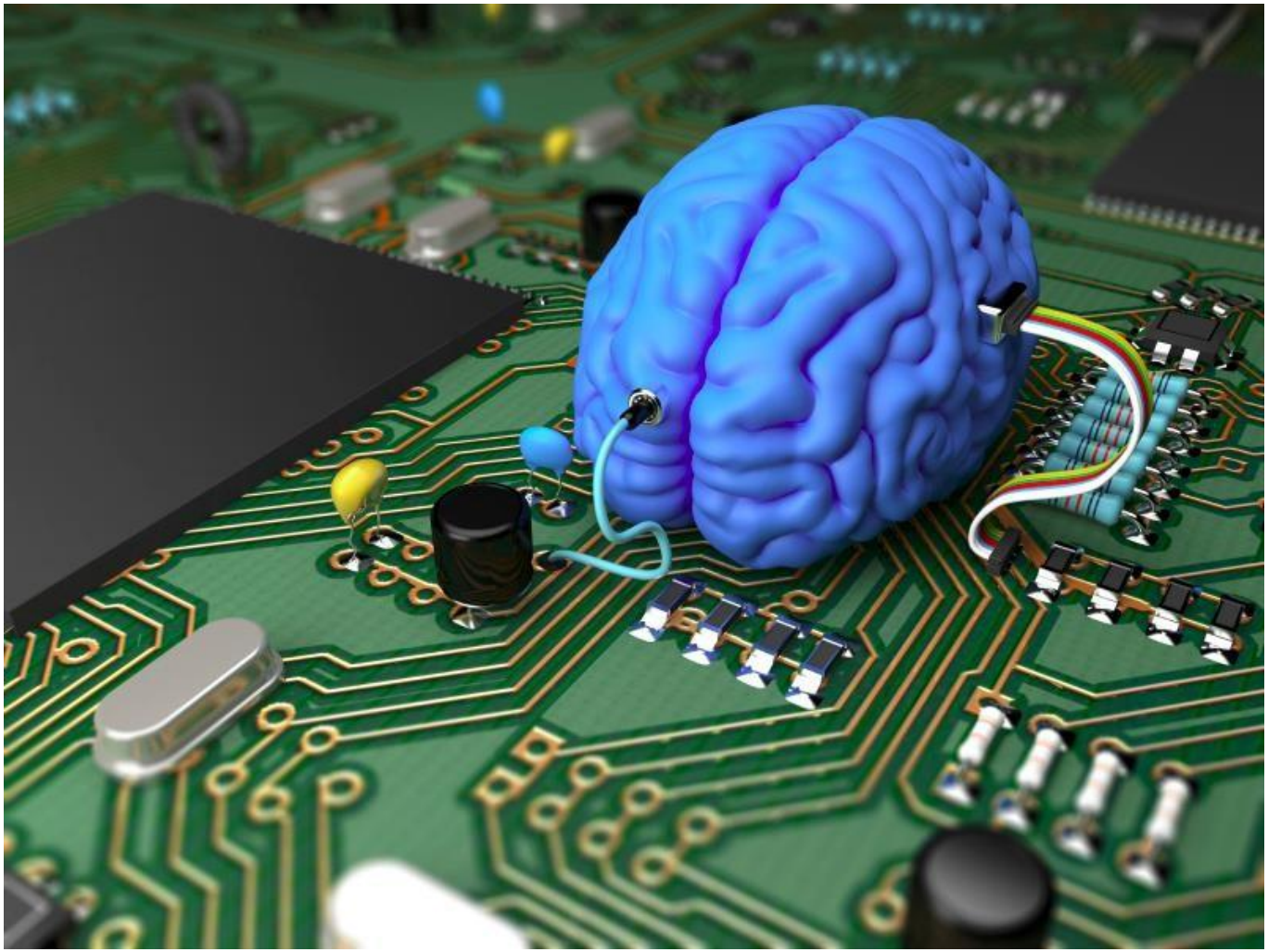
Machine learning I : Deep Learning



Master Universitario Oficial **Data Science**
con el apoyo del
UC UNIVERSIDAD DE CANTABRIA · UIMP Universidad Internacional Menéndez Pelayo · CSIC

Lara Lloret Iglesias

Santander, Febrero 2025

- Acordaos de rellenar la información de vuestras prácticas/TFM en el Google doc:

https://docs.google.com/spreadsheets/d/1FZzPHAqqoT3xYQ6R5MjEoyiMbAIQsx-h5vpnFsW5Lgw/edit?usp=sharing

Prácticas

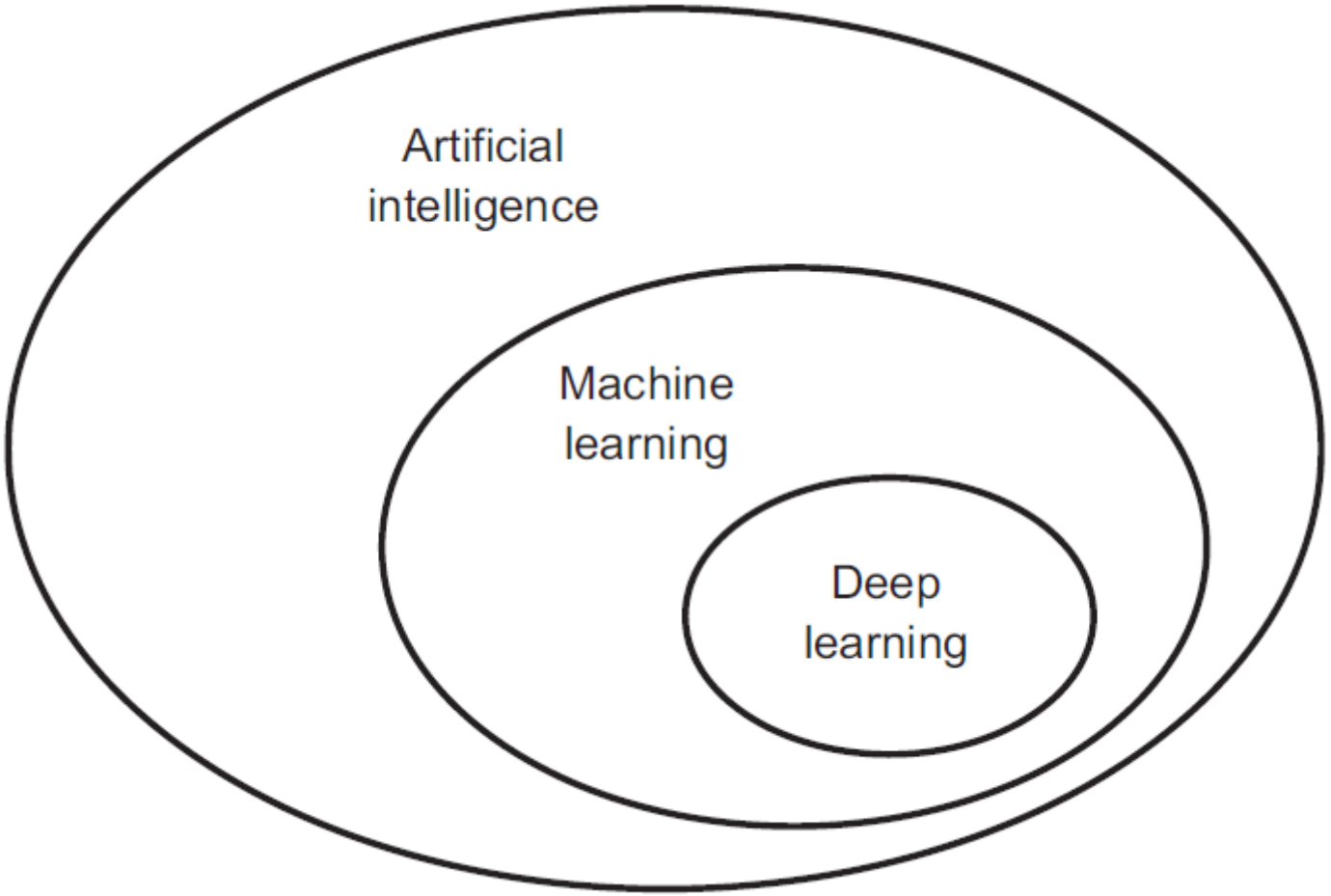
Clasificación de números (MNIST) con redes CNN

(Opcional) Generación de texto con redes RNN

- Subid las tareas a Moodle → **Subid el Notebook ejecutado** donde se vean los resultados que salen al correr.

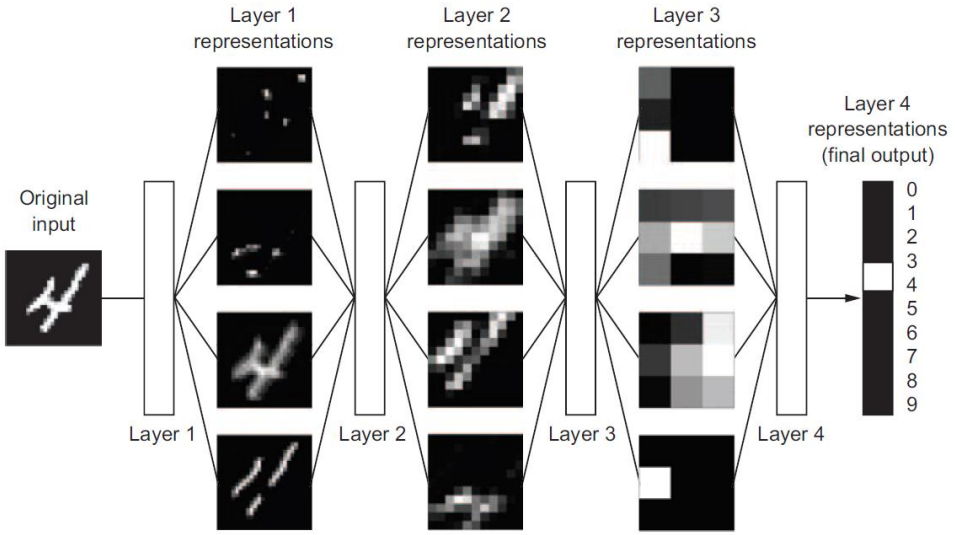
- Podéis usar datasciencehub, vuestro equipo o Google Colab.
- Podéis entregarlas hasta el 30 de abril a las 23:59.

Introducción



¿Qué significa "profundo"?

- Deep Learning es un enfoque diferente del aprendizaje de nuevas representaciones.
- Lo "profundo" no hace referencia a un entendimiento más profundo de los datos sino a aprender **capas sucesivas de representaciones cada vez más significativas**.
- El número de estas capas es lo que se conoce como la **profundidad del modelo**.



¿Por qué es diferente el Deep Learning?

- El Deep Learning ofrece un mayor rendimiento en general.
- Con Deep Learning la resolución de problemas resulta mucho más sencilla porque **automatiza** lo que antes era el problema crucial en machine-learning: **la ingeniería de características** (feature engineering).
- **Las técnicas anteriores** de Machine Learning transformaban los **datos solo en una o dos representaciones diferentes**. Normalmente a través de transformaciones simples como proyecciones no lineales (SVMs) o Árboles de Decisión.
- Esto implicaba que para sacar el máximo potencial de un conjunto de datos hubiera que **extraer previamente a mano representaciones/características** de los datos que ayudaran a clasificar.
- Con Deep Learning se aprenden las características "al vuelo" sin necesidad de hacer transformaciones previas a mano
- Además estas características son aprendidas al mismo tiempo: **todos los parámetros son actualizados a la vez**.

