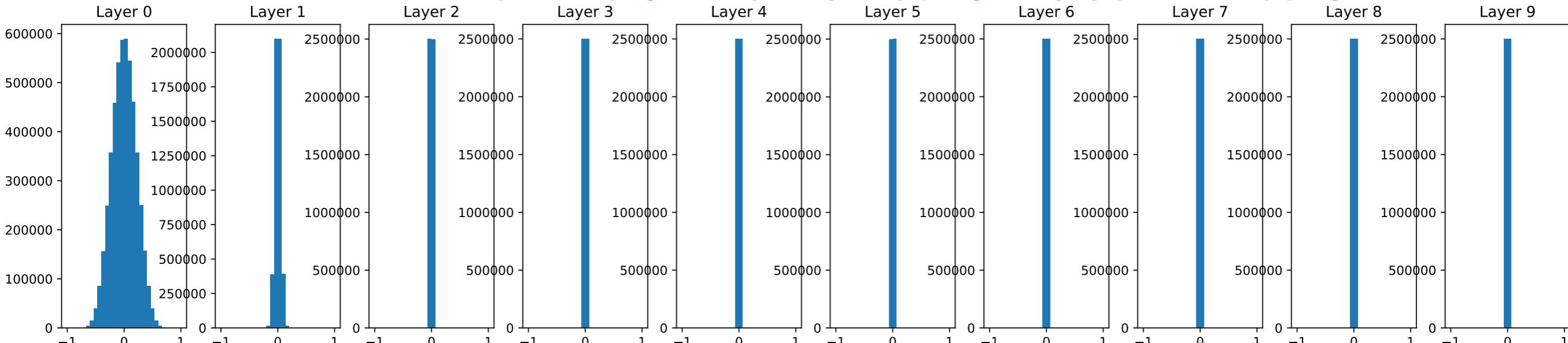
- ❖ Valores de activación promedio y std

Datos mean 0.000089 std 1.000056

Layer 0	mean 0.000169	std 0.213437
Layer 1	mean 0.000014	std 0.047578
Layer 2	mean -0.000002	std 0.010623
Layer 3	mean -0.000001	std 0.002379
Layer 4	mean 0.000000	std 0.000533
Layer 5	mean 0.000000	std 0.000119
Layer 6	mean 0.000000	std 0.000027
Layer 7	mean -0.000000	std 0.000006
Layer 8	mean 0.000000	std 0.000001
Layer 9	mean -0.000000	std 0.000000



La activación de las neuronas se va a cero

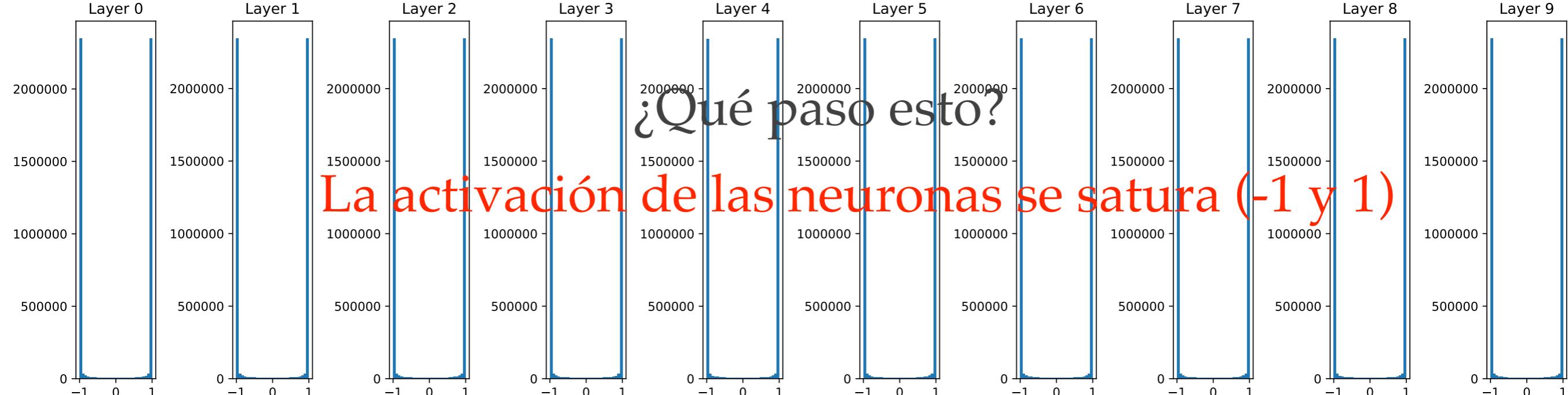
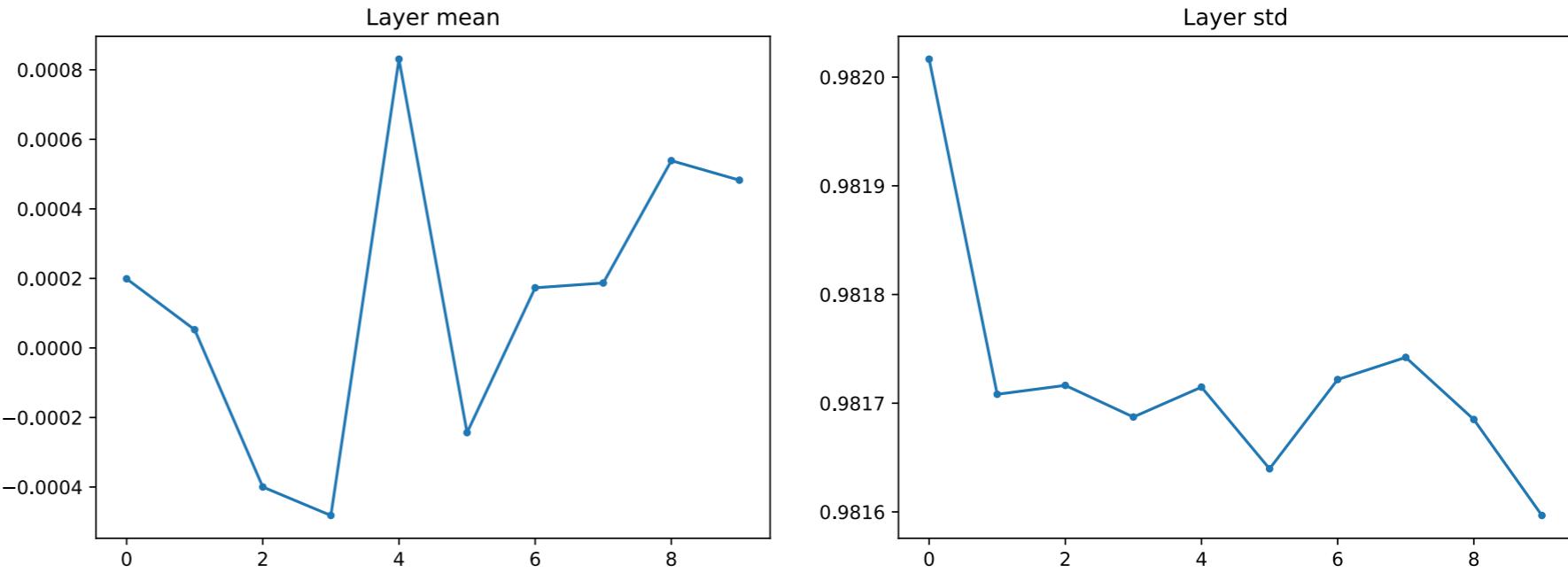


Inicialización de los pesos

- ❖ Aumentemos los pesos iniciales: std de 1e-2 a 1.

Datos mean 0.000925 std 1.000327

Layer 0	mean 0.000199	std 0.982017
Layer 1	mean 0.000052	std 0.981708
Layer 2	mean -0.000400	std 0.981716
Layer 3	mean -0.000482	std 0.981687
Layer 4	mean 0.000831	std 0.981715
Layer 5	mean -0.000244	std 0.981640
Layer 6	mean 0.000173	std 0.981722
Layer 7	mean 0.000187	std 0.981742
Layer 8	mean 0.000539	std 0.981685
Layer 9	mean 0.000483	std 0.981597



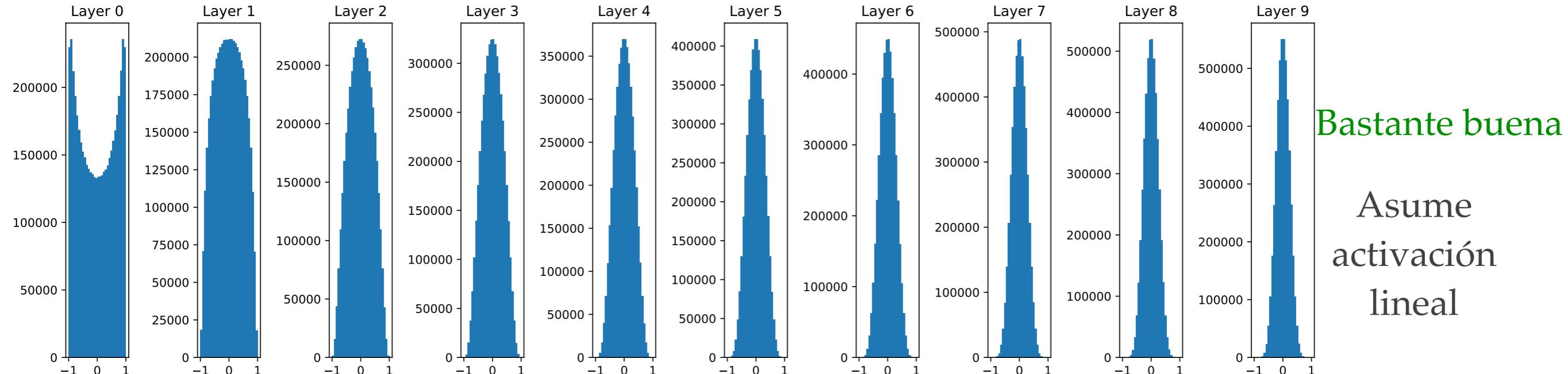
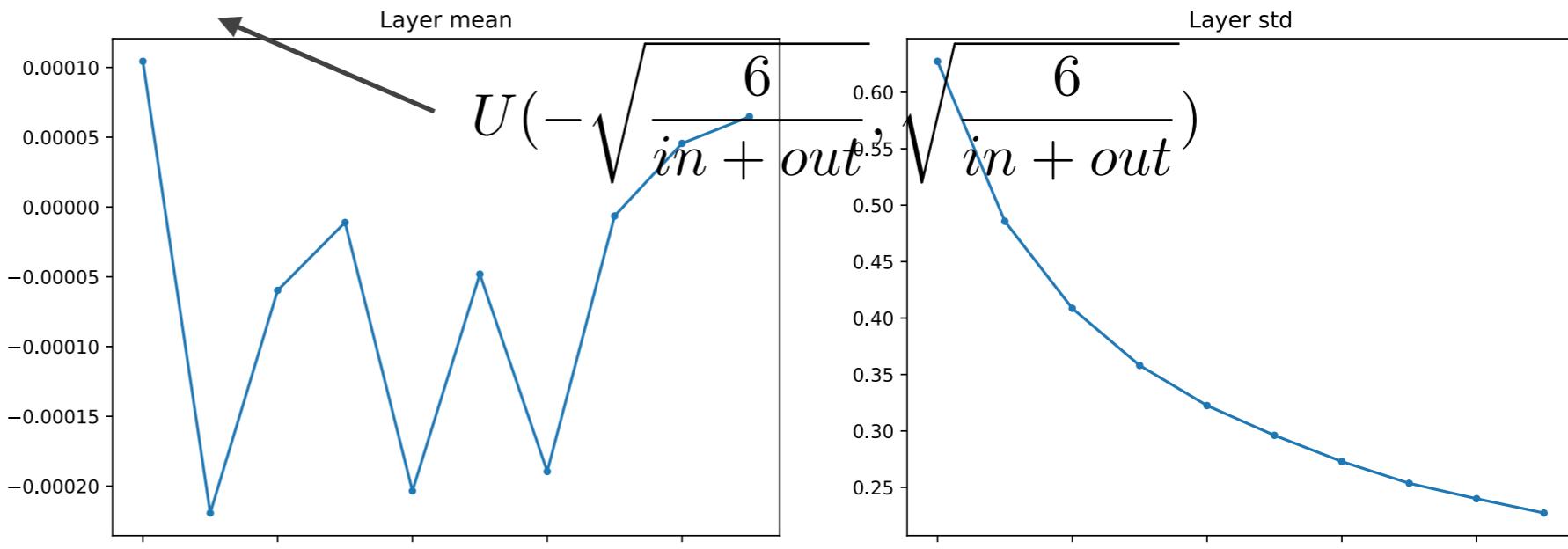
Inicialización de los pesos

- ❖ (tanh) Glorot_uniform: inicialización de Xavier [Glorot et al., 2010]

```
limit_uniform = [-np.sqrt(6/(fan_in+fan_out)), np.sqrt(6/(fan_in+fan_out))]
W = (limit_uniform[1] - limit_uniform[0]) * np.random.random((fan_in, fan_out)) + limit_uniform[0]
```

Datos mean 0.001048 std 0.999598

Layer 0	mean 0.000104	std 0.627455
Layer 1	mean -0.000219	std 0.485736
Layer 2	mean -0.000060	std 0.408628
Layer 3	mean -0.000011	std 0.358032
Layer 4	mean -0.000203	std 0.322473
Layer 5	mean -0.000048	std 0.296090
Layer 6	mean -0.000190	std 0.272879
Layer 7	mean -0.000006	std 0.253591
Layer 8	mean 0.000046	std 0.239951
Layer 9	mean 0.000065	std 0.227270