¿Por qué ahora?

Las ideas principales de Deep Learning ya se entendían en 1989.

Entonces…**¿por qué ahora?**

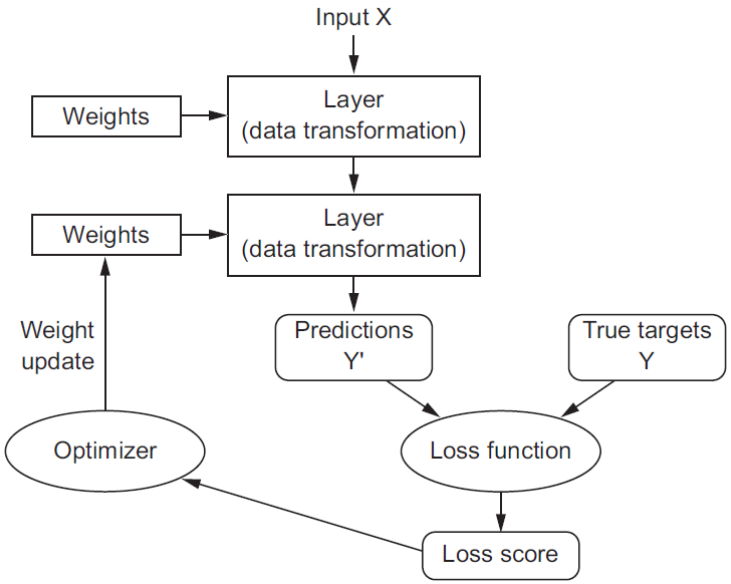
Los motivos principales son:

- **Hardware:**
  - Entre 1990 y 2010 las CPUs se hicieron aproximadamente 5000 veces más rápidas.
  - Desarrollo de GPUs: una GPU puede sustituir a clusters masivos de CPUs en tareas altamente paralelizables (i.e multiplicación de matrices).
- **Datasets y benchmarks:**
  - Además de las enormes mejoras en almacenamiento de datos, **el auténtico cambio vino de la mano de internet**, haciendo posible recolectar y distribuir enormes dataset de una manera sencilla y eficiente.
- **Avances algorítmicos*:**
  - Mejores funciones de activación
  - Mejores esquemas de inicialización de los pesos
  - Mejores esquemas de optimización, como **RMSProp** y **Adam**

***** Estos avances no fueron posibles hasta que no se produjeron los avances en el hardware y en los datasets accesibles.

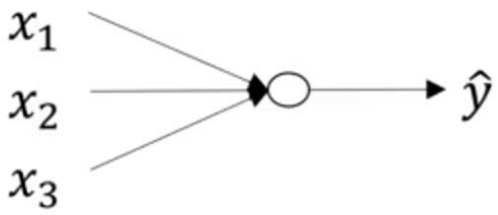
Redes Neuronales

- En Deep Learning estas "representaciones en capas" casi siempre suelen aprenderse a través de unos modelos llamados **Redes Neuronales** (ya vistas en la primera parte de esta asignatura)

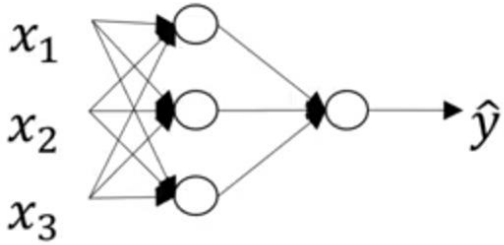


- El estado del arte en visión artificial incluye un modelo de Redes Neuronales llamadas **Redes Convolucionales**

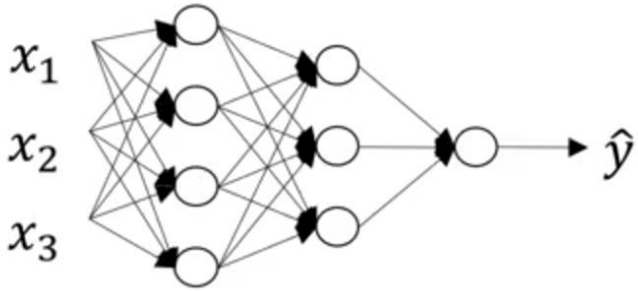
Redes Neuronales Deep



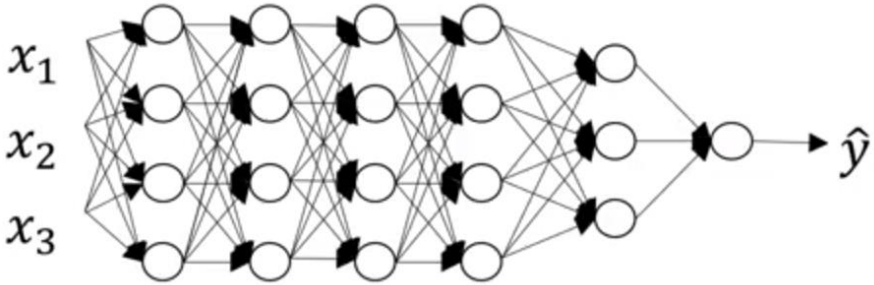
logistic regression



1 hidden layer

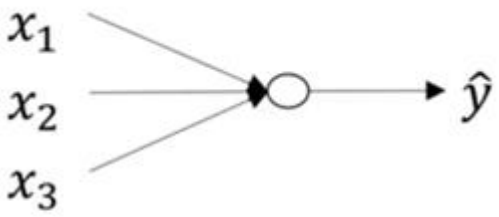


2 hidden layers

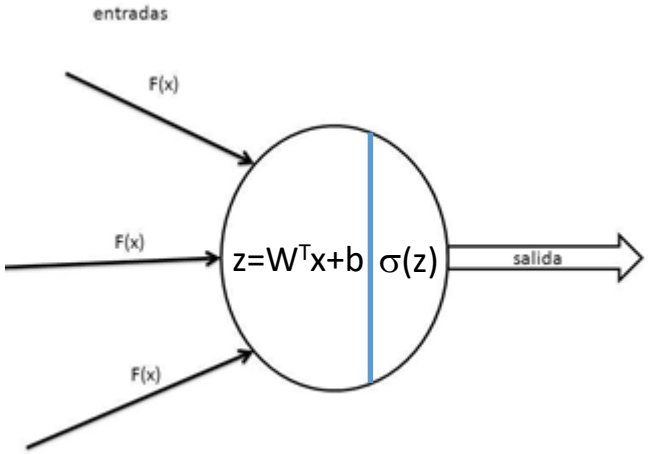


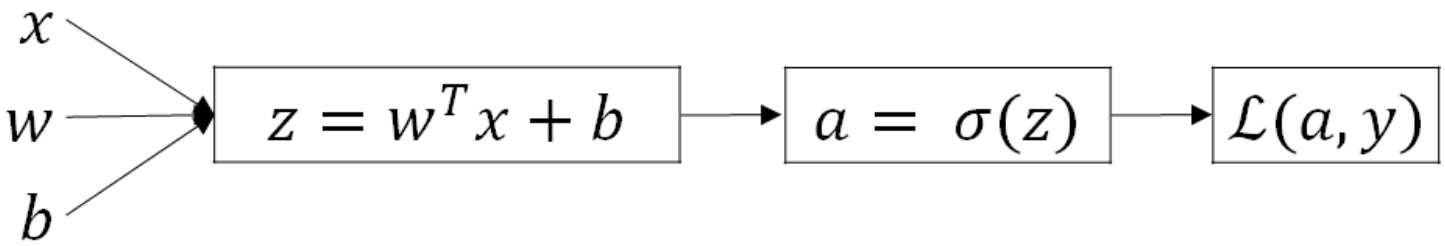
5 hidden layers

Logistic regression: Gradient descent



logistic regression





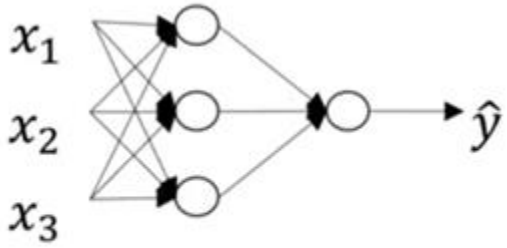
$$\frac{dL}{dw} = dw = \frac{dL}{da}\frac{da}{dz}\frac{dz}{dw}$$

$$\frac{dL}{db} = db = \frac{dL}{da}\frac{da}{dz}\frac{dz}{db}$$

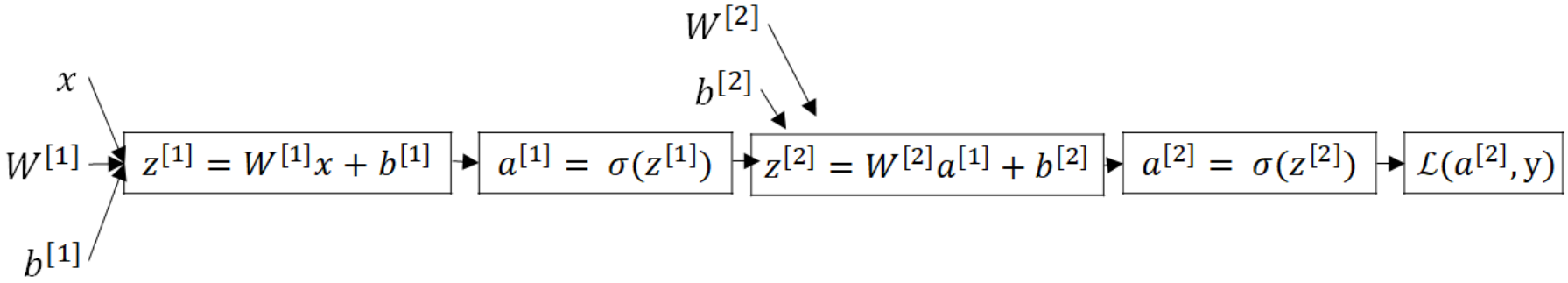
$$W = W - \alpha\, dW$$

$$b = b - \alpha\, db$$

Logistic regression: Gradient descent



1 hidden layer

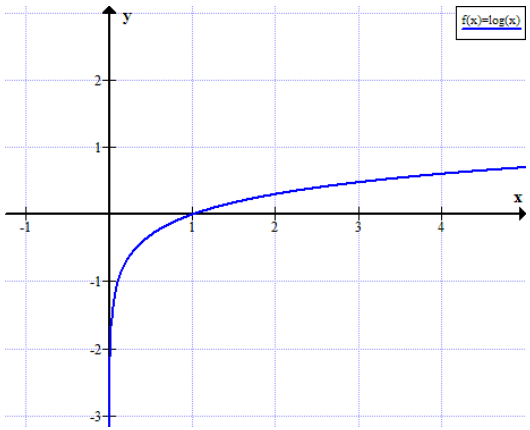


- ¿Cuáles son las dimensiones de $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$?
  (h, n) (h, 1) (1, h) (1, 1)

Logistic regression: función de pérdida

$$L(y, \hat{y}) = - (y \log \hat{y} + (1- y) \log (1- \hat{y}))$$

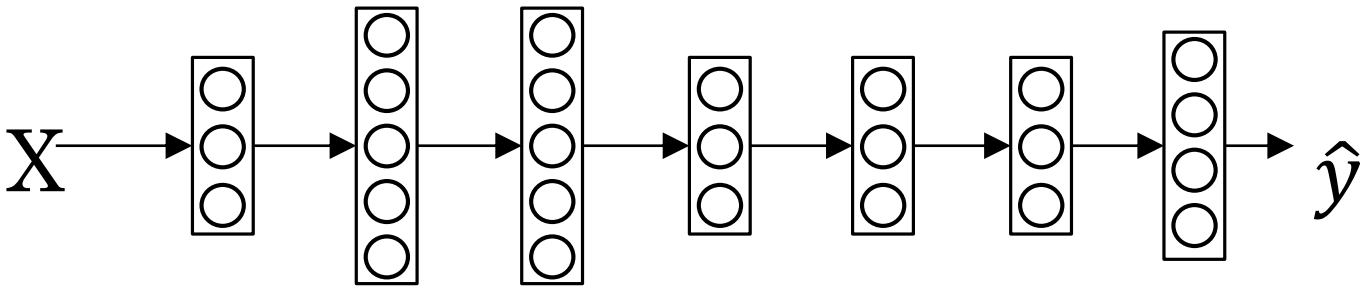
- Si y=1 $L(y, \hat{y}) = - \log \hat{y}$: minimizar $L(y, \hat{y})$ → maximizar $\hat{y}$
- Si y=0 $L(y, \hat{y}) = - \log (1 - \hat{y})$: minimizar $L(y, \hat{y})$ → minimizar $\hat{y}$



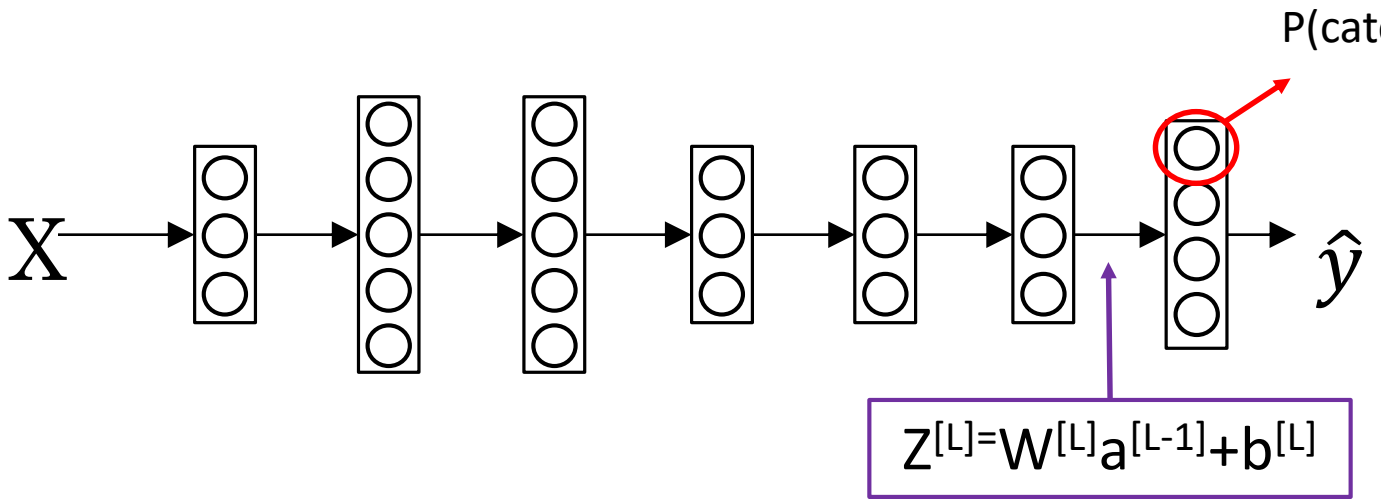
Clasificación multiple: Softmax regression

- Generalización de *logistic regression*
- En lugar de clasificar en dos categorías sirve para clasificar en varias

- La *output layer* tiene el mismo número de unidades que categorías queremos clasificar



Clasificación multiple: Softmax regression



Softmax activation function: $g_i(z^{[L]}) = \frac{e^{z_i^{[L]}}}{\sum_j e^{z_j^{[L]}}}$

Peculiaridad:
Toma como input todo el vector $z^{[L]}$

Ejemplo: $z^{[L]} = \begin{matrix}3\\-2\\6\end{matrix}$

$g_0(z^{[L]}) = \frac{e^3}{e^3+e^{-2}+e^6} = 0.047$

$g_1(z^{[L]}) = \frac{e^{-2}}{e^3+e^{-2}+e^6} = 0.0003$

$g_2(z^{[L]}) = \frac{e^6}{e^3+e^{-2}+e^6} = 0.952$

$a^{[L]} = \begin{matrix}0.047\\0.0003\\0.952\end{matrix}$ Softmax

$a^{[L]} = \begin{matrix}0\\0\\1\end{matrix}$ Hardmax

Clasificación Multiple: Loss function

- En el caso de *logistic regression* usábamos como función de pérdida:

$$L(y, \hat{y}) = - (y \log \hat{y} + (1- y) \log (1- \hat{y}))$$

- Para la clasificación múltiple usamos la versión más general:

$$L(y, \hat{y}) = - \sum_j y_j \log \hat{y}_j$$ ← Cross-Entropy

Ejemplo: $y = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ = gato $\quad \hat{y} = \begin{pmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{pmatrix}$ $\quad L(y, \hat{y}) = -\log \hat{y}_2$

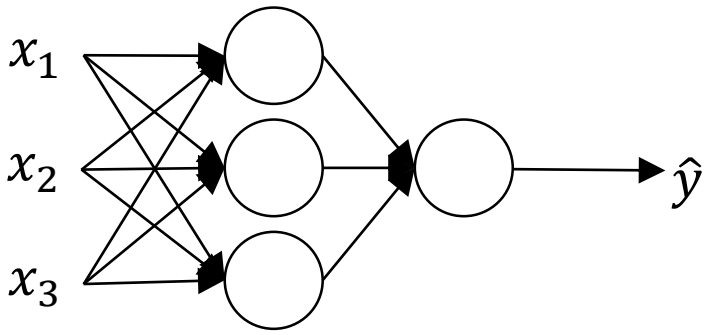
Si queremos que $L(y, \hat{y})$ sea lo más pequeño posible, significa que $\log \hat{y}_2$ (y por tanto $\hat{y}_2$) **tiene que ser lo mayor posible**

Minimizar $L(y, \hat{y})$ → maximizar la probabilidad de la categoría correcta

$$Cost = \frac{1}{m} \sum_{i=1}^{m} L(y_i, \hat{y}_i)$$

Gato

¿Por qué necesitamos las funciones no lineales (funciones de activación)?



Dado un cierto x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]}) = z^{[1]}$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]}) = z^{[2]}$$

$$a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

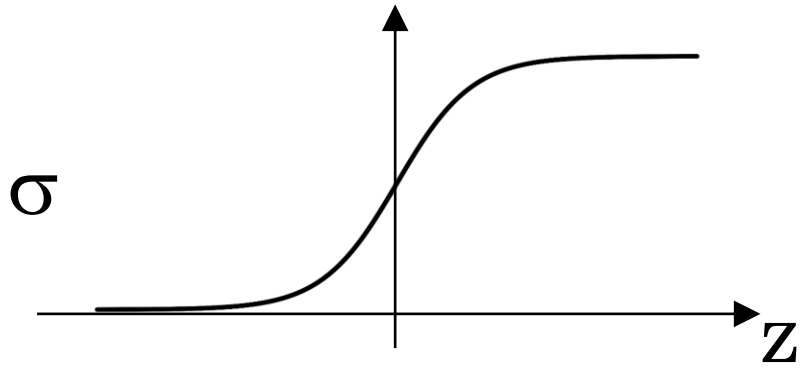
$$a^{[2]} = W^{[2]}\left(\underbrace{W^{[1]}x + b^{[1]}}_{a^{[1]}}\right) + b^{[2]}$$

$$= \underbrace{\left(W^{[2]}W^{[1]}\right)}_{W'}x + \underbrace{\left(W^{[2]}b^{[1]} + b^{[2]}\right)}_{b'}$$

$$= W'x + b'$$

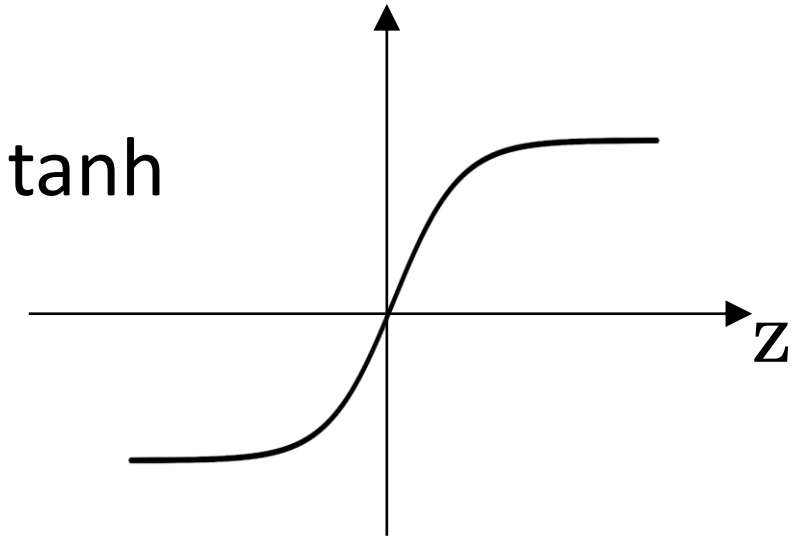
¡Sería equivalente a tener un único nodo!

¿Qué función de activación elegir?



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

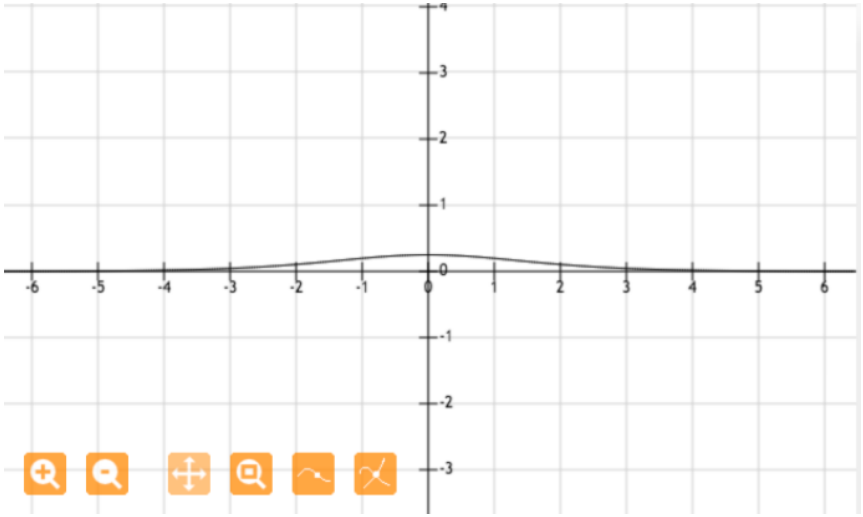
$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$$

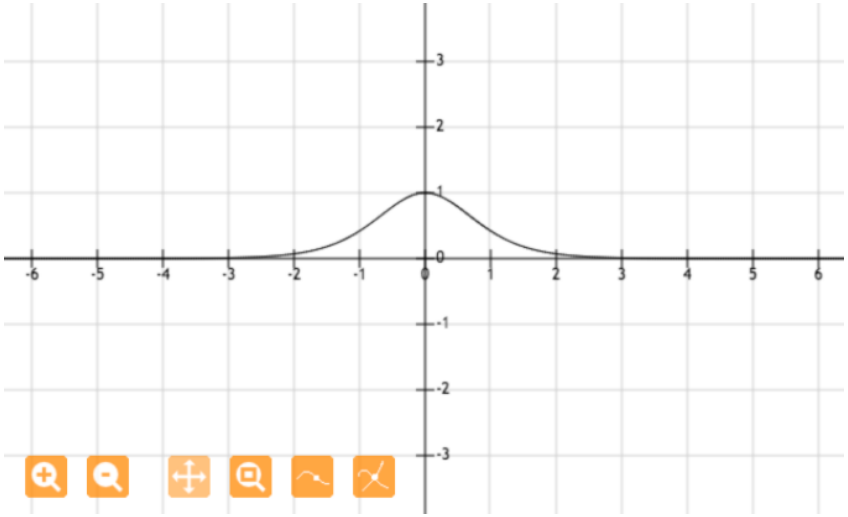


$$g(z) = \tanh(z)$$

$$g'(z) = 1 - \left(\tanh(z)\right)^2$$

¿Qué función de activación elegir?