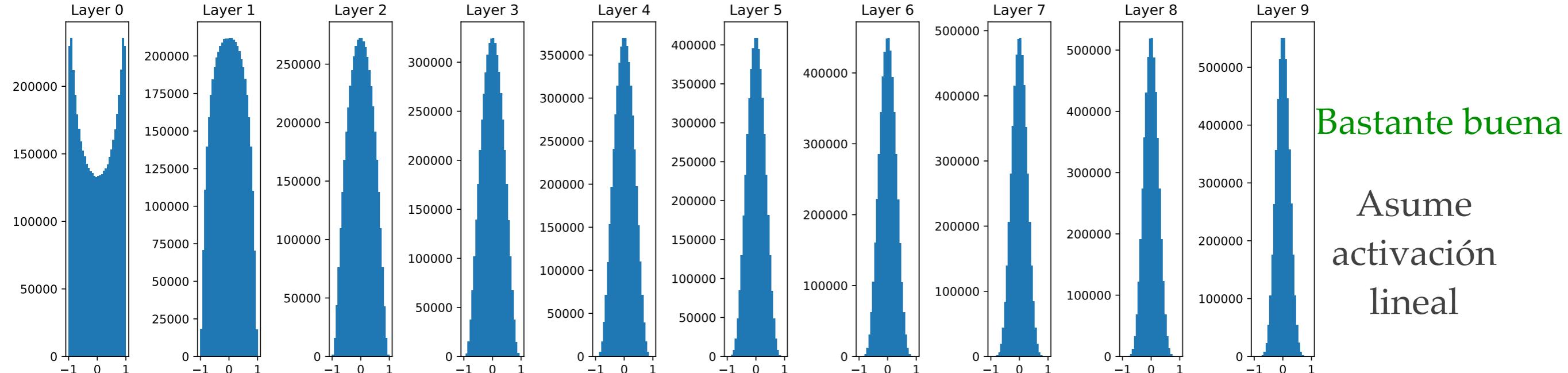
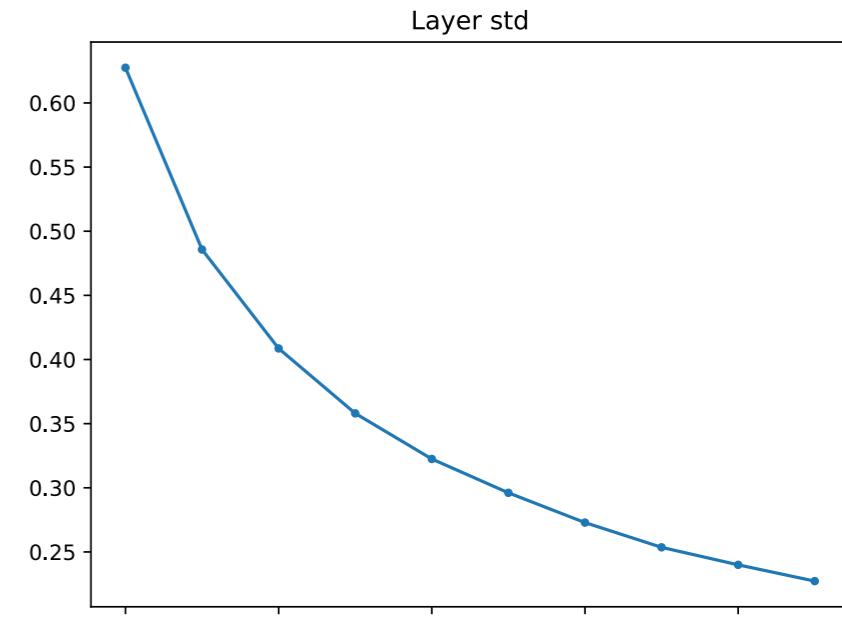
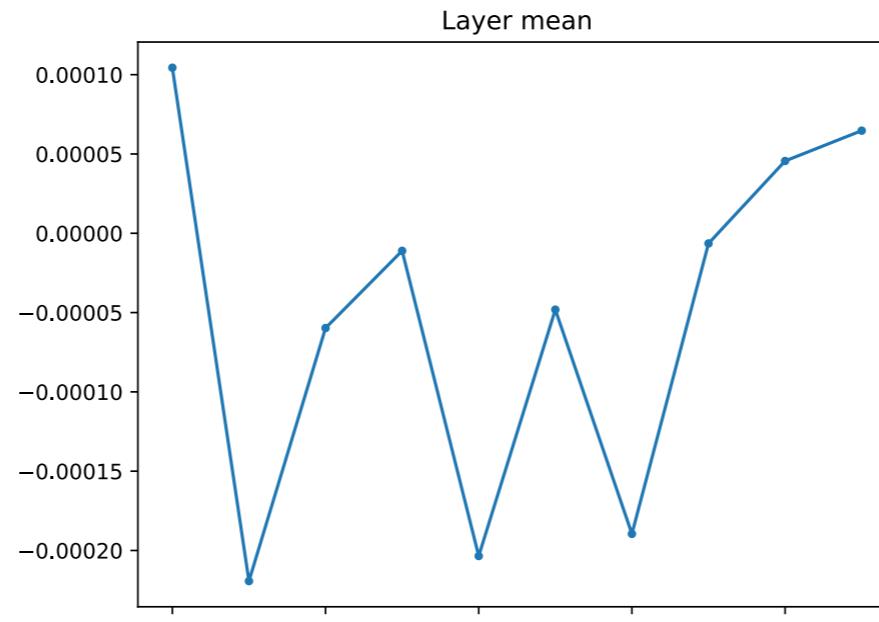
Inicialización de los pesos

- ❖ (tanh) Glorot_norm = Glorot_uniform [Glorot et al., 2010]

```
W = np.random.randn(fan_in, fan_out) * np.sqrt(2/(fan_in+fan_out))
```

Datos mean 0.001048 std 0.999598

Layer 0	mean 0.000104	std 0.627455
Layer 1	mean -0.000219	std 0.485736
Layer 2	mean -0.000060	std 0.408628
Layer 3	mean -0.000011	std 0.358032
Layer 4	mean -0.000203	std 0.322473
Layer 5	mean -0.000048	std 0.296090
Layer 6	mean -0.000190	std 0.272879
Layer 7	mean -0.000006	std 0.253591
Layer 8	mean 0.000046	std 0.239951
Layer 9	mean 0.000065	std 0.227270

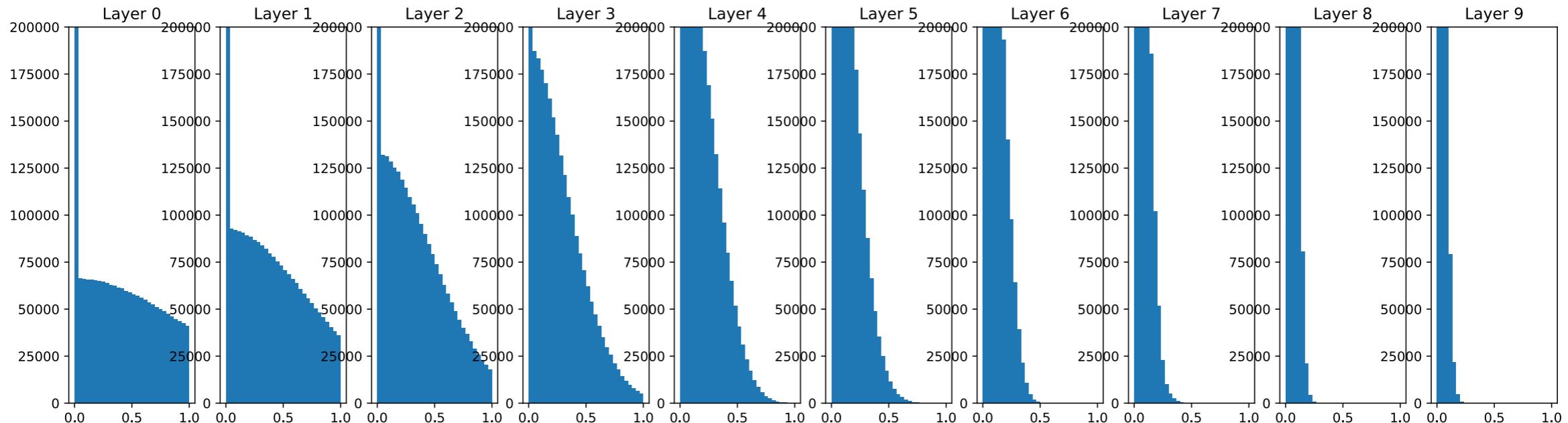
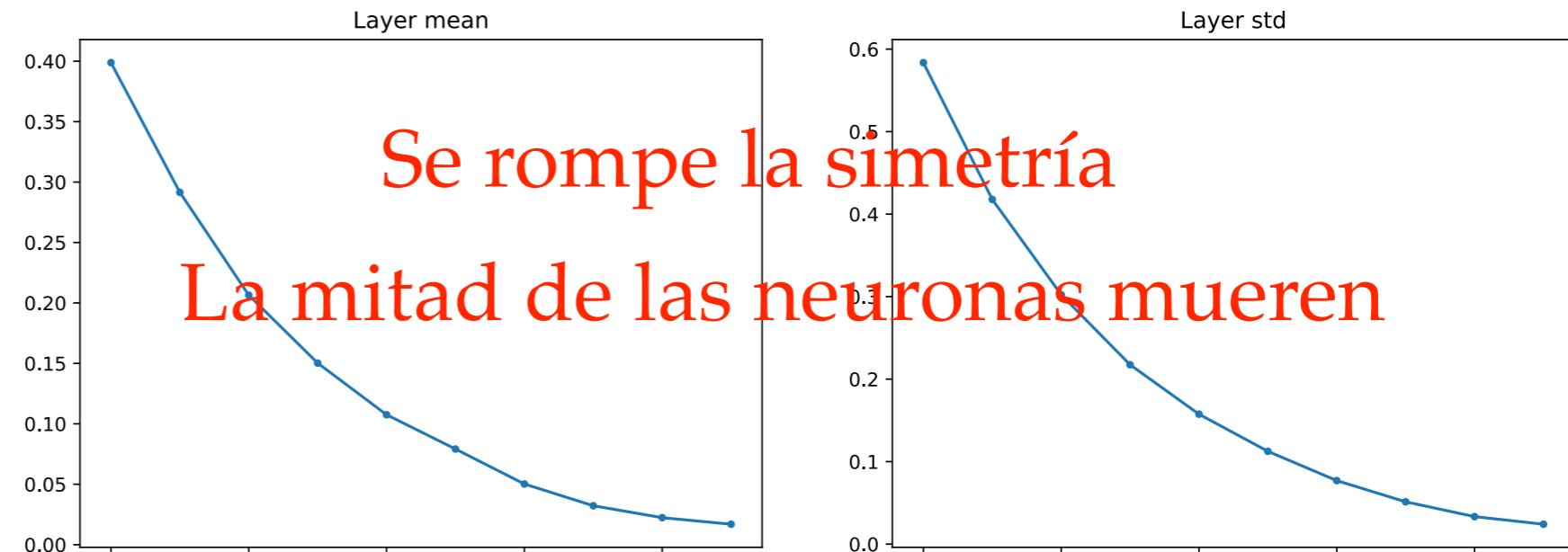


Inicialización de los pesos

- ❖ Glorot_uniform o Glorot_norm con activación ReLU

Datos mean 0.000030 std 1.000023

Layer 0	mean 0.398799	std 0.583589
Layer 1	mean 0.291489	std 0.417777
Layer 2	mean 0.206426	std 0.302370
Layer 3	mean 0.150248	std 0.217431
Layer 4	mean 0.107512	std 0.157537
Layer 5	mean 0.079131	std 0.112494
Layer 6	mean 0.050218	std 0.077003
Layer 7	mean 0.032226	std 0.051280
Layer 8	mean 0.022358	std 0.033343
Layer 9	mean 0.016938	std 0.024128