$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z))$$

$$g'(z) = 1 - \left(\tanh(z)\right)^2$$

¿Qué función de activación elegir?

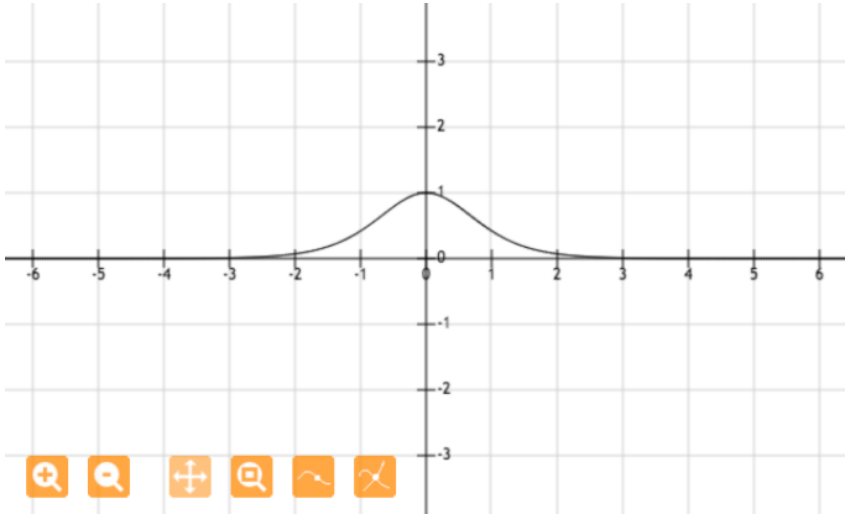
El valor de la derivada es mayor para g(z)= tanh(z) → el aprendizaje es más rápido

En el caso de la sigmoide, el valor de la derivada es como máximo 0.25

Cuando tienes muchas capas, tendrás que multiplicar muchas veces números muy pequeños, dándote un gradiente tan pequeño que prácticamente la red no aprenderá nada (vanishing gradient).

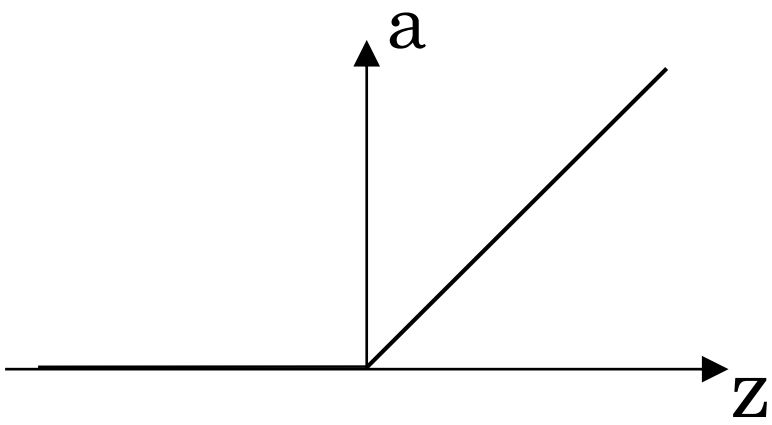
No se puede hacer Deep Learning teniendo sigmoides en las capas ocultas.

tanh(z) funciona mejor (máximo = 1) pero desciende muy rápido.



$$g'(z) = 1 - \big(\tanh(z)\big)^2$$

Nuevas funciones de activación



ReLU

$$f(x) = \max(0, x)$$

Ventajas:

Evitamos el problema del desvanecimiento del gradiente, ya que la derivada es 1 para x > 0.

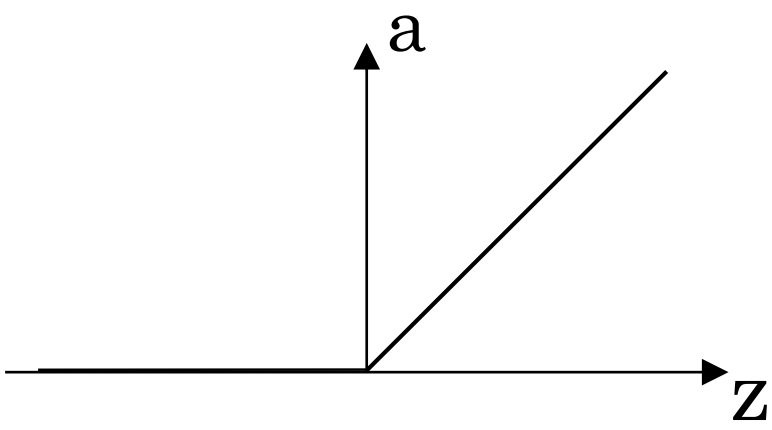
ReLU tarda menos en aprender y es menos costoso computacionalmente que tanh y sigmoide.

Dado que su salida es 0 siempre que su entrada es negativa, **se activan menos neuronas, lo que da lugar a una red dispersa y, por tanto, a una mayor eficiencia computacional.**

ReLU implica operaciones matemáticas más sencillas que tanh y sigmoide, lo que aumenta aún más su rendimiento computacional.

https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24

Nuevas funciones de activación



ReLU

$$f(x) = \max(0, x)$$

(Posibles) Desventajas:

No derivable → A un nivel práctico no importa, nunca va a ser exactamente 0.

Desde el punto de vista computación, si alguna vez es 0, **el algoritmo va a elegir entre 0 o 1** como derivada y seguirá con el backpropagation.

(Reales pero poco frecuentes) Desventajas:

The dying ReLU problem → El problema de la dying ReLU se refiere al escenario donde **muchas neuronas ReLU toman el valor de 0** como salida.

En ese caso los gradientes dejan de propagarse y los pesos no se actualizan, por lo que una gran parte de la red puede dejar de aprender.

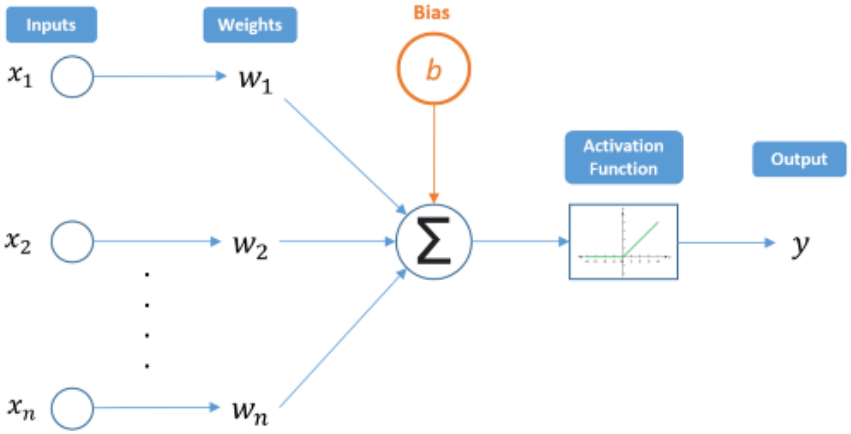
The dying ReLU problem

- Learning rate demasiado alto

$$W_{ij}^{*} = W_{ij} - a\left(\frac{\partial E}{\partial W_{ij}}\right)$$

New Weight | Old Weight | Learning Rate | Derivative of Error with respect to Weight

- Bias muy negativo

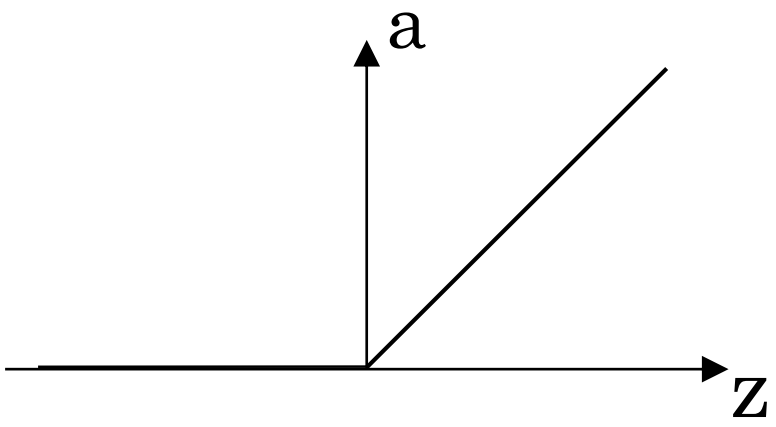


Si el learning rate (**α**) es demasiado alto, hay una probabilidad significativa de que los nuevos pesos acaben teniendo un valor muy negativo también.

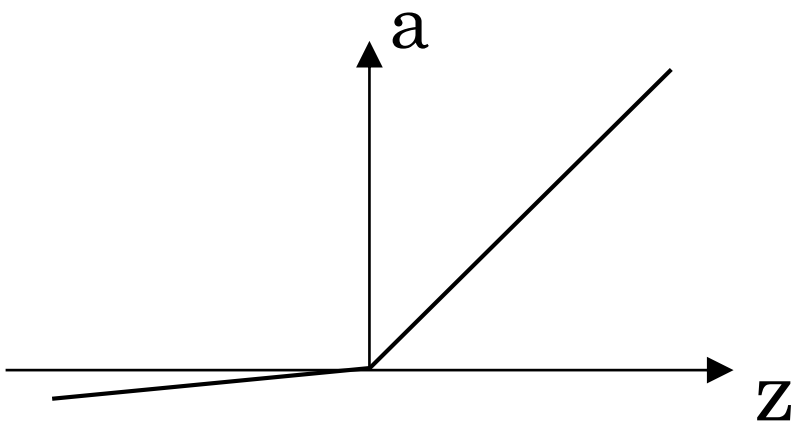
Un bias grande y negative puede hacer que los inputs de la ReLU sean negativos.