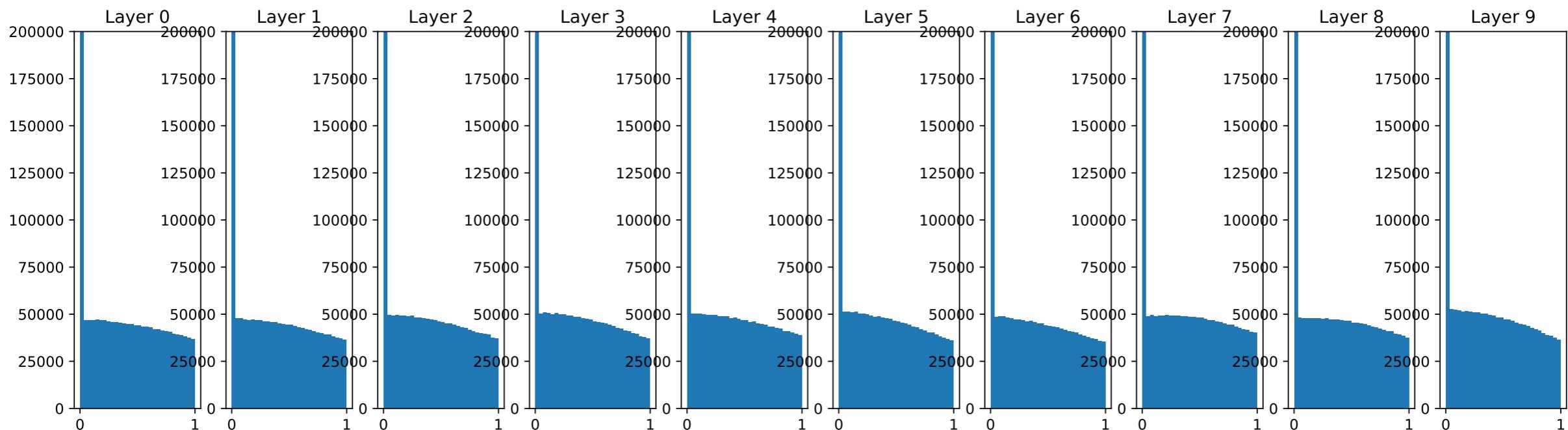
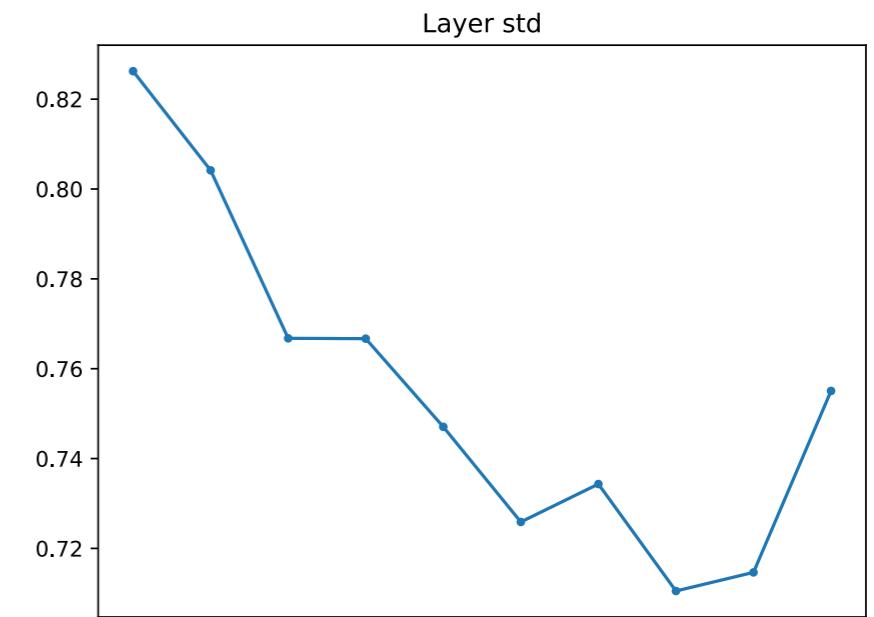
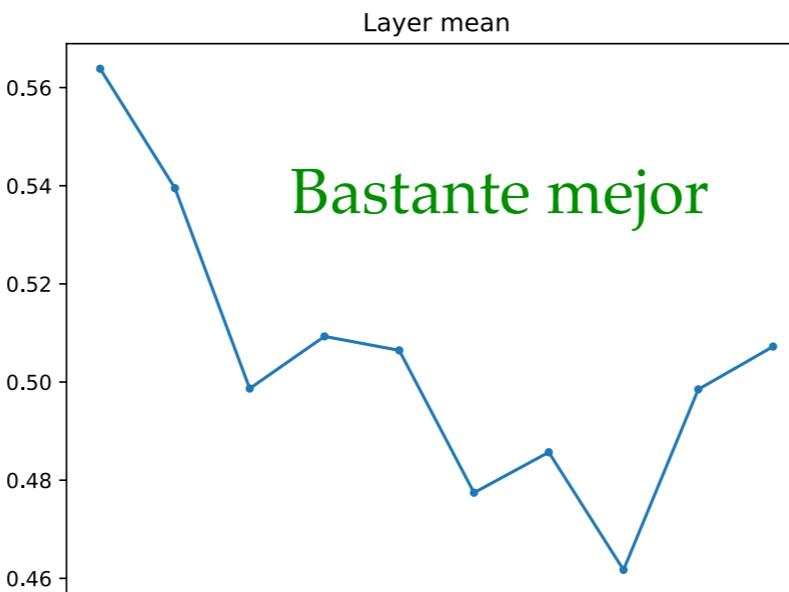
Inicialización de los pesos

- ❖ He et al., 2015 con activación ReLU

$W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2)$

Datos mean -0.000815 std 1.000866

Layer 0	mean 0.564415	std 0.82678
Layer 1	mean 0.534349	std 0.79922
Layer 2	mean 0.538379	std 0.79719
Layer 3	mean 0.513971	std 0.77983
Layer 4	mean 0.549288	std 0.78366
Layer 5	mean 0.541315	std 0.81966
Layer 6	mean 0.511459	std 0.78625
Layer 7	mean 0.571692	std 0.82060
Layer 8	mean 0.507583	std 0.77077
Layer 9	mean 0.484145	std 0.73925



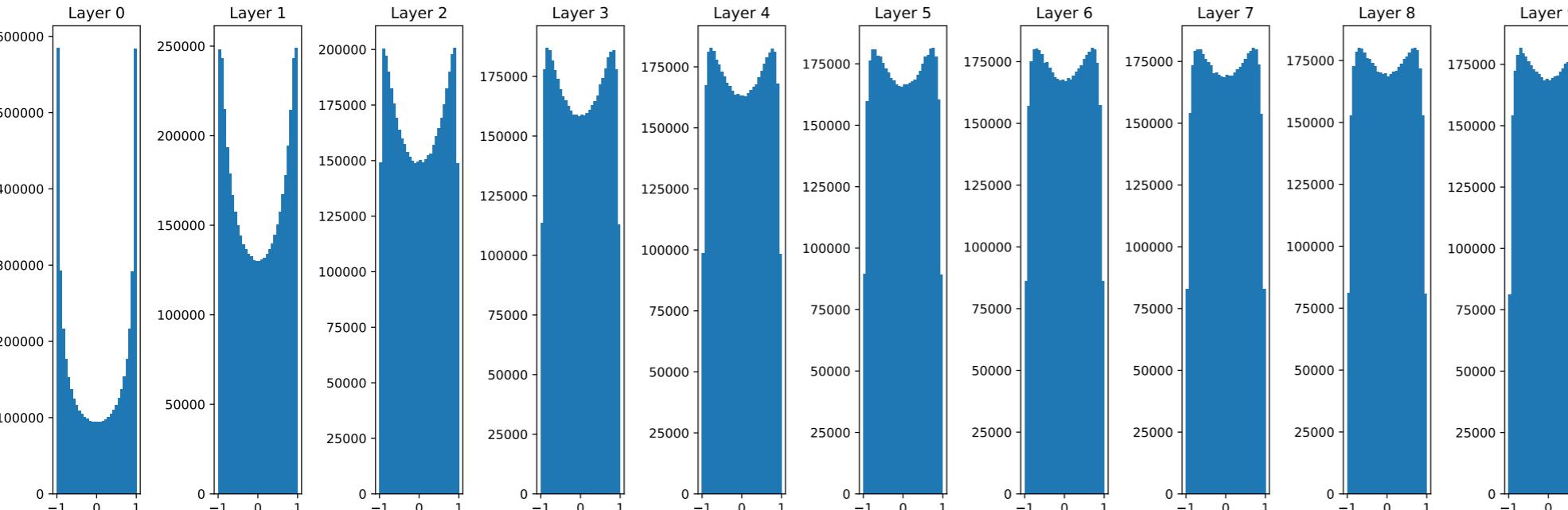
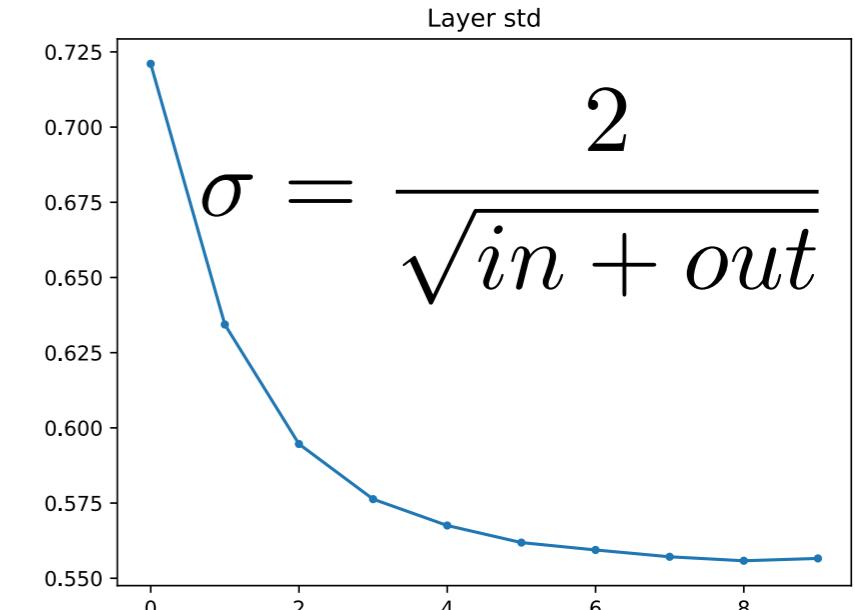
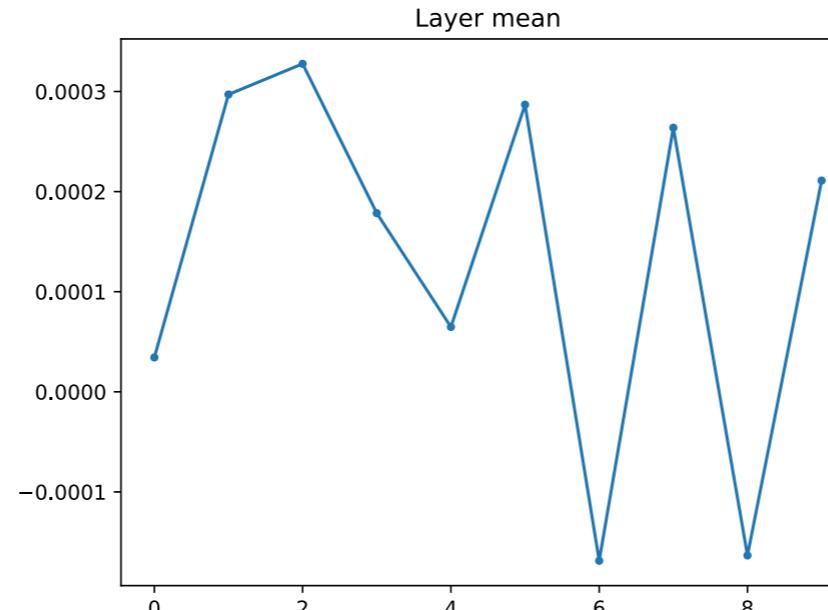
Inicialización de los pesos

- ❖ Glorot_normal V2 (`tanh`) tiene el mismo efecto que He et al.

```
W = np.random.randn(fan_in, fan_out) * 2/np.sqrt(fan_in+fan_out)
```

Datos mean 0.000110 std 1.000046

Layer 0	mean 0.000034	std 0.721043
Layer 1	mean 0.000297	std 0.634361
Layer 2	mean 0.000328	std 0.594643
Layer 3	mean 0.000178	std 0.576300
Layer 4	mean 0.000065	std 0.567542
Layer 5	mean 0.000287	std 0.561861
Layer 6	mean -0.000169	std 0.559411
Layer 7	mean 0.000264	std 0.557141
Layer 8	mean -0.000164	std 0.555816
Layer 9	mean 0.000211	std 0.556590



Bastante buena

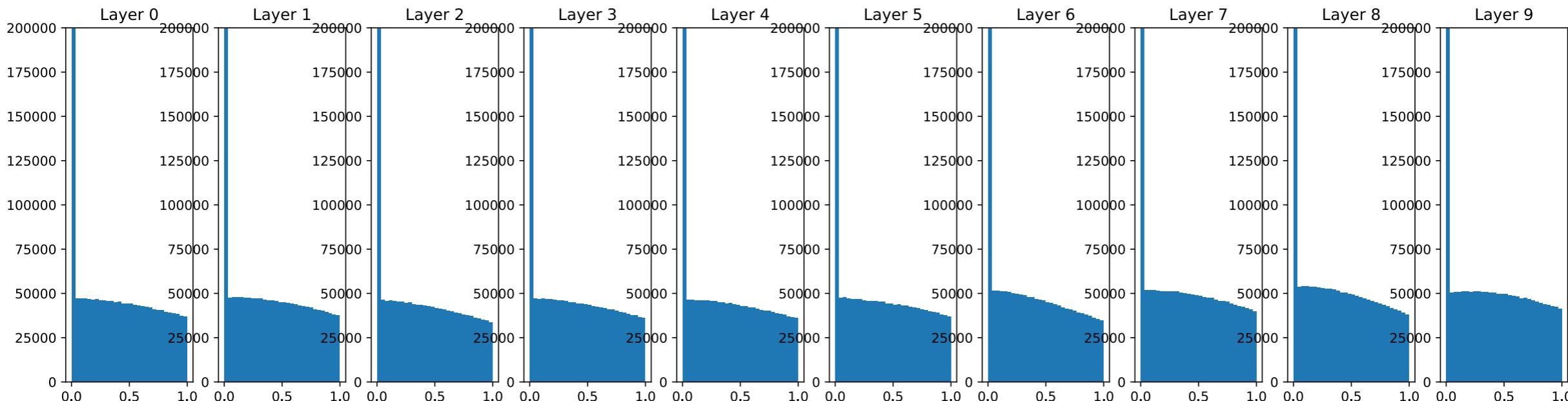
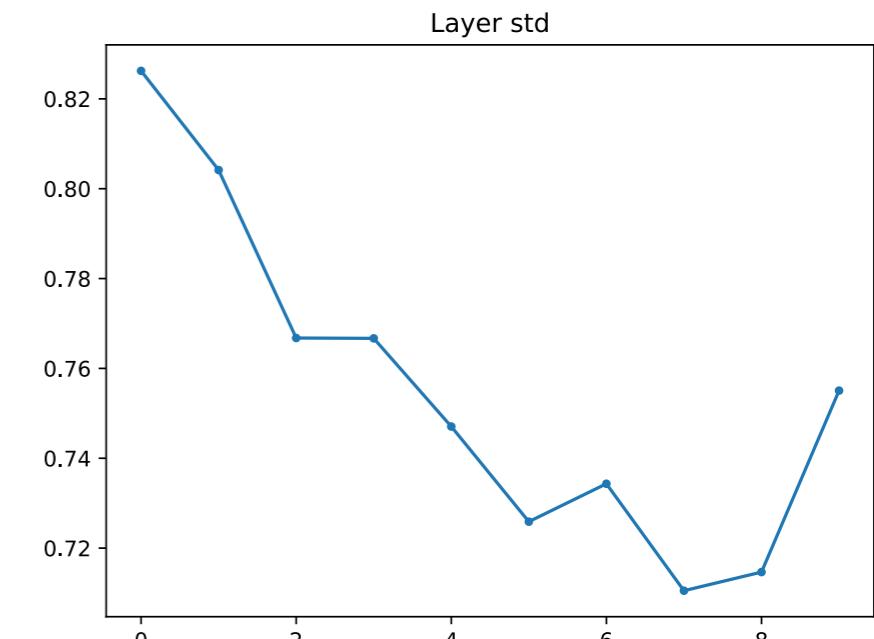
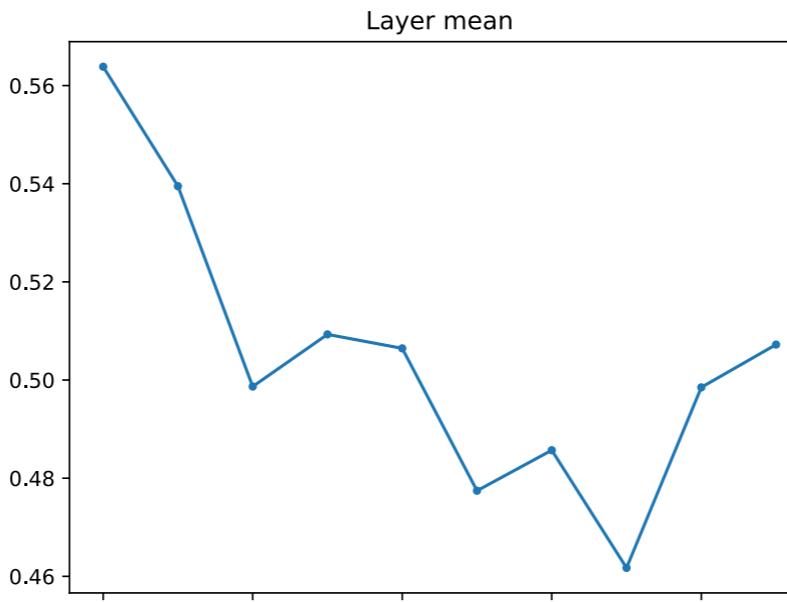
Inicialización de los pesos

- ❖ Glorot_normal V2 con activación ReLU

```
W = np.random.randn(fan_in, fan_out) * 2/np.sqrt(fan_in+fan_out)
```

Datos mean -0.000655 std 1.000799

Layer 0	mean 0.563841	std 0.82623
Layer 1	mean 0.539501	std 0.80415
Layer 2	mean 0.498673	std 0.76676
Layer 3	mean 0.509303	std 0.76669
Layer 4	mean 0.506454	std 0.74706
Layer 5	mean 0.477445	std 0.72588
Layer 6	mean 0.485699	std 0.73432
Layer 7	mean 0.461725	std 0.71051
Layer 8	mean 0.498508	std 0.71466
Layer 9	mean 0.507227	std 0.75505

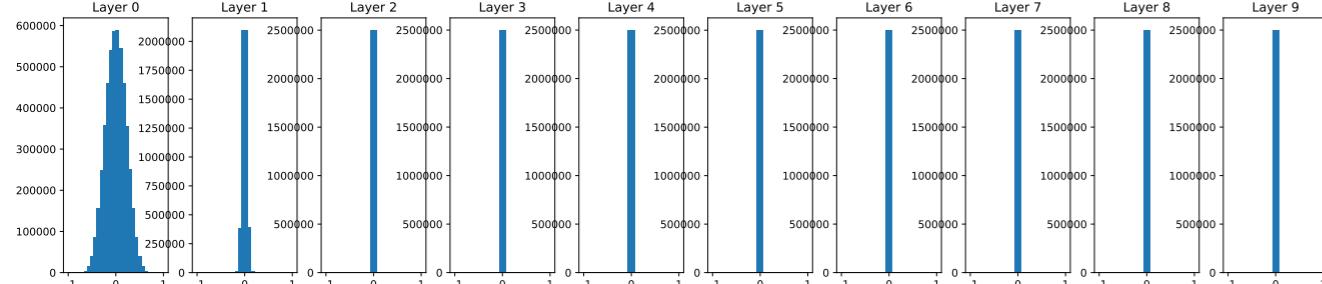


Batch Normalization

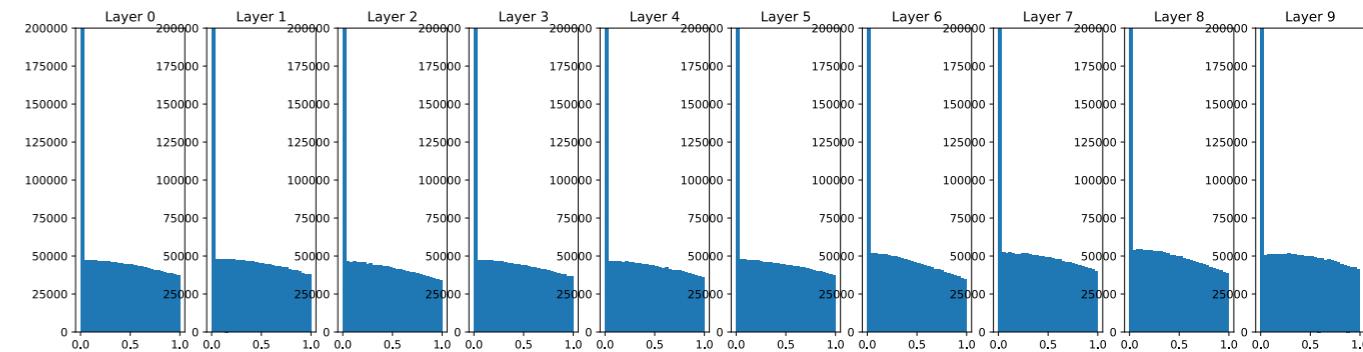
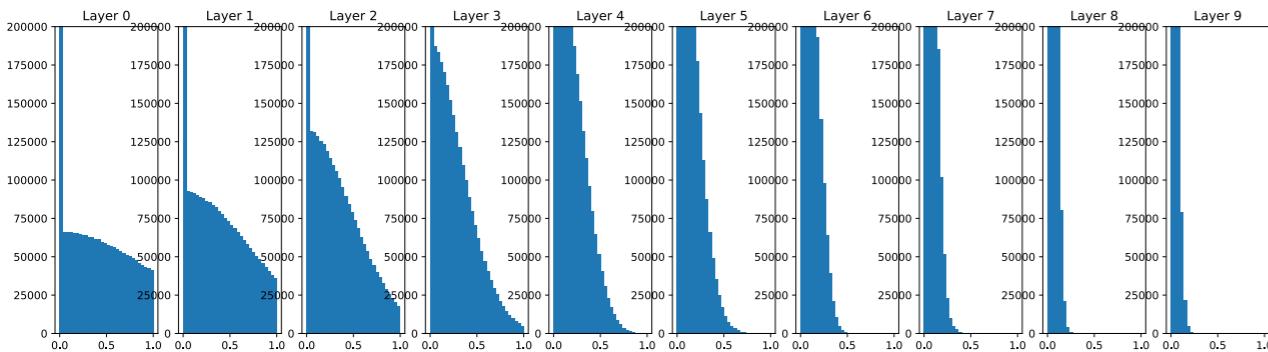
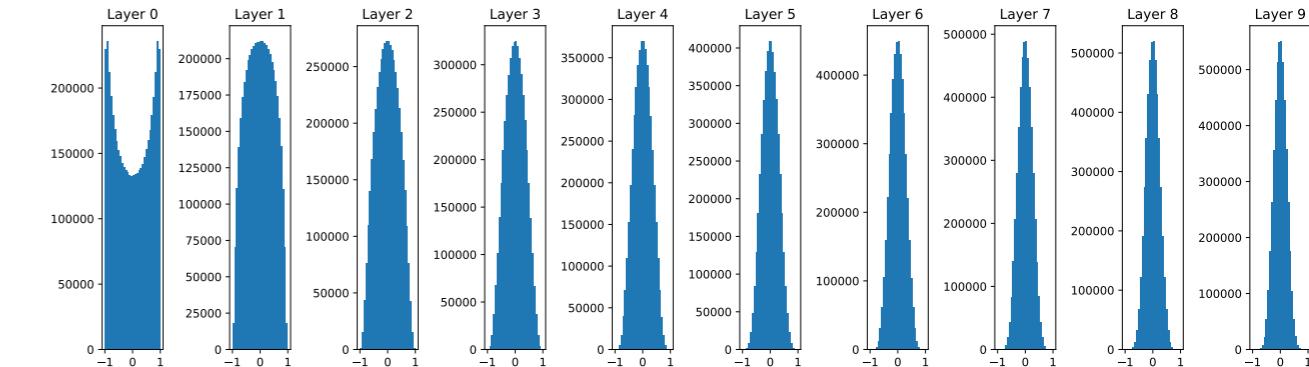
Batch Normalization

Recordemos cual era el problema de las activaciones

Malo



Bueno



Batch Normalization

[Ioffe and Szegedy, 2015]

- ❖ La idea es mantener la activación de cada capa en un rango de una distribución con media 0 y varianza 1

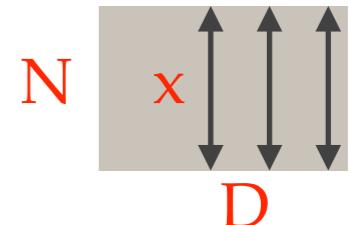
¿Cómo se logra?

Forzando a que la salida de una capa tenga una distribución con media 0 y varianza 1

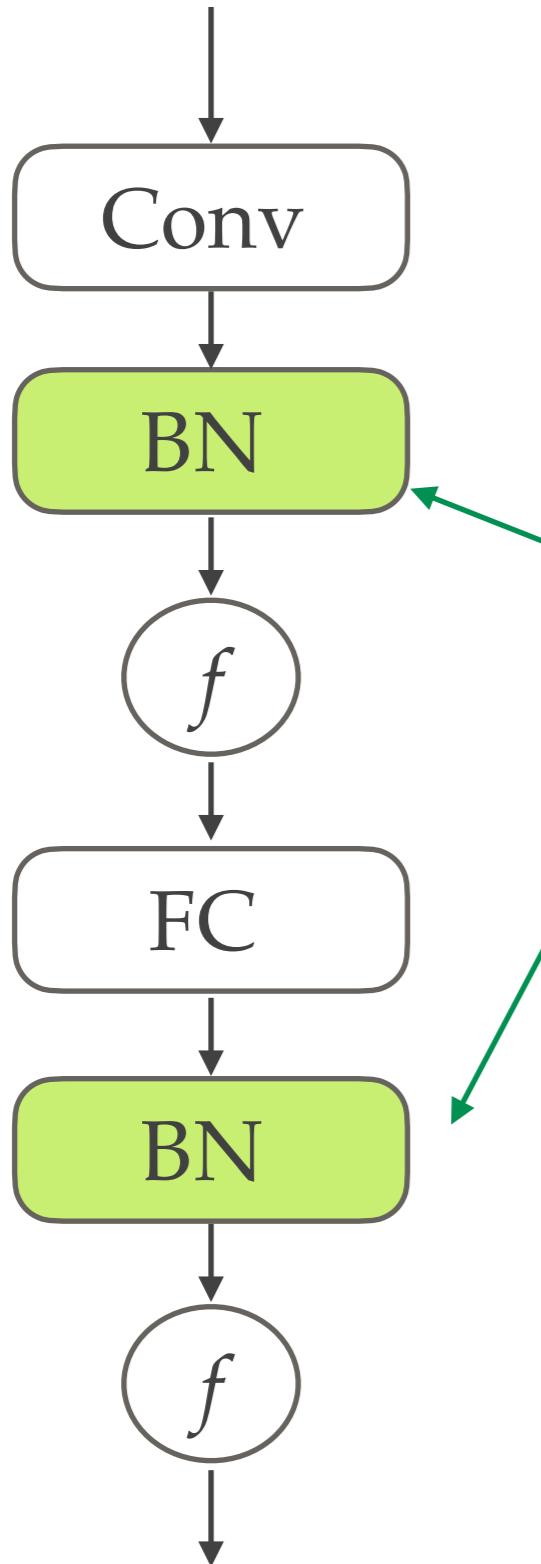
Activación de la capa k

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Del conjunto de datos o mini-batch, se calculan de forma independiente para cada dimensión

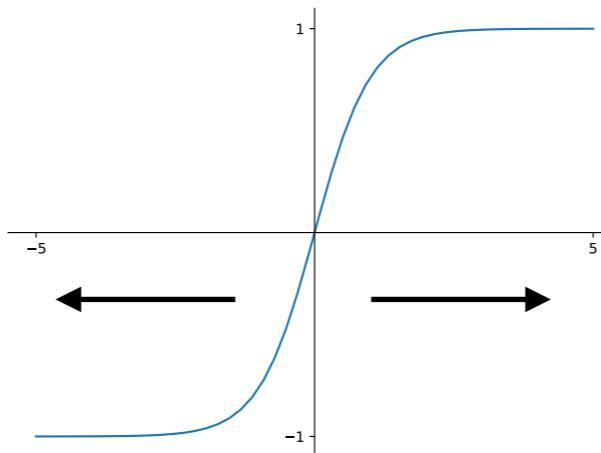


Batch Normalization



Normalmente se pone **luego** de las capas **densas** o **convolucionales** y **antes** de las **funciones de activación** no lineales.