Soluciones: bajar la learning rate, mejor inicialización de los pesos (pronto), y variaciones de ReLU

Nuevas funciones de activación



ReLU



Leaky ReLU

$$f(x) = \max(0, x)$$

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Recetario para las funciones de activación

Cada problema es un mundo, pero podemos establecer un recetario general para las funciones de activación y hacer pruebas usando esta configuración como base:

- Usa ReLU (o leaky ReLU) para todas las capas escondidas
- Usa tangh o sigmoide solo en la última capa cuando estés en un problema de clasificación binaria
- Usa softmax en la última capa cuando estés en un problema de clasificación multiple

Resumiendo

Hasta aquí hemos visto:

- Como funciona la red neuronal más básica: logistic regression (arquitectura y *loss function*)
- Como funciona una red neuronal para clasificación múltiple (arquitectura y *loss function*)
- Para que necesitamos funciones no lineales en las redes neuronales y cuales son las más populares

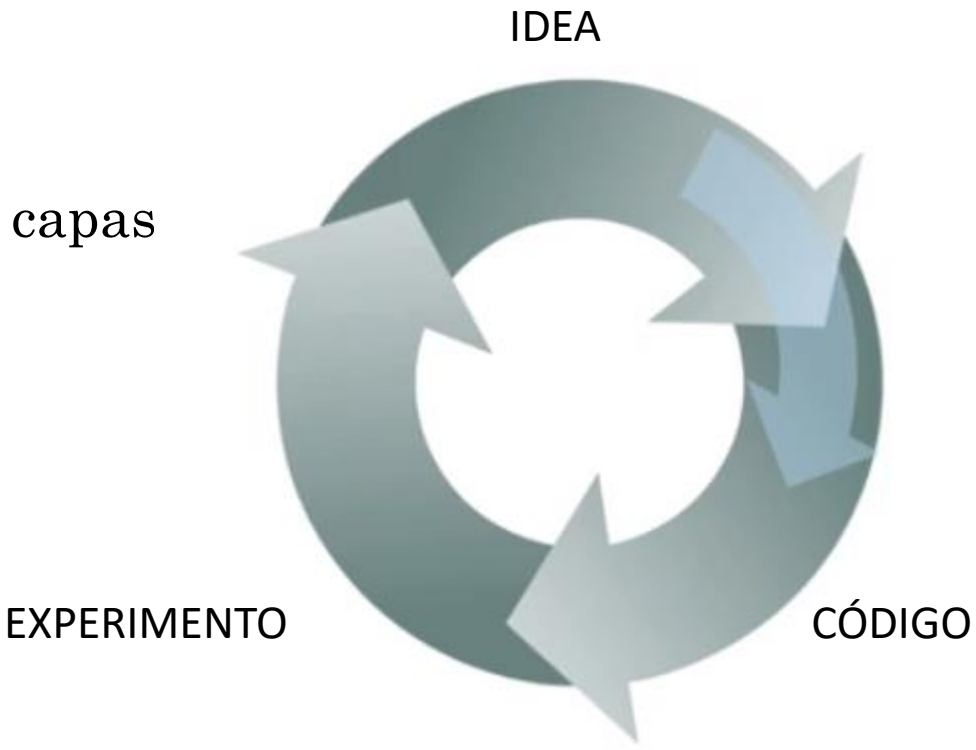
Deep Learning práctico: Dataset, problemas y optimización

Deep Learning: proceso iterativo

Hiperparámetros

- Número de capas
- Número de unidades en cada capas
- Learning rate
- Funciones de activación

…



Datasets

- Entrenar usando el **Training Set** usando diferentes modelos
- Probarlos en **Development Set** y elegir el modelo que dé mejores resultados
- Una vez elegido este modelo probarlo en el **Test Set** para tener una estimación sin sesgo de cómo de bien funciona

Antes de Big Data: 70% (train) / 30% (test) ~10.000
60% (train) / 20% (dev) / 20% (test)

Big Data: 98% (train) / 1% (dev) / 1% (test) ~1.000.000

10000!

Datasets

Problema típico : Diferentes tipos de datasets

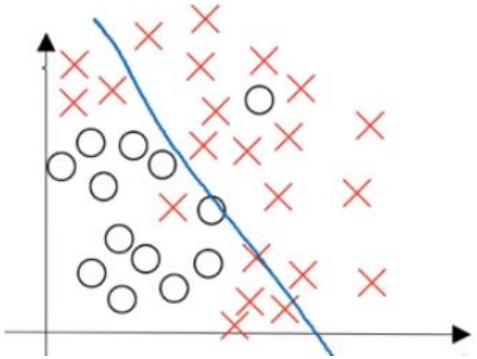
Entrenas tu red con
fotos de la web (alta resolución)

Dev/Test datasets
con imágenes de móvil

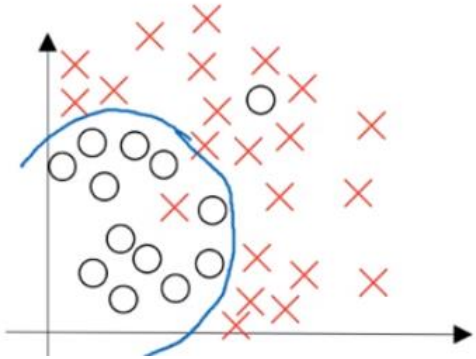
No hay ningún problema mientras Dev y Test tengan el mismo tipo de imágenes

Varianza/Sesgo

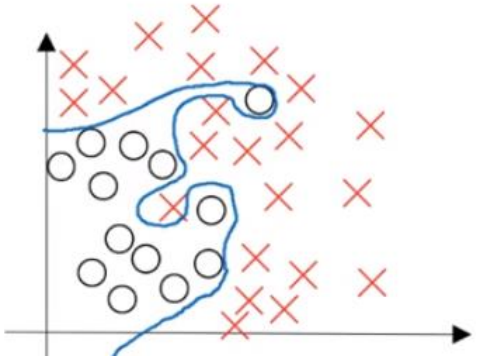
REPASO



Sesgo alto
(underfitting)



"Bien"



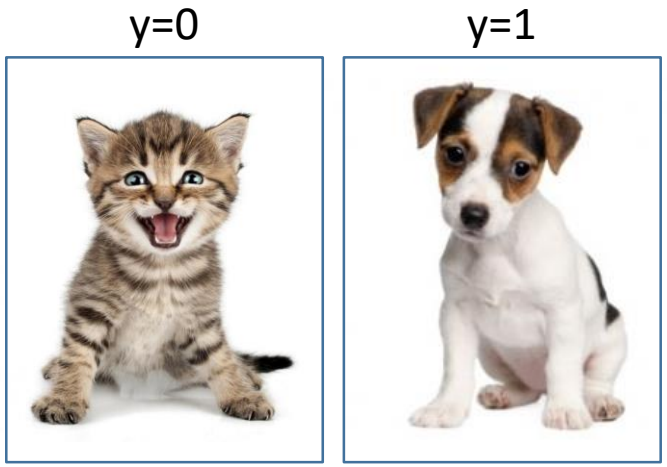
Varianza alta
(overfitting)

Por desgracia, no es fácil representar problemas de altas dimensionalidad (como las imágenes) para poder ver cual es la línea de separación entre categorías.

Pero existen otras métricas

Varianza/Sesgo

REPASO



Error humano ~ 0%

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%

Error óptimo (Error de Bayes)

Varianza/Sesgo

REPASO

y=0

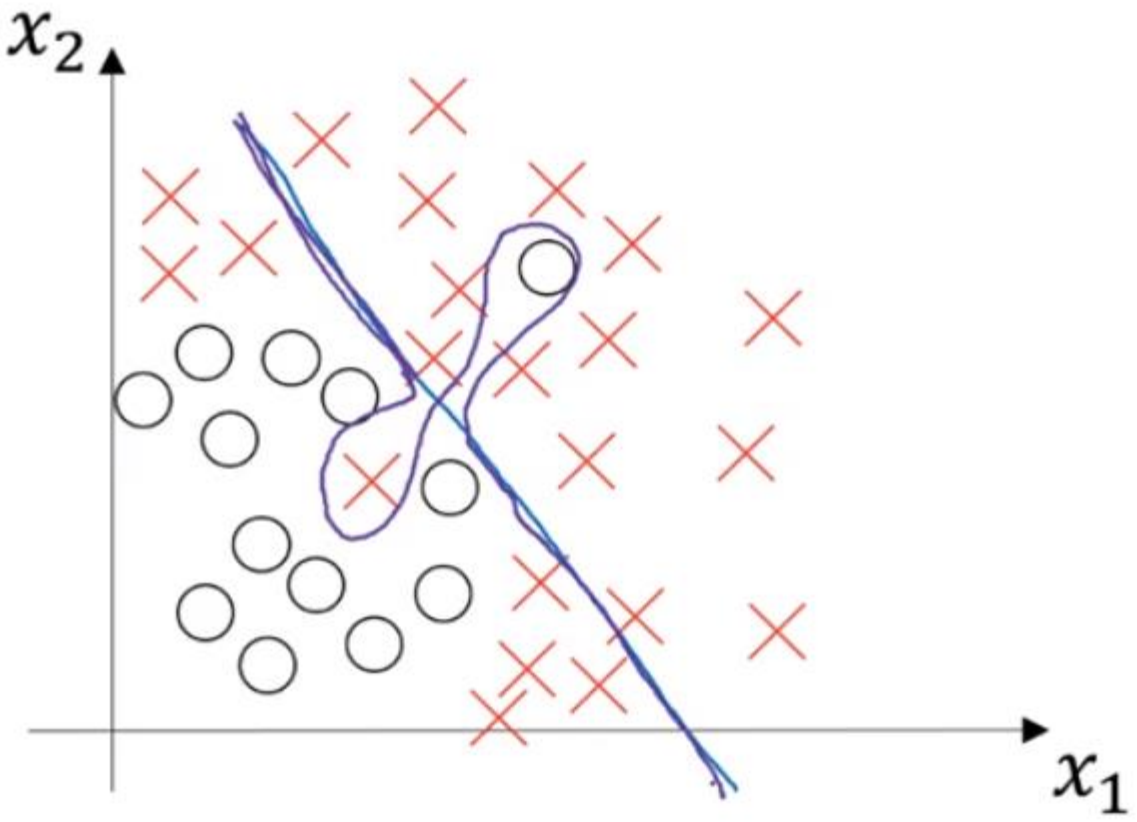
y=1

Error humano ~ 0%

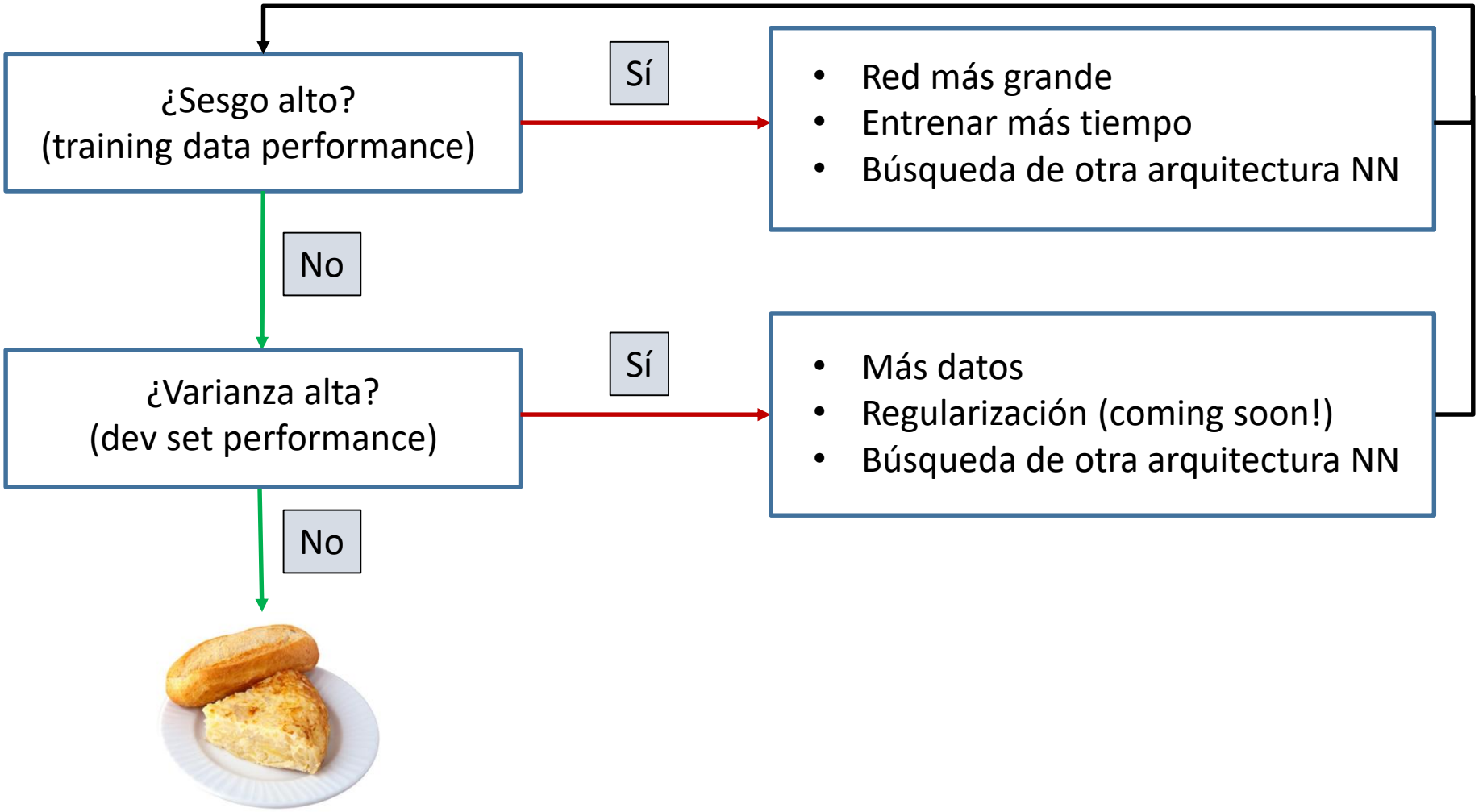
Error óptimo (Error de Bayes)

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	Varianza Alta	Sesgo alto	Sesgo alto Varianza Alta	Sesgo bajo Varianza baja

Varianza alta y Sesgo alto



Receta


¿Sesgo alto?
(training data performance)
Sí
Red más grande
Entrenar más tiempo
Búsqueda de otra arquitectura NN
No
¿Varianza alta?
(dev set performance)
Sí
Más datos
Regularización (coming soon!)
Búsqueda de otra arquitectura NN
No

Regularización: L2

- Si tienes un problema varianza alta (*overfitting*) una de las primeras cosas para probar es conseguir más datos → No siempre es posible.
- En ese caso, el siguiente paso es aplicar algún tipo de **regularización.**

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|w\|^2 \qquad w \in R^{n_x} \qquad b \in R$$

- Regularización L2: $\|w\|^2 = \sum w_j^2 = w^T w$
- **λ:** parámetro de regularización → Se elige en el dev set

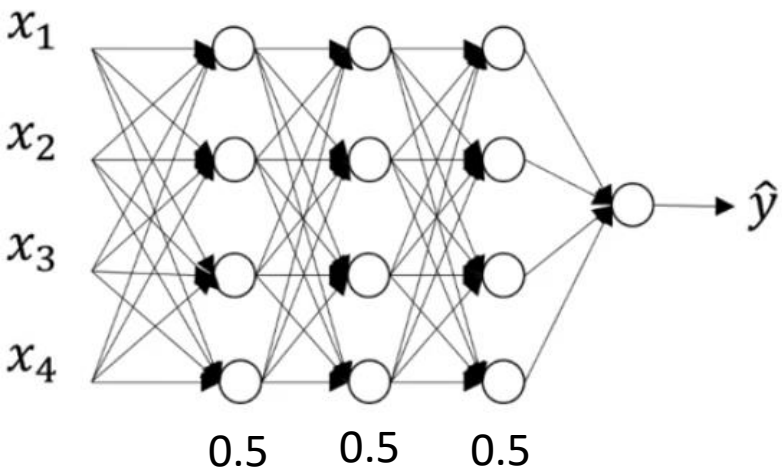
Regularización: L2

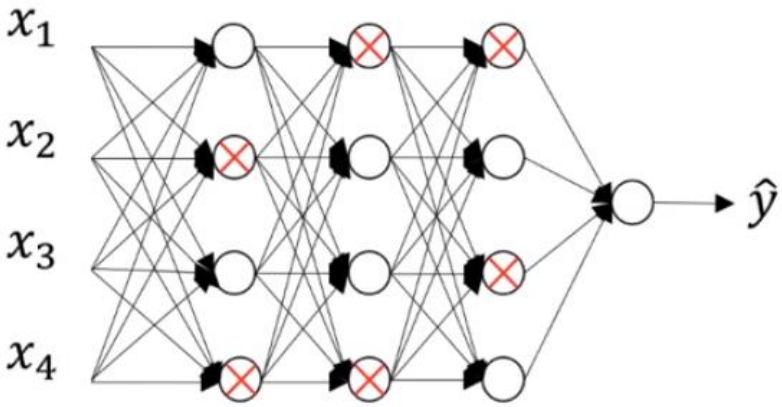
$$J(w,b) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right)}_{\text{Ajustarse al modelo lo mejor posible}} + \underbrace{\frac{\lambda}{2m}\|w\|^2}_{\text{Hacer que los w sean lo menor posibles}}$$

Si los pesos w son lo más pequeños posibles, la red tenderá a ser menos compleja

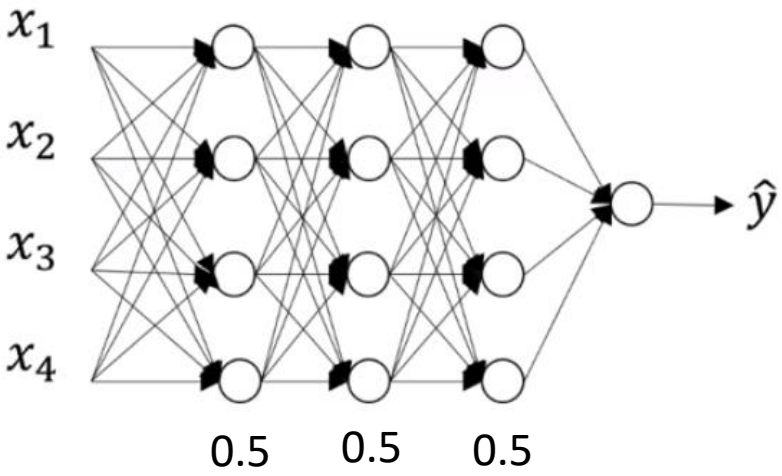
Menos especializada

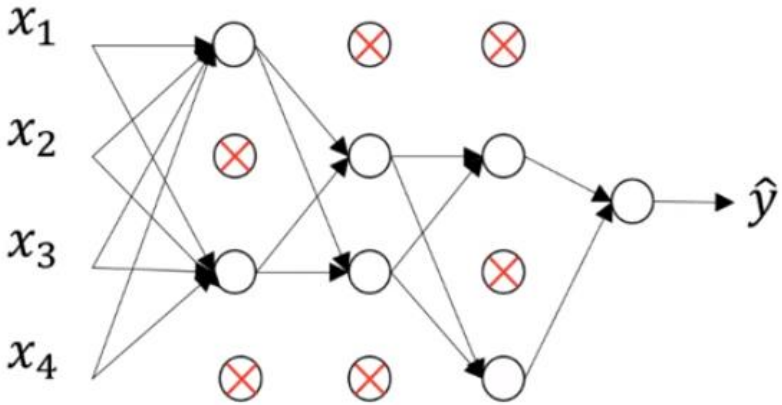
Regularización: Dropout





Regularización: Dropout

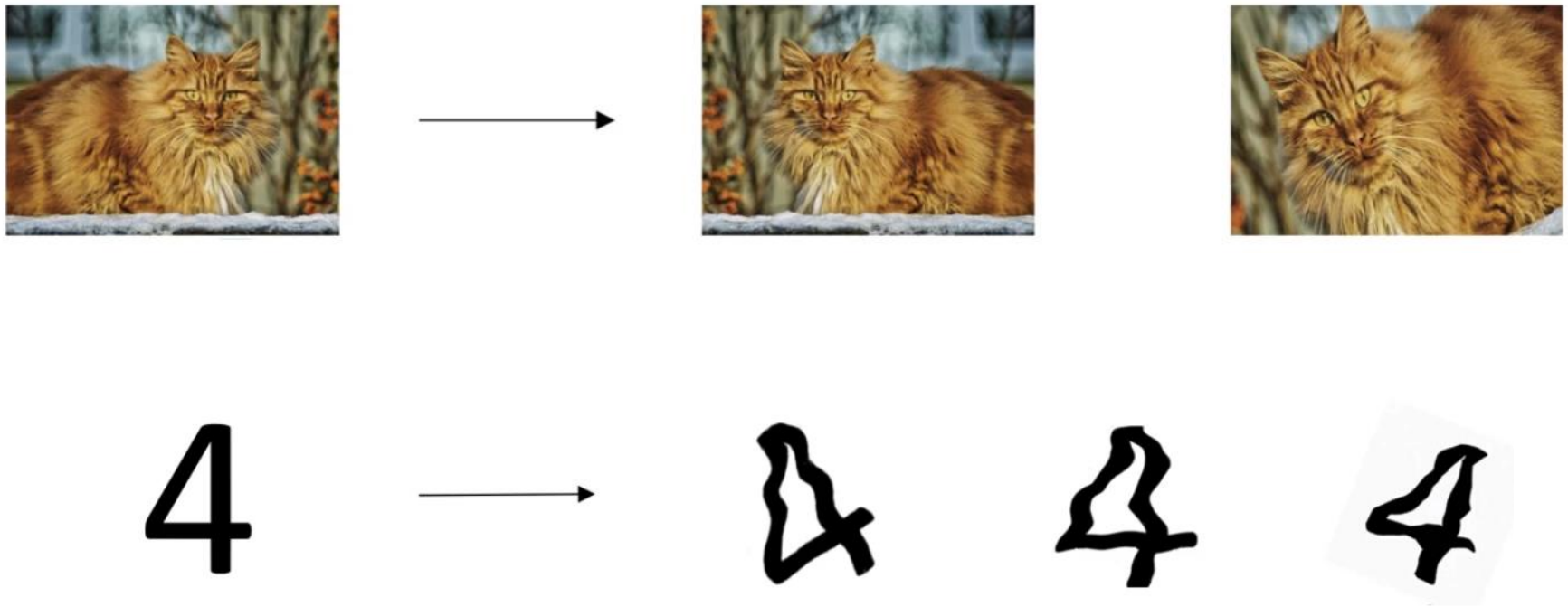




- Simplificas la red
- Evitas que algún nodo se "sobre-especialice"

No puede depender de un input concreto porque éste puede desaparecer en cualquier momento