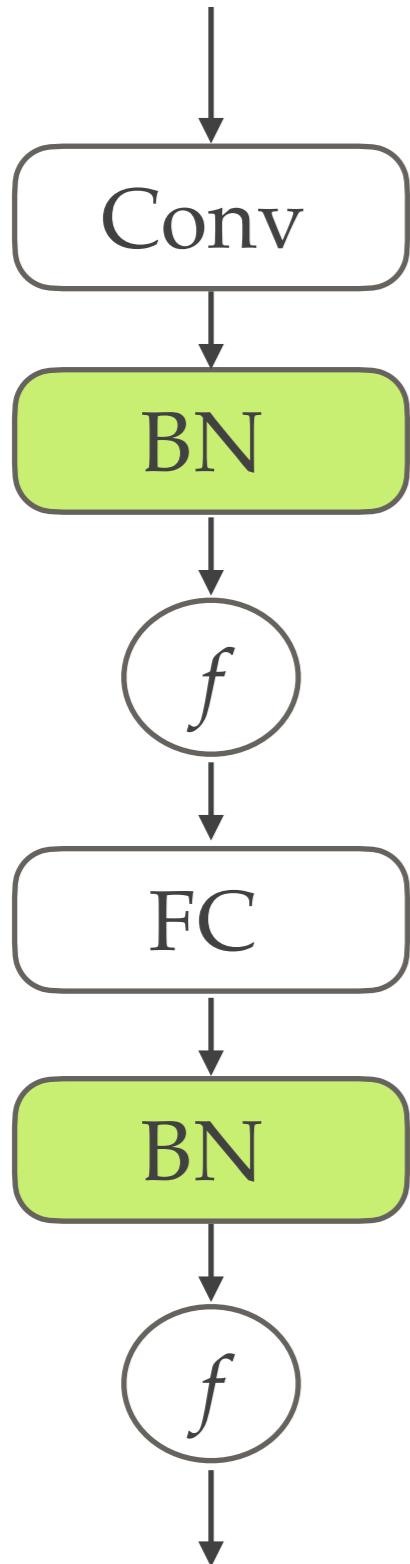
¿Cuál es el efecto de la normalización en funciones de activación como sigmoide o tanh?



Escala los datos, los lleva fuera del cero.

Puede llegar a **saturar** la activación.

Batch Normalization



¿Realmente siempre queremos hacer esto?

¿Cómo podemos evitarlo?

$$\text{Normalización} \quad \hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Estimamos la
activación original

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

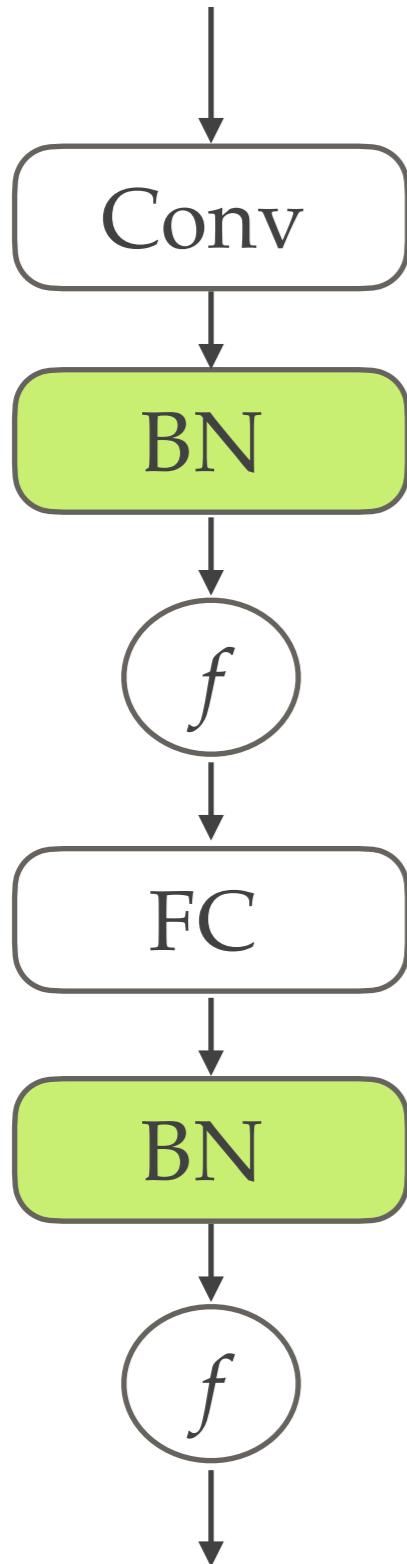
La red aprende estos parámetros y
decide cuando es necesario recuperar la
activación original

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

En la práctica, no se aprende la identidad.
Por tanto, no es redundante el BN

Batch Normalization



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

- ❖ Mejora el flujo del **gradiente** en la red
- ❖ Permite pasos de aprendizaje (**learning rates**) mas grande. Lo que implica aprendizaje más rápido.
- ❖ Actúa como una forma de **regularización** reduciendo la necesidad de Dropouts.

Con BN la activación o salida de cada layer depende no solo de la entrada, sino también del conjunto de datos o mini-batch debido a la normalización. Esto ocasiona una respuesta estocástica haciendo en parte la función de regularización

Batch Normalization

❖ Batch Normalization en la etapa de test

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\begin{aligned}\mu_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i && // \text{mini-batch mean} \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 && // \text{mini-batch variance} \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && // \text{normalize} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) && // \text{scale and shift}\end{aligned}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

La media y varianza son fijas.
No se calculan con los datos de test

Se estiman en la etapa de entrenamiento solamente y se fijan para la etapa de test

Por ejemplo se puede utilizar el promedio de los distintos mini-batch.

Batch Normalization

- ❖ Antes de BN era realmente un desafío lograr una buena convergencia de las redes profundas ($\sim > 10$)
- ❖ Algunas estrategias para este fin eran por ejemplo las propuestas en VGG y Google Net
- ❖ Estas estrategias ya no son necesarias para lograr que los modelos converjan

Antes de BN: VGG



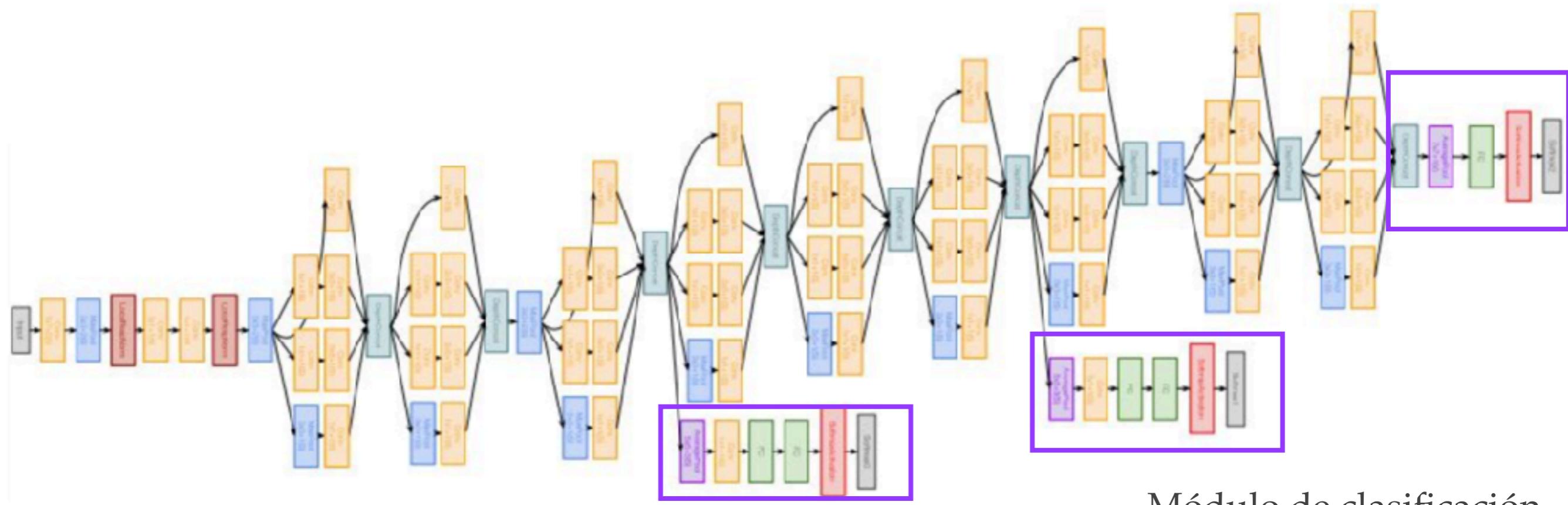
- ❖ Primero entrenaban un modelo con 11 capas donde se lograba una buena convergencia
- ❖ Luego agregaban otras capas intermedias (de forma aleatoria) hasta llegar a 16 por ejemplo.

VGG16

VGG19

Antes de BN: GoogLeNet

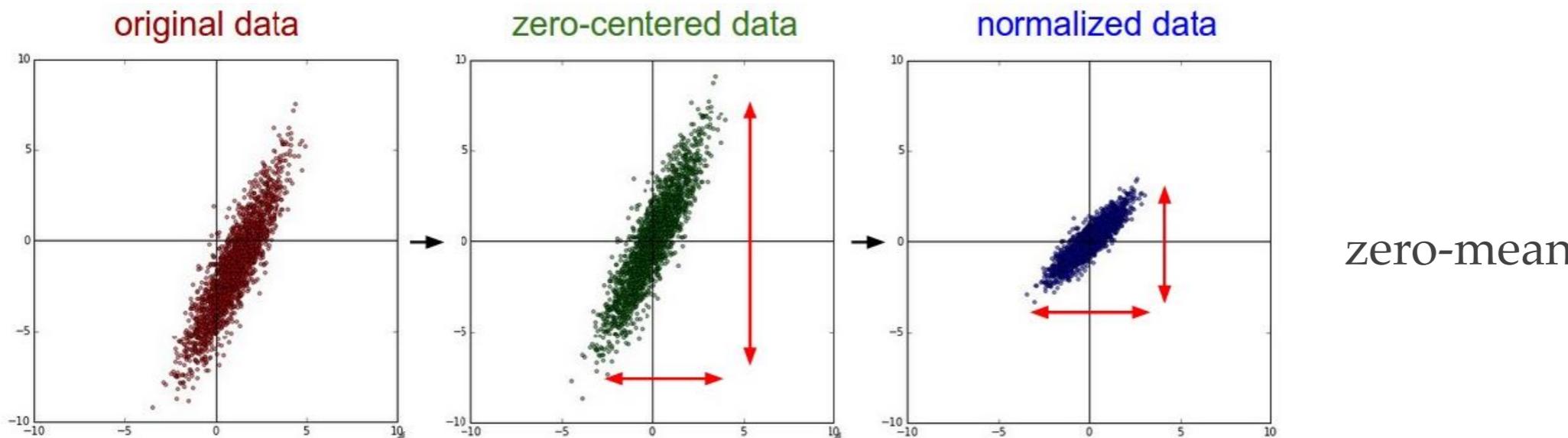
[Szegedy et al., 2014]



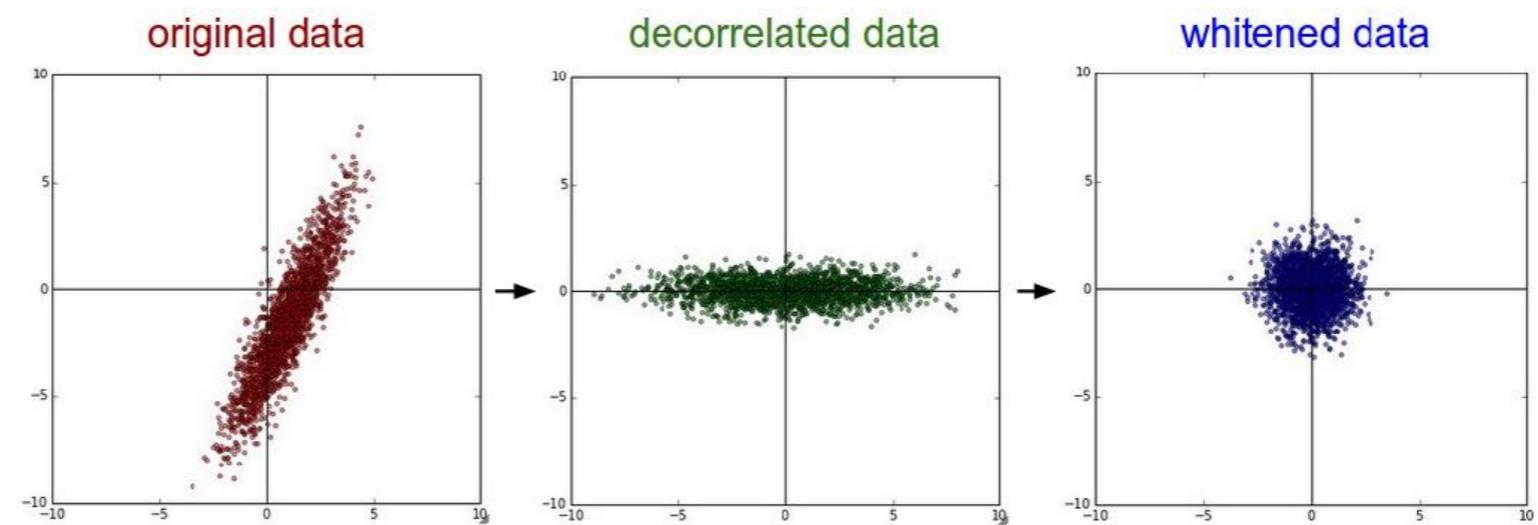
Estrategia de reinyección del gradiente mediante una clasificación adicional