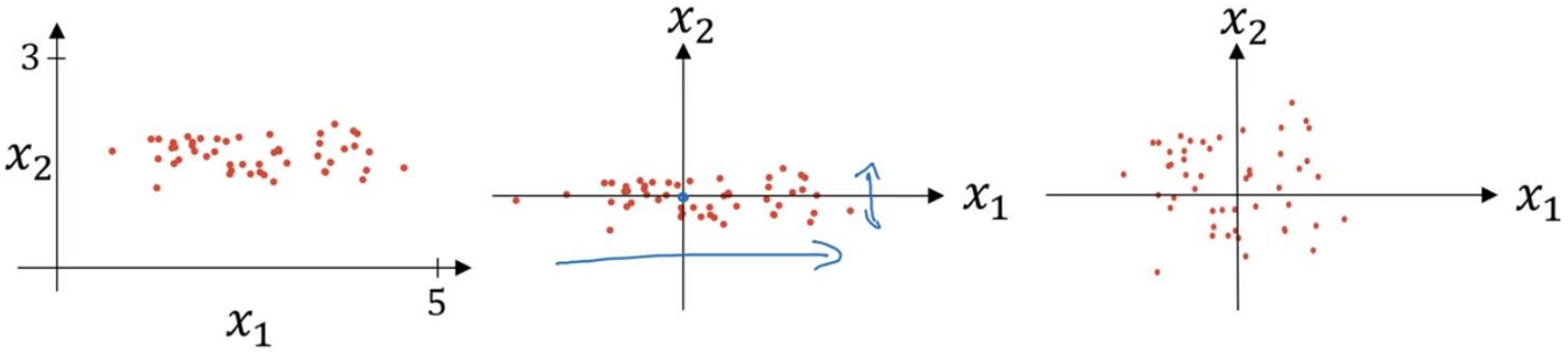
Regularización: Data Augmentation



No funciona tan bien como conseguir más datos independientes pero es gratis ☺

Normalización del input

- Input $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



Substraemos la media:

$$\mu = \frac{1}{m}\sum_{I=1}^{m} x_i \qquad x = x - \mu$$

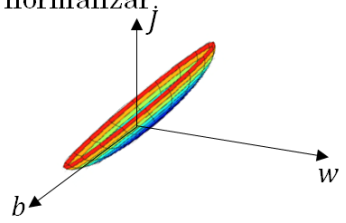
Normalizar por la desviación estándar:

$$\sigma^2 = \frac{1}{m}\sum_{I=1}^{m} x_i{}^2 \qquad x /= sqrt(\sigma^2)$$

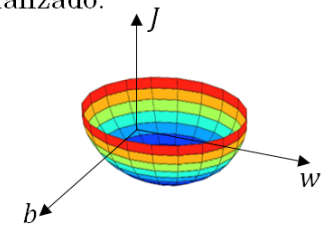
Normalización del input

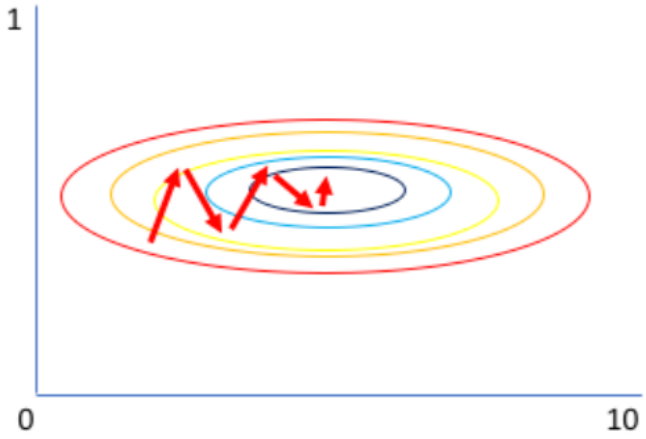
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Sin normalizar:

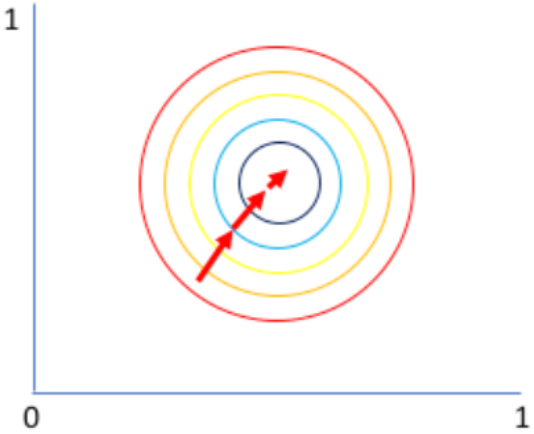


Normalizado:





Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

Inicialización de los pesos

$$z = w_1x_1 + w_2x_2 + w_3x_3 + \ ... + \ w_nx_n$$

var(Z) = var(X) → Si estás usando ReLU, para conseguir esto:

$W^{[l]}$ = np.random.randn(shape)* np.sqrt ($2/n^{[l-1]}$)

Inicialización de pesos Xavier (sigmoide y tanh)

"Understanding the difficulty of training deep feedforward neural networks"

Inicialización de pesos He (ReLU)

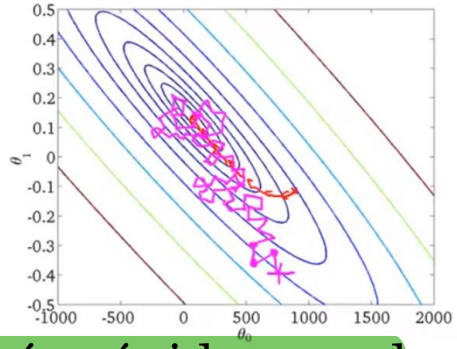
"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"

Stochastic Gradient Descent

- Batch gradient descent: se actualizan los pesos tras haber recorrido todo el dataset (una época entera)

Stochastic gradient descent:

- Los pesos se actualizan para cada muestra del dataset de training
- Cada actualización intenta mejorar la predicción para ajustarse a una única muestra
- Cada iteración será extremadamente rápida
- Conviene hacer un *random shuffle* de los datos



- El stochastic gradient descent puede ser mucho más rápido que el batch gradient descent para datasets muy grandes
- Puede tener problemas convergiendo, una posible solución:

$$\alpha = \frac{const\_1}{n_{iteración} + const\_2}$$

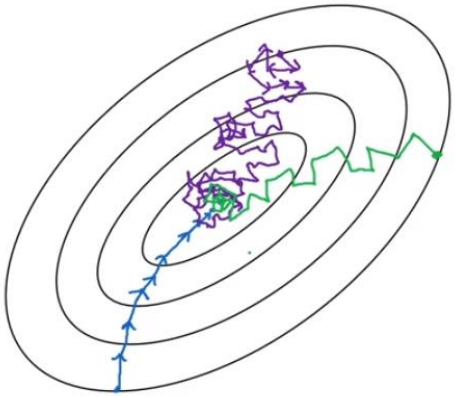
Mini-Batch gradient descent

- Batch gradient descent: se actualizan los pesos tras haber recorrido todo el dataset (una época entera)
- Stochastic gradient descent: se actualizan los pesos muestra a muestra

Mini-batch gradient descent:

- Dividir el dataset de entrenamiento en fragmentos más pequeños (mini-batches)
- Los parámetros de la red se actualizan para cada mini-batch.
- Si tienes una buena vectorización el mini-batch gradient descent puede ser más rápido que el stochastic (paralelización)
- El tamaño del mini-batch depende de tu muestra

Stochastic, Mini-batch y Batch



Stochastic gradient descent

Actualizas los parámetros sin tener que recorrer toda la muestra

Se pierde la potencia de la vectorización

Puede tener problemas de convergencia en una región relativamente grande

Mini-Batch gradient descent:

Aprendizaje más rápido:

Actualizas los parámetros sin tener que recorrer toda la muestra

Explotas la vectorización

Mejor convergencia

Batch gradient descent:

Si la muestra de entrenamiento es muy grande, cada iteración lleva mucho tiempo

¿Cómo elegir el tamaño del mini-batch?

- Si tienes una muestra pequeña: usa Batch Gradient Descent (m < 2000)

- Si tienes una muestra grande:
  - Típicos tamaños para el mini-batch: 64, 128, 256…1024
  - Por como funciona el acceso a memoria en los ordenadores suele funcionar mejor que sea una potencia de 2
  - Estar seguro de que tu mini-batch entra dentro de la memoria de la CPU/GPU
  - En la práctica el tamaño del mini-batch es otro hiperparámetro: hacer pruebas con unos cuantos y decidir cuál es mejor

Batch normalization

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

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

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

- γ y β se aprenden durante el entrenamiento
- Red más pesada (más parámetros) pero más estable

Local vs Global

¿Cómo podemos estar seguros de que el algoritmo ha convergido a un mínimo global y no a un mínimo local?

No podemos.

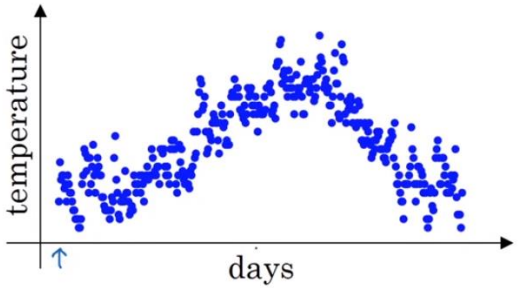
Pero afortunadamente no lo necesitamos.

Se ha demostrado empíricamente que a pesar de la no-convexidad siempre se llega a una solución razonable: los mínimo locales son raros y muy similares entre ellos y al mínimo local.

Si alguien se queda con ganas de una visión más teórica:

https://arxiv.org/abs/1412.0233 https://arxiv.org/abs/1406.2572

Media Móvil Exponencial



- Si quieres calcular la tendencia (media móvil) de la temperatura:

$$V_t=\beta V_{t-1} + (1-\beta)\theta_t$$

V_t es como sacar la media de temperatura de los últimos $\frac{1}{1-\beta}$ días

$\beta = 0.9 \sim 10$ días
$\beta = 0.98 \sim 50$ días
$\beta = 0.5 \sim 2$ días

Si tienes un valor de β muy alto → se adapta mal a cambios bruscos ya que da más peso a la temperatura anterior que a la presente (latencia)

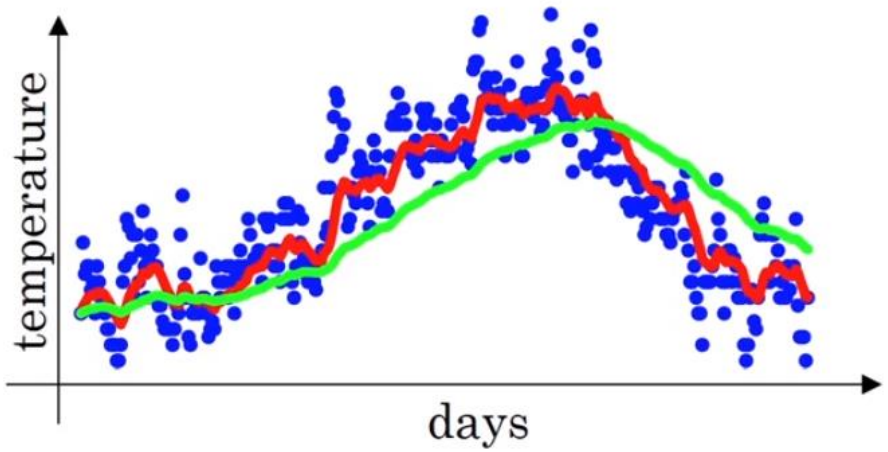
Si tienes un valor de β muy bajo → Más susceptible de dejarse influir por valores atípicos

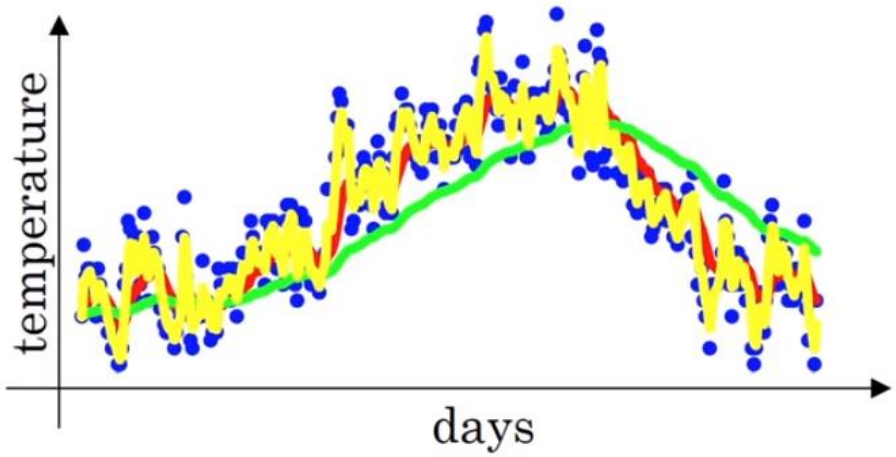
Media Móvil Exponencial

$\beta = 0.9 \sim 10$ días

$\beta = 0.98 \sim 50$ días

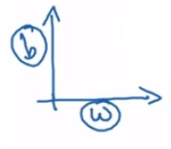
$\beta = 0.5 \sim 2$ días



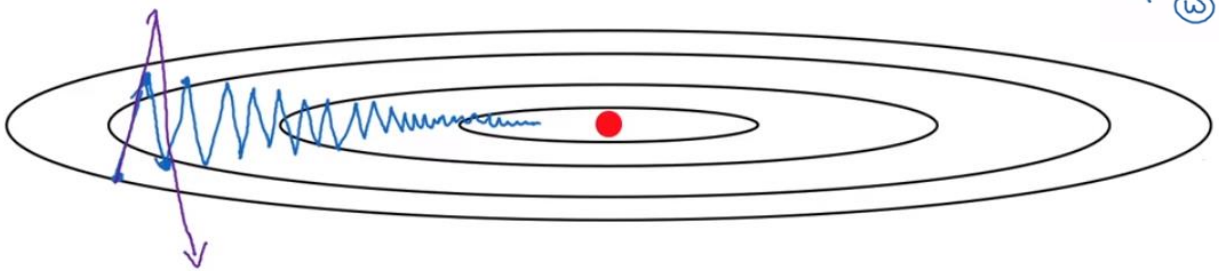


Gradient Descent con momento

- Usar la media móvil exponencial de los gradientes y actualizar los pesos con ese valor
- Casi siempre funciona más rápido que el *mini-batch gradient descent* normal



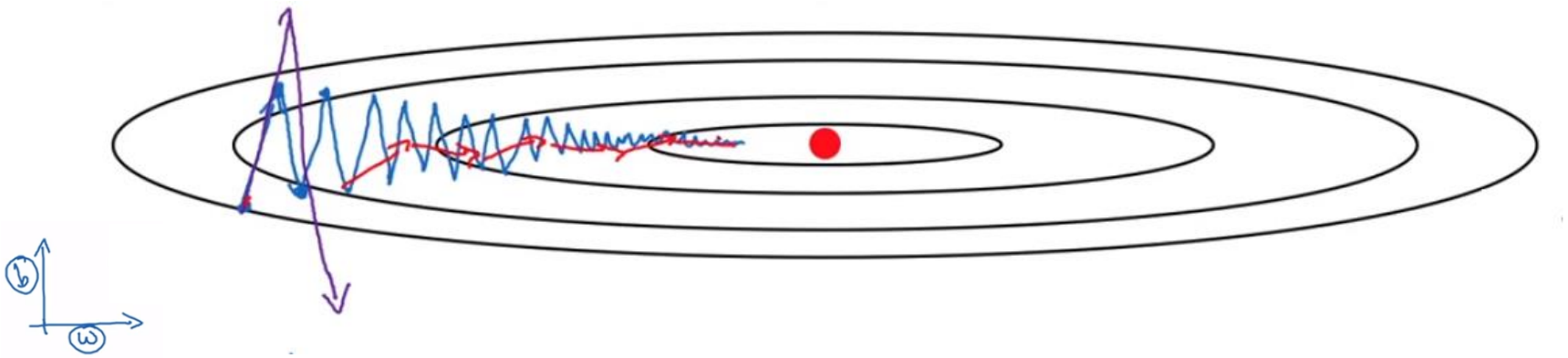
Las oscilaciones hacen que no puedas coger un α grande



En la iteración sobre cada mini-batch:

$$Calcula\ dW\ y\ db$$
$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$
$$v_{db} = \beta v_{db} + (1-\beta)db$$
$$W = W - \alpha v_{dW} \quad b = b - \alpha v_{db}$$

Gradient Descent con momento



- Hiperparámetros: α y β
- El valor más comúnmente usado es β=0.9 (~ últimos 10 gradientes)
- Se puede probar para ver si hay algún otro valor que funcione mejor

Otros Algoritmos de Optimización

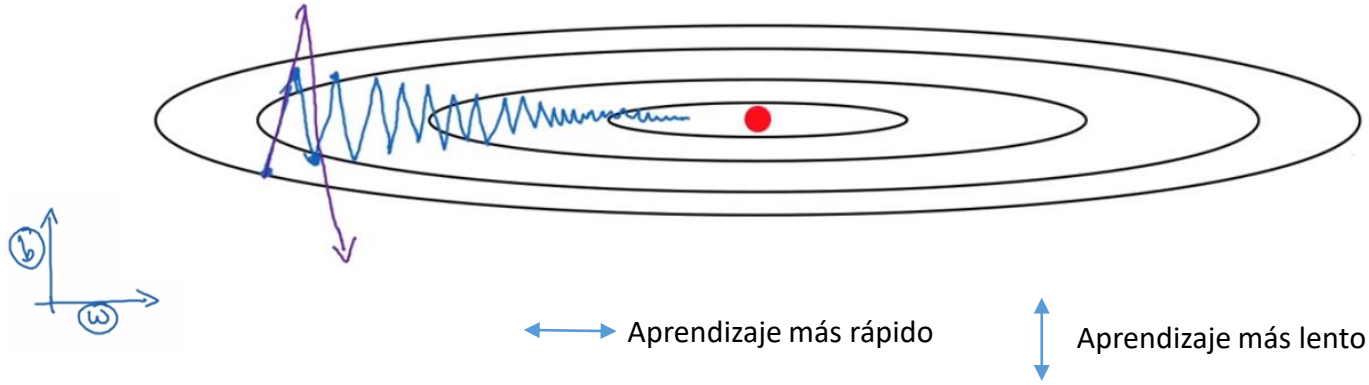
- Muchos de los algoritmos de optimización no generalizan bien → reticencia frente a los nuevos algoritmos

- De los pocos que han conseguido hacerse un camino frente al *gradient descent con momento*.

RMSProp

Adam Optimization

RMSProp



En la iteración sobre cada mini-batch:

$$Calcula\ dW\ y\ db$$

$$s_{dW} = \beta s_{dW} + (1-\beta) dW^2$$ ← pequeño

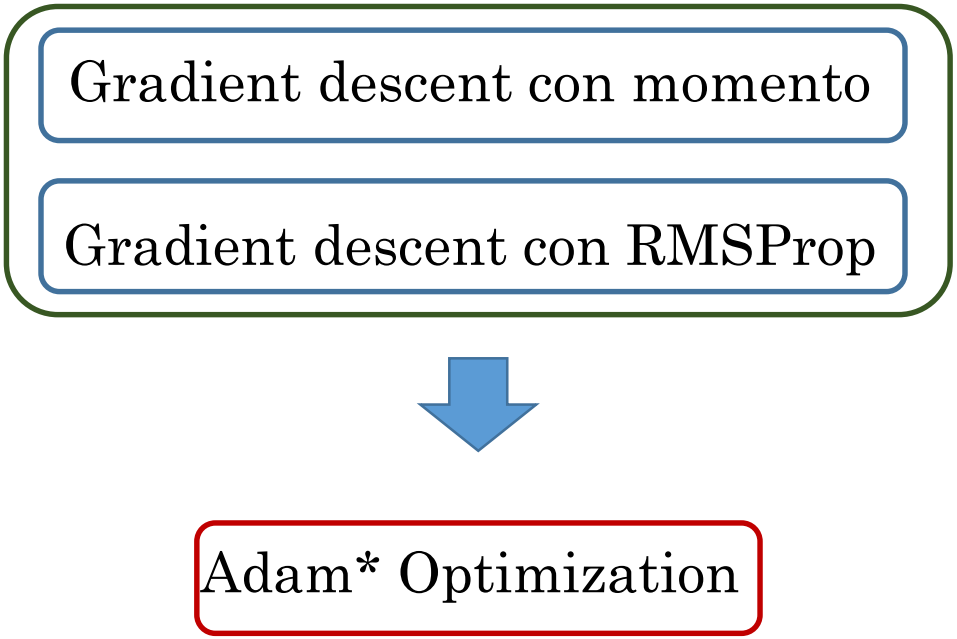
$$s_{db} = \beta s_{db} + (1-\beta) db^2$$ ← grande

$$W = W - \alpha \frac{dW}{\sqrt{s_{dW}} + \varepsilon} \qquad b = b - \alpha \frac{db}{\sqrt{s_{db}} + \varepsilon}$$

Consigues regular la oscilación → puedes usar α mayores

Adam

Ponemos juntos



*Adaptive Moment estimation

Adam Optimization

$v_{dw}=0$, $s_{dw}=0$, $v_{db}=0$, $s_{db}=0$

En la iteración sobre cada mini-batch:

Calculamos dW, db

$$v_{dW} = \beta_1 v_{dW} + (1-\beta_1) dW$$
$$v_{db} = \beta_1 v_{db} + (1-\beta_1) db$$

Momento

$$s_{dW} = \beta_2 s_{dW} + (1-\beta_2) dW^2$$
$$s_{db} = \beta_2 s_{db} + (1-\beta_2) db^2$$

RMSProp

$$W = W - \alpha \frac{v_{dW}}{\sqrt{s_{dW}} + \varepsilon} \qquad b = b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \varepsilon}$$

Adam Optimization

Hiperparámetros:

α : hay que tunearlo

β_1: 0.9

β_2: 0.999

ε: 10^{-8}

Recomendaciones del artículo original de Adam

Resumiendo

- Como dividir el dataset de entrenamiento
- Repaso de varianza y sesgo
- Regularización: L2, dropout, data augmentation
- Minibatch y stochastic gradient descent
- Algoritmos de optimización: Momento, RMSProp y Adam

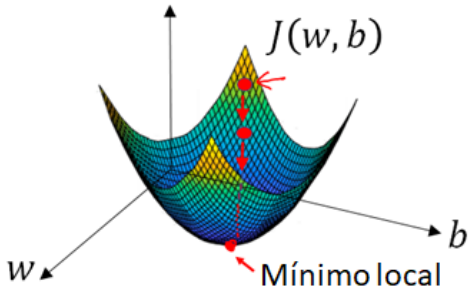
Backup

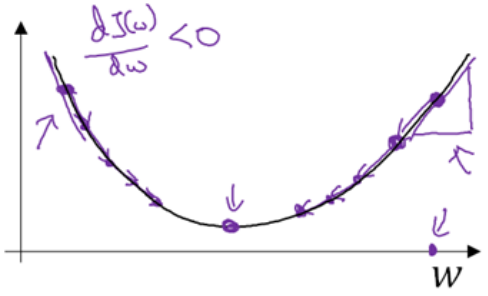
Repaso: Minimización

$$\hat{y} = \sigma(w^T x + b), \quad \sigma(z) = \frac{1}{1+e^{-z}}$$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

Want to find w, b that minimize $J(w,b)$





$$w := w - \alpha \frac{\partial J(w,b)}{\partial w}$$

$$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$$