Algunas consideraciones en el entrenamiento de redes neuronales

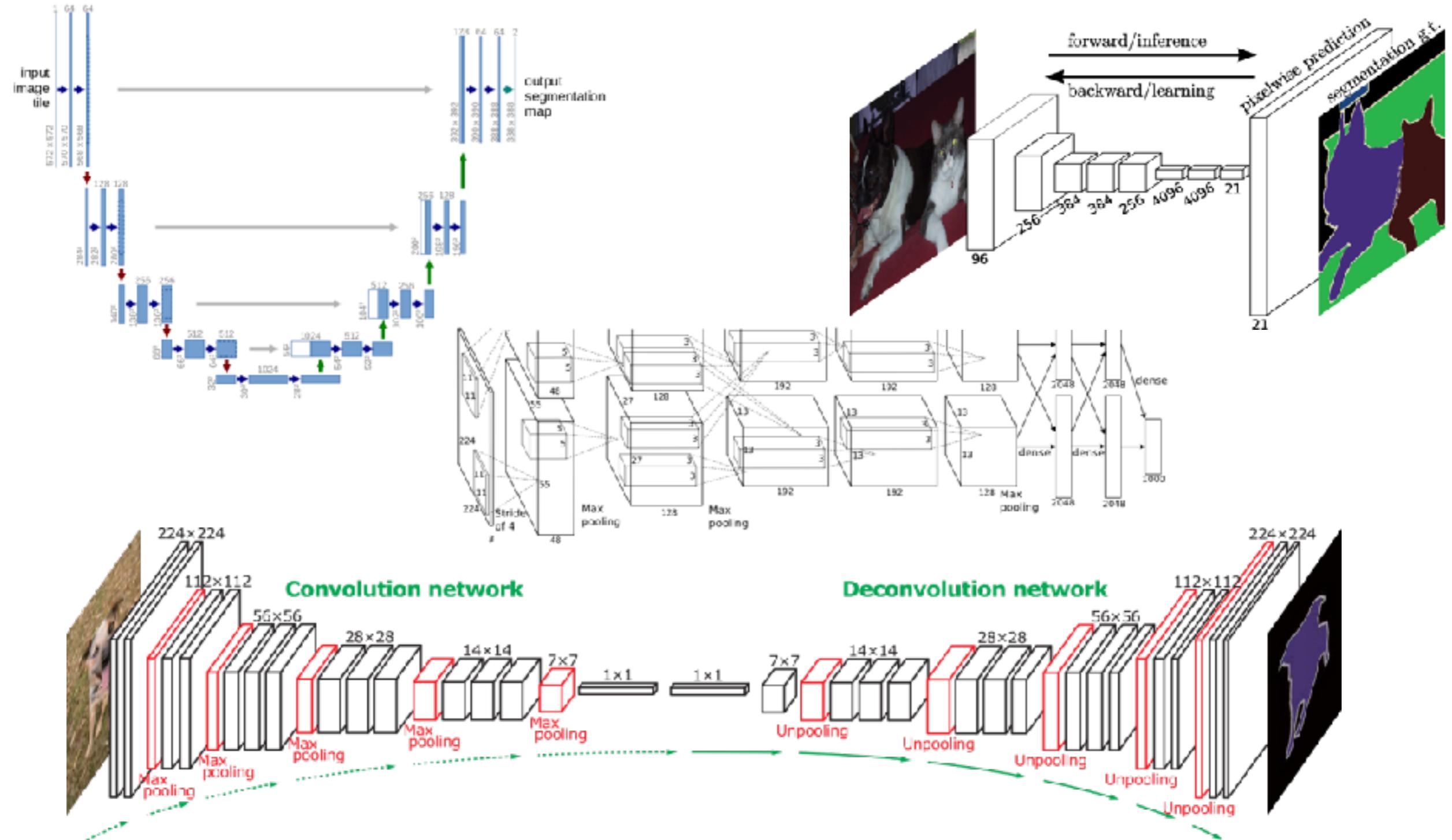
1. Preprocesado



PCA y whitening



2. Elegir la arquitectura



3. Verificación básica

- ❖ Verificamos que la función de costo se comporta de forma razonable:
 1. Inabilitamos la regularización y vemos que el resultado es consistente con lo que se mide (contar ej. dice).
 2. Comparamos un resultado con y sin regularización. La función de costo debería aumentar.

3. Verificación básica

- ❖ Para un conjunto reducido de entrenamiento (~20), entrenamos la red de forma de generar un overfitting con los datos (ajustar lr y cantidad de épocas). (problema 2.8)

```
In [31]: history = model.fit(x_tiny, y_tiny, 200, validation=[x_tiny, y_tiny], verbose=True)

epoch 001 loss: 0.809 mse:0.803 accuracy:0.600 val_loss:0.861 val_mse:0.855 val_accuracy:0.250
epoch 002 loss: 0.769 mse:0.763 accuracy:0.400 val_loss:0.852 val_mse:0.847 val_accuracy:0.250
epoch 003 loss: 0.756 mse:0.751 accuracy:0.450 val_loss:0.851 val_mse:0.846 val_accuracy:0.250
epoch 004 loss: 0.766 mse:0.761 accuracy:0.350 val_loss:0.838 val_mse:0.832 val_accuracy:0.250
epoch 005 loss: 0.745 mse:0.740 accuracy:0.450 val_loss:0.838 val_mse:0.833 val_accuracy:0.400
epoch 006 loss: 0.722 mse:0.717 accuracy:0.400 val_loss:0.845 val_mse:0.840 val_accuracy:0.150
epoch 007 loss: 0.749 mse:0.743 accuracy:0.450 val_loss:0.816 val_mse:0.811 val_accuracy:0.250
epoch 008 loss: 0.688 mse:0.683 accuracy:0.400 val_loss:0.834 val_mse:0.828 val_accuracy:0.250
epoch 009 loss: 0.695 mse:0.690 accuracy:0.550 val_loss:0.803 val_mse:0.797 val_accuracy:0.350
epoch 010 loss: 0.686 mse:0.681 accuracy:0.450 val_loss:0.785 val_mse:0.779 val_accuracy:0.250
.....
epoch 050 loss: 0.111 mse:0.105 accuracy:1.000 val_loss:0.130 val_mse:0.124 val_accuracy:0.950
epoch 051 loss: 0.106 mse:0.100 accuracy:1.000 val_loss:0.126 val_mse:0.120 val_accuracy:0.950
epoch 052 loss: 0.104 mse:0.099 accuracy:1.000 val_loss:0.118 val_mse:0.112 val_accuracy:1.000
epoch 053 loss: 0.096 mse:0.091 accuracy:1.000 val_loss:0.115 val_mse:0.109 val_accuracy:0.950
epoch 054 loss: 0.095 mse:0.089 accuracy:1.000 val_loss:0.109 val_mse:0.103 val_accuracy:0.950
epoch 055 loss: 0.093 mse:0.087 accuracy:1.000 val_loss:0.104 val_mse:0.098 val_accuracy:1.000
epoch 056 loss: 0.088 mse:0.083 accuracy:1.000 val_loss:0.101 val_mse:0.095 val_accuracy:1.000
epoch 057 loss: 0.085 mse:0.079 accuracy:1.000 val_loss:0.097 val_mse:0.091 val_accuracy:1.000
epoch 058 loss: 0.083 mse:0.077 accuracy:1.000 val_loss:0.093 val_mse:0.087 val_accuracy:1.000
epoch 059 loss: 0.076 mse:0.070 accuracy:1.000 val_loss:0.091 val_mse:0.085 val_accuracy:1.000
epoch 060 loss: 0.077 mse:0.071 accuracy:1.000 val_loss:0.088 val_mse:0.082 val_accuracy:1.000
```

4. Configuración de los hiperparámetros

- ❖ Comenzamos con el hiperparámetro mas importante, la tasa de aprendizaje o learning rate.
 1. Definimos que optimizador vamos a usar y dejamos todos sus parámetros por defecto.
 2. En caso de utilizar regularización, comenzamos con un valor pequeño $\sim 1e-5$.

4. Configuración de los hiperparámetros

- ❖ Para todo el conjunto de entrenamiento observamos la evolución de la función de costo. Comenzamos con un lr~1e-5 (buscamos un paso pequeño, depende del optim. y el problema/arquitectura).

Learning rate muy chico

```
In [48]: history = model.fit(x_train, y_train, 200, validation=[x_val, y_val], verbose=True)
epoch 001 loss: 2.321 mse:2.316 accuracy:0.100 val_loss:2.320 val_mse:2.315 val_accuracy:0.100
epoch 002 loss: 2.319 mse:2.314 accuracy:0.100 val_loss:2.318 val_mse:2.313 val_accuracy:0.100
epoch 003 loss: 2.317 mse:2.312 accuracy:0.100 val_loss:2.316 val_mse:2.311 val_accuracy:0.100
epoch 004 loss: 2.315 mse:2.310 accuracy:0.100 val_loss:2.314 val_mse:2.309 val_accuracy:0.100
epoch 005 loss: 2.313 mse:2.308 accuracy:0.100 val_loss:2.312 val_mse:2.307 val_accuracy:0.100
epoch 006 loss: 2.311 mse:2.306 accuracy:0.100 val_loss:2.310 val_mse:2.305 val_accuracy:0.100
epoch 007 loss: 2.309 mse:2.304 accuracy:0.100 val_loss:2.308 val_mse:2.303 val_accuracy:0.100
epoch 008 loss: 2.307 mse:2.302 accuracy:0.100 val_loss:2.306 val_mse:2.301 val_accuracy:0.100
.....
epoch 100 loss: 2.141 mse:2.136 accuracy:0.100 val_loss:2.141 val_mse:2.135 val_accuracy:0.100
epoch 101 loss: 2.140 mse:2.134 accuracy:0.100 val_loss:2.139 val_mse:2.134 val_accuracy:0.100
epoch 102 loss: 2.138 mse:2.133 accuracy:0.100 val_loss:2.137 val_mse:2.132 val_accuracy:0.100
epoch 103 loss: 2.136 mse:2.131 accuracy:0.100 val_loss:2.136 val_mse:2.130 val_accuracy:0.100
epoch 104 loss: 2.135 mse:2.129 accuracy:0.100 val_loss:2.134 val_mse:2.129 val_accuracy:0.100
epoch 105 loss: 2.133 mse:2.128 accuracy:0.100 val_loss:2.132 val_mse:2.127 val_accuracy:0.100
```

4. Hiperparámetros

- ❖ Con un lr pequeño $\sim 1e-5$ la loss no cambia.
- ❖ Con un lr grande $1e6$ la loss explota (NaN o inf).
En la práctica: Si $\text{loss} > 3 * \text{loss_epoch}$ 1 parar \rightarrow explota

```
In [48]: history = model.fit(x_train, y_train, 200, validation=[x_val, y_val], verbose=True)
/Users/ariel/Docencia/Balseiro/Materias/CV_DeepLearning/Practica/problemas_resueltos/p2_lib/
losses.py:50: RuntimeWarning: overflow encountered in multiply
    return 0.5 * self.delta * (W * W).sum()
epoch 001  loss: inf  mse:5.0  accuracy:0.100  val_loss:inf  val_mse:9.0  val_accuracy:0.100
epoch 002  loss: nan  mse:nan  accuracy:0.100  val_loss:nan  val_mse:nan  val_accuracy:0.100
epoch 003  loss: nan  mse:nan  accuracy:0.100  val_loss:nan  val_mse:nan  val_accuracy:0.100
epoch 004  loss: nan  mse:nan  accuracy:0.100  val_loss:nan  val_mse:nan  val_accuracy:0.100
```

En mi caso (problema 2.8) lr $\sim 1e3$ sigue siendo muy grande

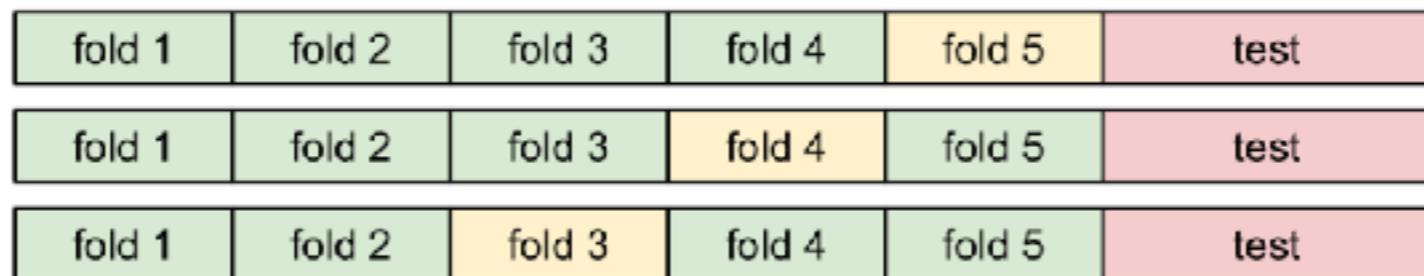
```
epoch 001  loss: 1.297  mse:1.000  accuracy:0.100  val_loss:1.297val_mse:1.000val_accuracy:0.100
epoch 002  loss: 1.002  mse:1.000  accuracy:0.100  val_loss:1.002  val_mse:1.000val_accuracy 0.100
epoch 003  loss: 2.347  mse:2.310  accuracy:0.099  val_loss:5.037  val_mse:5.000val_accuracy:0.100
epoch 004  loss: 4.048  mse:3.119  accuracy:0.101  val_loss:1.928  val_mse:1.000val_accuracy:0.122
epoch 005  loss: 1.602  mse:1.281  accuracy:0.101  val_loss:3.721  val_mse:3.400val_accuracy:0.092
epoch 006  loss: 8.232  mse:3.299  accuracy:0.098  val_loss:7.533  val_mse:2.600val_accuracy:0.100
```

No disminuye y oscila mucho

Un rango aceptable parece estar entre [1e-2 ... 1e1]

4. Hiperparámetros

- ❖ Estrategia de validación cruzada o Cross-Validation: Dividimos el conjunto de entrenamiento en validación y entrenamiento.



- ❖ Ajuste de grueso a fino o Coarse to Fine (se puede combinar con la validación cruzada):
 - ❖ Primer etapa: utilizamos pocas épocas para ver qué está pasando con los parámetros.
 - ❖ Segunda etapa: varias épocas en una búsqueda más refinada

4. Hiperparámetros

- ❖ Ejemplo: 10 épocas, bs=100, 30% de los datos de entrenamiento se utilizan para validación

```
9 epochs = 10
10 itmax = 50
11 for it in range(itmax):
12
13     reg_factor = 10**np.random.uniform(-6,0)
14     lr = 10**np.random.uniform(-3,1) #--> Mejor en un espacio log.
15     seed = None #--> No repetitibilidad
16     inputs = Input(input_dim)
17     l1 = Dense(100, act_func='sigmoid', seed=seed)(inputs)
18     l2 = Dense(100, act_func='sigmoid', seed=seed)(l1)
19     outputs = Dense(10, act_func='sigmoid', seed=seed)(l2)
20
21     loss = MSE()
22     metrics = [mse, accuracy]
23     optimization = SGD(bs=bs, lr=lr)
24     reg = L2(reg_factor)
25     model = Network(inputs, outputs, loss, metrics, optimization, reg=reg)
26     history = model.fit(x_train, y_train, epochs, validation=[x_val, y_val],
27                           verbose=False)
28     print('lr: {:.6f}\t reg: {:.6f}\t val_acc: {:.6f}\t ({}/{}) '.format(lr,
29         reg_factor, history['val_accuracy'][-1]), it, itmax)
```

4. Hiperparámetros

- ❖ Ejemplo (problema 2.8): 10 épocas, bs=100, 30% de los datos de entrenamiento se utilizan para validación

```
lr: 0.011785    reg: 0.025376    val_acc: 0.274533 (1/50)
lr: 0.346997    reg: 0.001071    val_acc: 0.344133 (2/50)
lr: 7.300293    reg: 0.000490    val_acc: 0.262533 (3/50)
lr: 8.848671    reg: 0.000001    val_acc: 0.101400 (4/50)
lr: 0.655791    reg: 0.469060    val_acc: 0.098333 (5/50)
lr: 0.158438    reg: 0.000007    val_acc: 0.359400 (6/50)
lr: 4.470351    reg: 0.857240    val_acc: 0.102533 (7/50)
lr: 0.036313    reg: 0.000012    val_acc: 0.361200 (8/50)
lr: 0.145830    reg: 0.000277    val_acc: 0.361933 (9/50)
lr: 0.011910    reg: 0.017476    val_acc: 0.299467 (10/50)
lr: 0.002054    reg: 0.040106    val_acc: 0.247600 (11/50)
lr: 0.004204    reg: 0.000003    val_acc: 0.281533 (12/50)
lr: 5.542544    reg: 0.012275    val_acc: 0.101400 (13/50)
lr: 0.753310    reg: 0.000035    val_acc: 0.347733 (14/50)
lr: 0.020305    reg: 0.000025    val_acc: 0.341800 (15/50)
lr: 0.020305    reg: 0.000025    val_acc: 0.341800 (15/50)
lr: 0.047148    reg: 0.000003    val_acc: 0.372933 (16/50)
lr: 0.715592    reg: 0.000038    val_acc: 0.344800 (17/50)
lr: 3.136087    reg: 0.000027    val_acc: 0.308200 (18/50)
lr: 4.079735    reg: 0.000005    val_acc: 0.101400 (19/50)
lr: 0.058329    reg: 0.005209    val_acc: 0.346600 (20/50)
```

4. Hiperparámetros

```
13     reg_factor = 10**np.random.uniform(-6,0)
14     lr = 10**np.random.uniform(-3,1)
```

```
13     reg_factor = 10**np.random.uniform(-6,-3)
14     lr = 10**np.random.uniform(-2,-1)
```

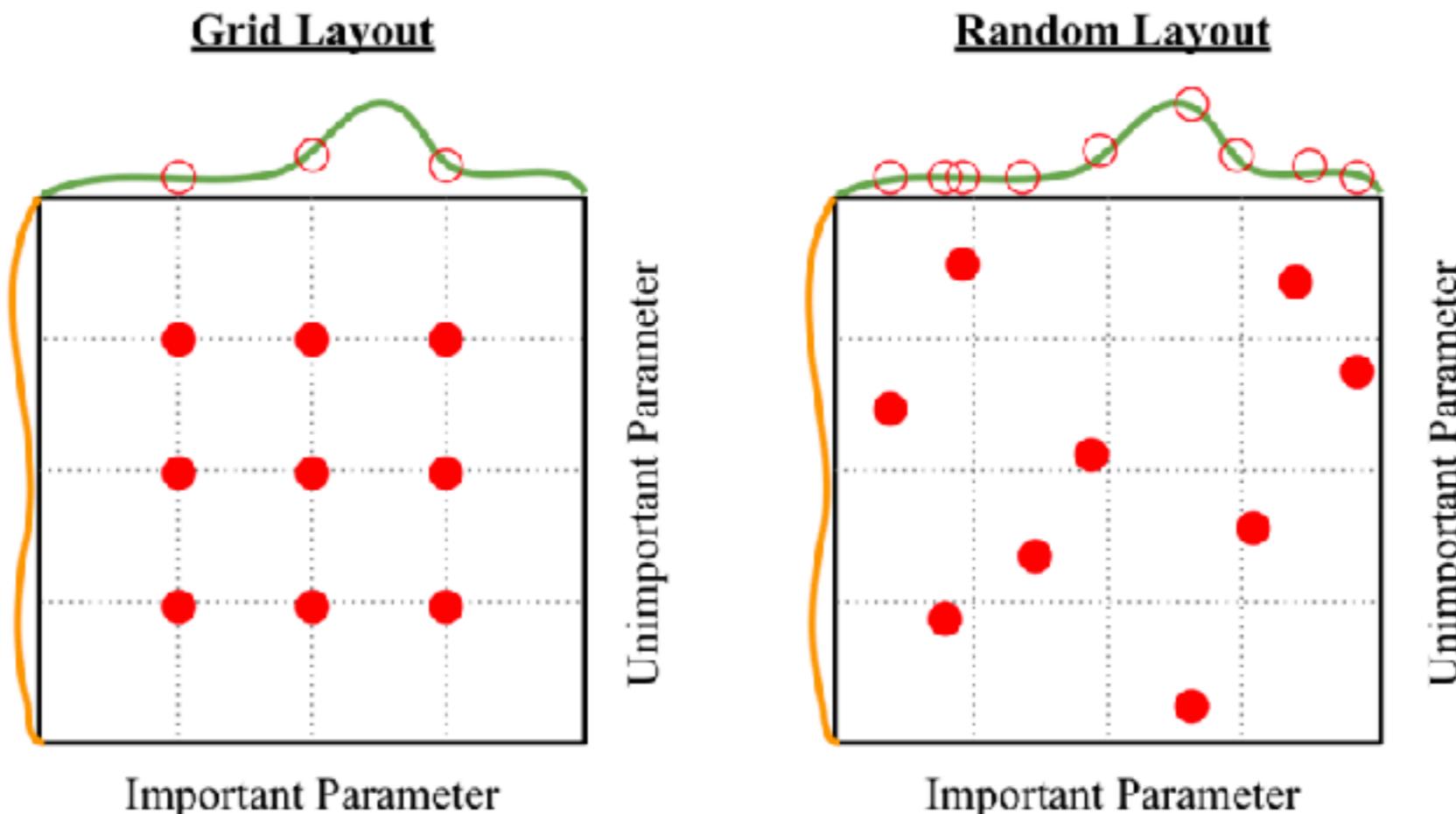
lr: 0.074153	reg: 0.000009	val_acc: 0.372267 (1/50)
lr: 0.056680	reg: 0.000086	val_acc: 0.374467 (2/50)
lr: 0.012457	reg: 0.000156	val_acc: 0.323267 (3/50)
lr: 0.013878	reg: 0.000002	val_acc: 0.326867 (4/50)
lr: 0.022867	reg: 0.000002	val_acc: 0.350000 (5/50)
lr: 0.012395	reg: 0.000296	val_acc: 0.327400 (6/50)
lr: 0.045546	reg: 0.000033	val_acc: 0.364867 (7/50)
lr: 0.084343	reg: 0.000420	val_acc: 0.369133 (8/50)
lr: 0.010996	reg: 0.000121	val_acc: 0.318600 (9/50)
lr: 0.042293	reg: 0.000829	val_acc: 0.366200 (10/50)
lr: 0.068953	reg: 0.000082	val_acc: 0.368800 (11/50)
lr: 0.057531	reg: 0.000129	val_acc: 0.366400 (12/50)
lr: 0.029347	reg: 0.000171	val_acc: 0.359667 (13/50)
lr: 0.051847	reg: 0.000118	val_acc: 0.367733 (14/50)
lr: 0.045753	reg: 0.000011	val_acc: 0.365333 (15/50)
lr: 0.017673	reg: 0.000009	val_acc: 0.334000 (16/50)
lr: 0.034508	reg: 0.000163	val_acc: 0.362800 (17/50)
lr: 0.046635	reg: 0.000307	val_acc: 0.368400 (18/50)
lr: 0.010515	reg: 0.000038	val_acc: 0.317267 (19/50)
lr: 0.084896	reg: 0.000303	val_acc: 0.373800 (20/50)
lr: 0.058557	reg: 0.000021	val_acc: 0.367933 (21/50)
lr: 0.026351	reg: 0.000002	val_acc: 0.357400 (22/50)
lr: 0.012540	reg: 0.000398	val_acc: 0.325867 (23/50)

Muchas más precisión no parece que logremos. Igual no está mal para una red simple (arquitectura, inicialización y optimizador SGD simples)

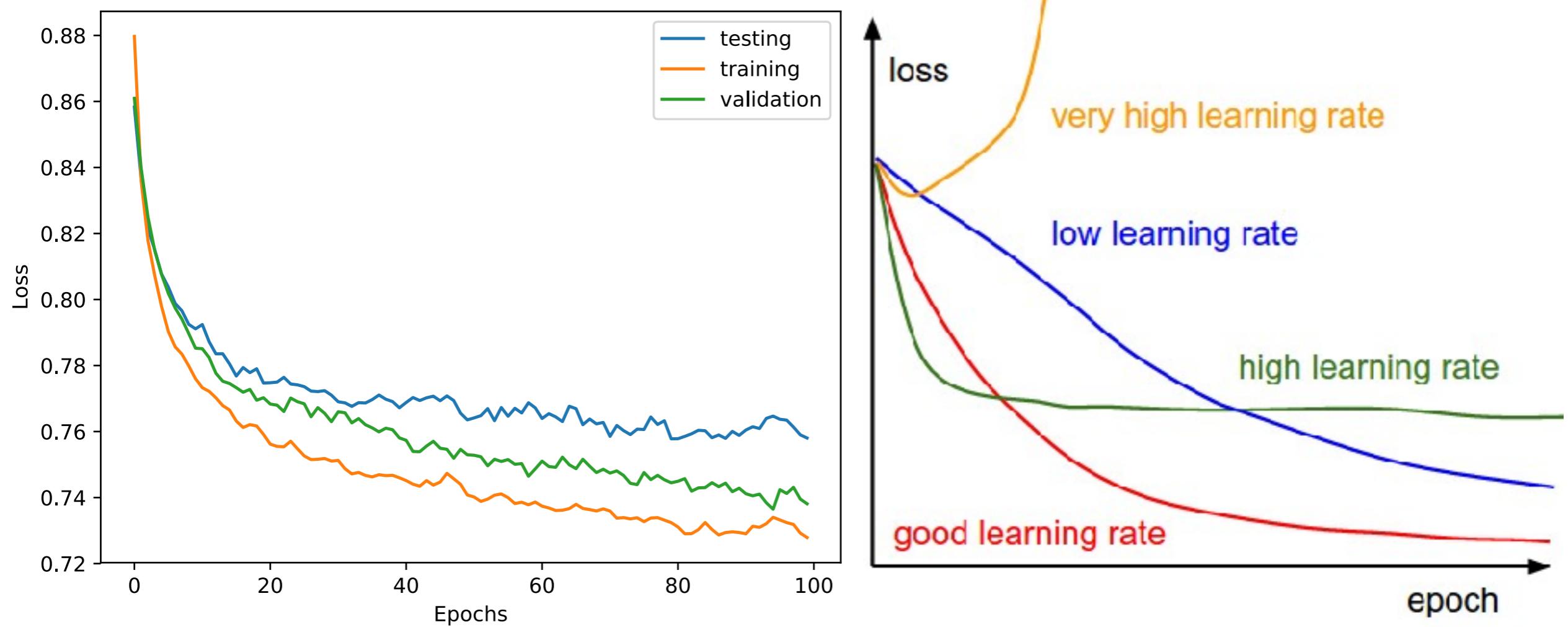
Si el mejor resultado se encuentra sobre los límites de la búsqueda, entonces extender.

4. Hiperparámetros

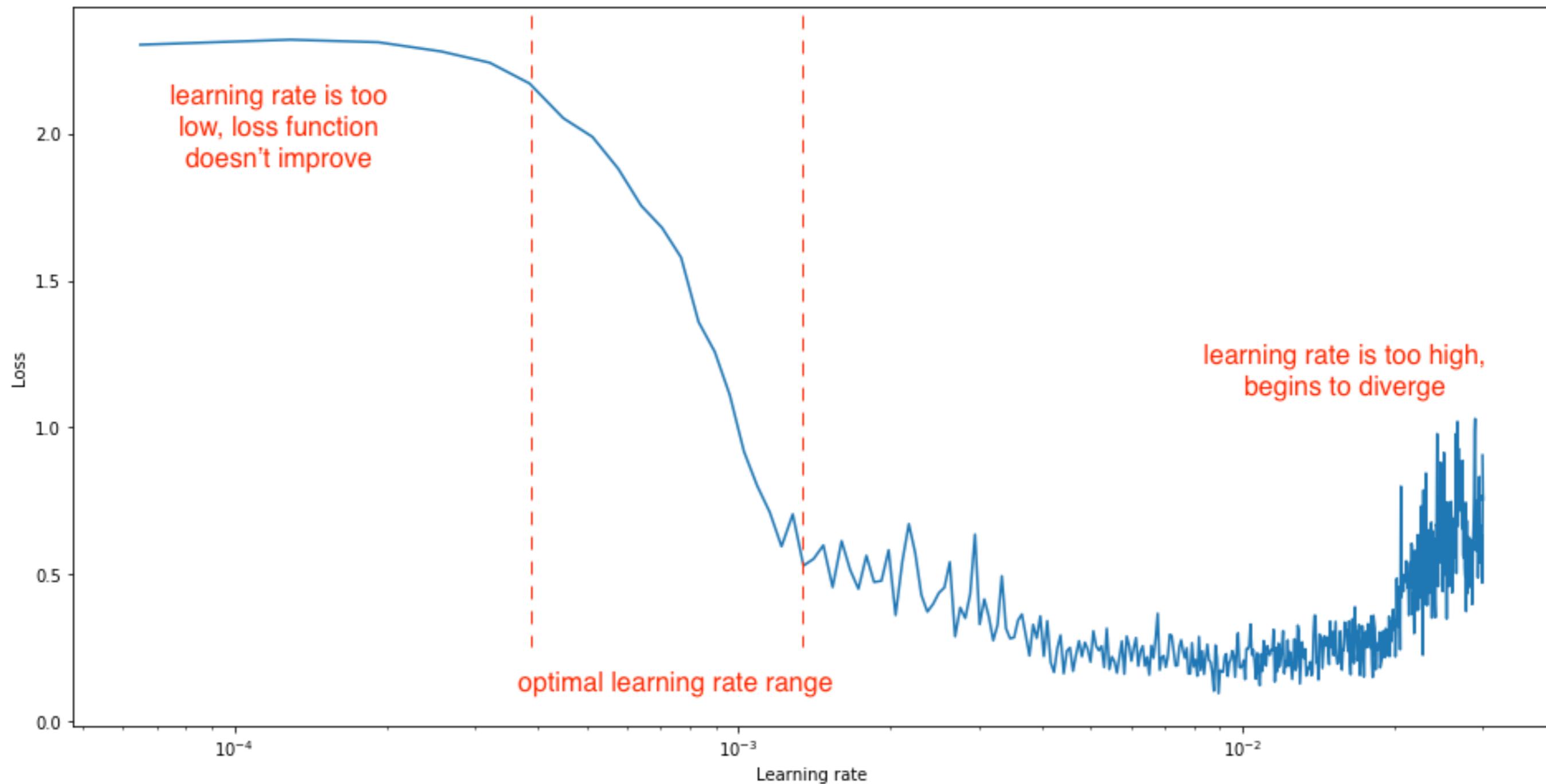
- ❖ La estrategia búsqueda aleatoria o Random Search es mejor en caso de tener algunos hiperparámetros más importantes que otros [Bergstra, J. and Bengio, Y., 2012]



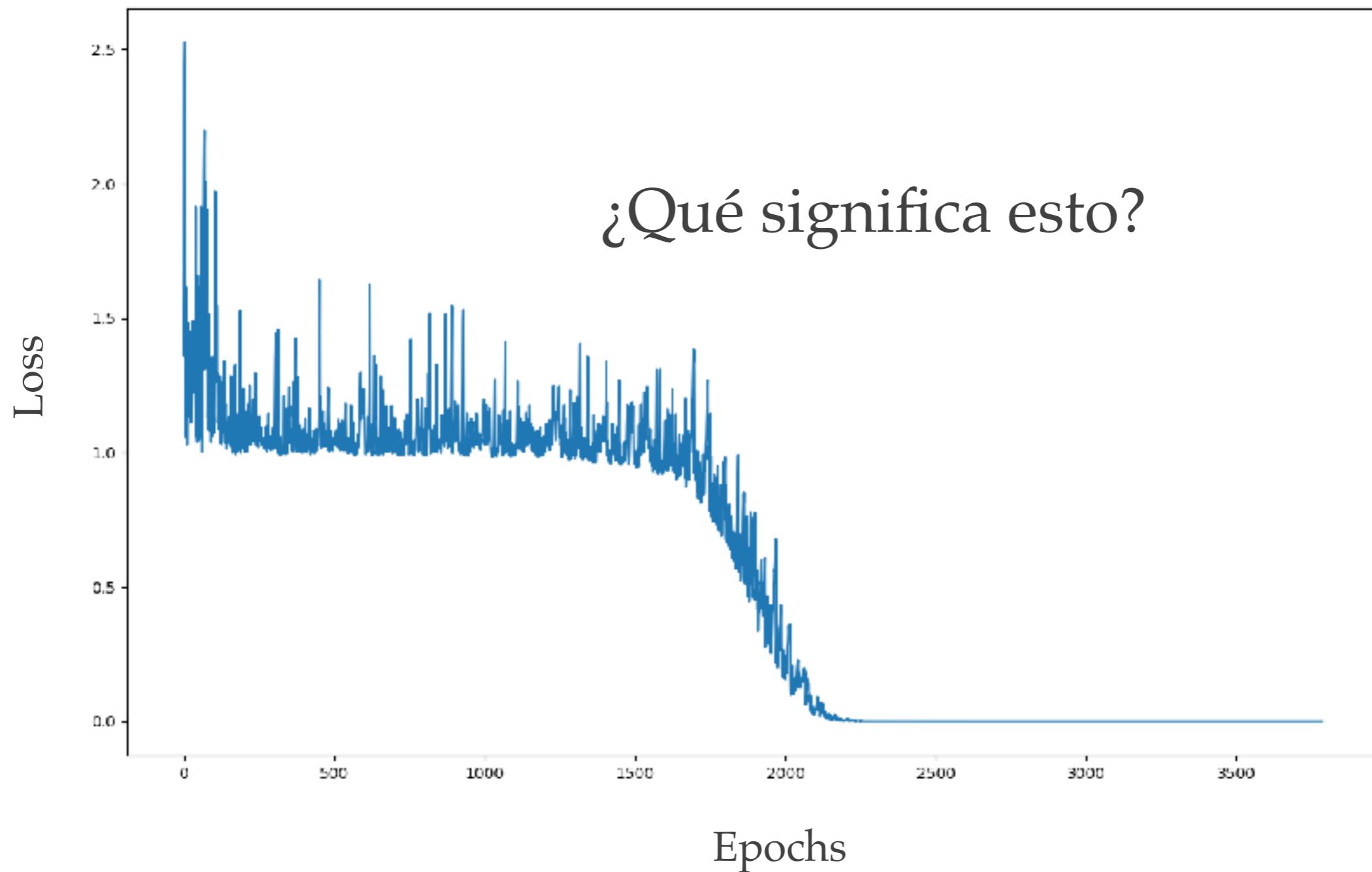
5. Monitoreo



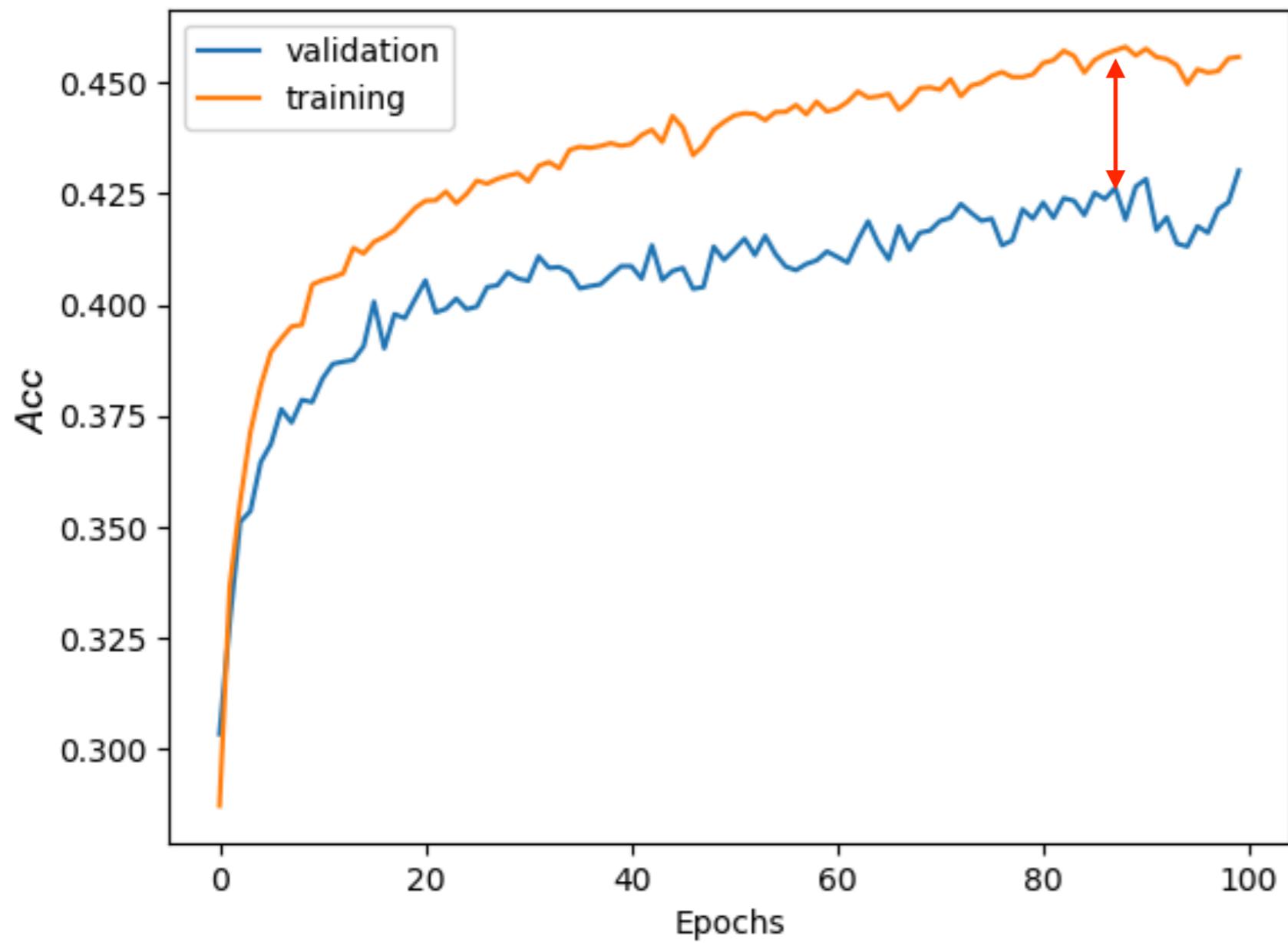
5. Monitoreo



5. Monitoreo



5. Monitoreo



Mayor = overfitting

Pequeño = Mejor
generalización

5. Monitoreo

- ❖ Es útil ir mirando la evolución de los pesos normalizados acorde a su escala: No se quieren que sean muy grandes (cambia mucho) ni muy chicos (no aprende). Valores razonables pueden ser ~ 0.01

```
9 scale = np.linalg.norm(w.ravel())
10 print('W ratio:{:.6f}\t({}/{})'.format(np.linalg.norm(w_update)/scale, it, itmax))
```