

# Redes Neuronales

## Sobre-ajuste (Overfitting)

# Hay dos conceptos que van de la mano

## Sub-ajuste

El modelo no ha capturado la lógica subyacente de los datos

El modelo no sabe qué hacer y por lo tanto da respuestas lejos de lo correcto.

## Sobre-ajuste

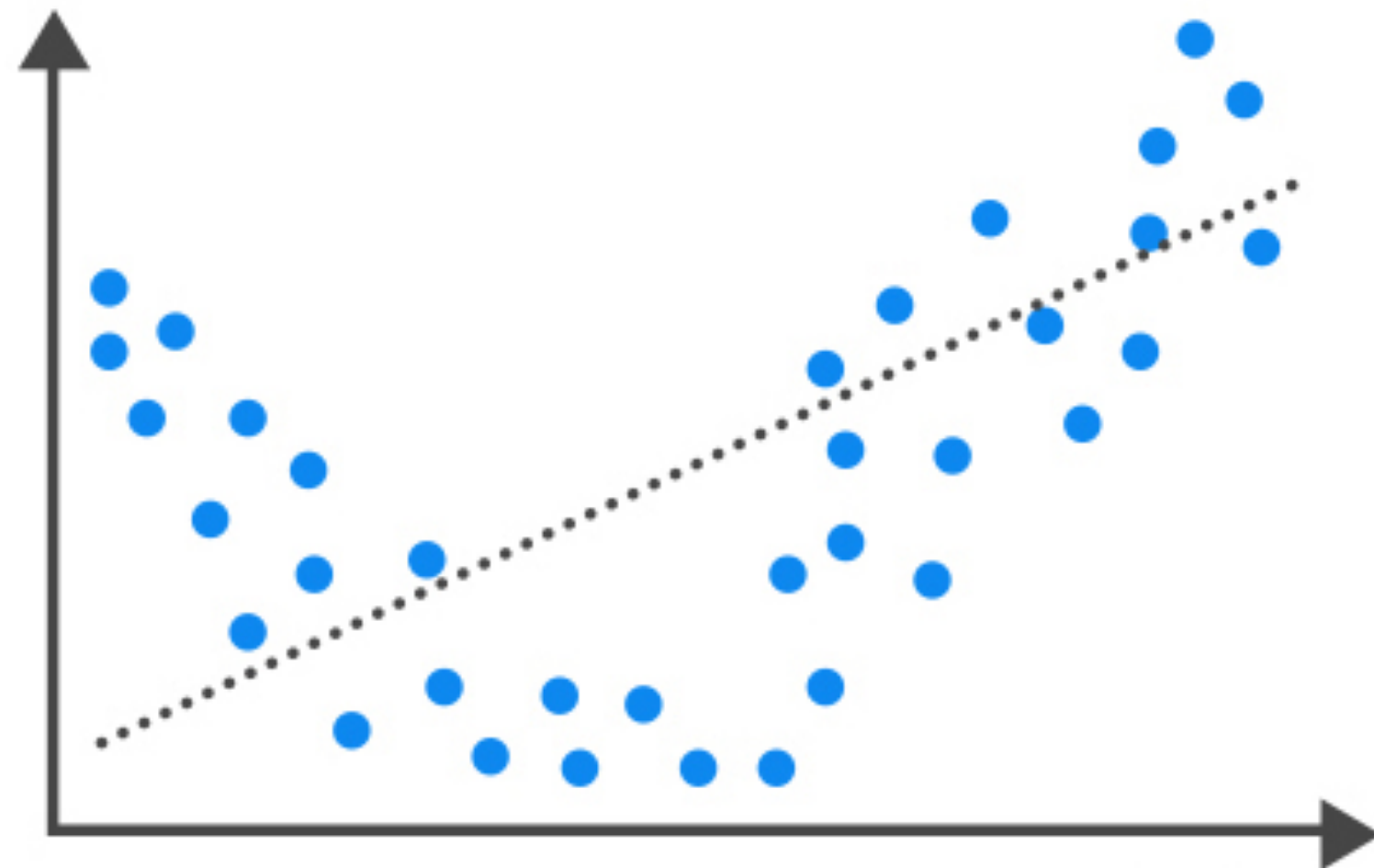
El entrenamiento se ha enfocado tanto en el conjunto de datos, que no “entendió el objetivo” del entrenamiento

Da respuestas correctas, pero solo para los datos de entrenamiento

# Gráficamente

Pérdida alta (Loss)

Exactitud baja

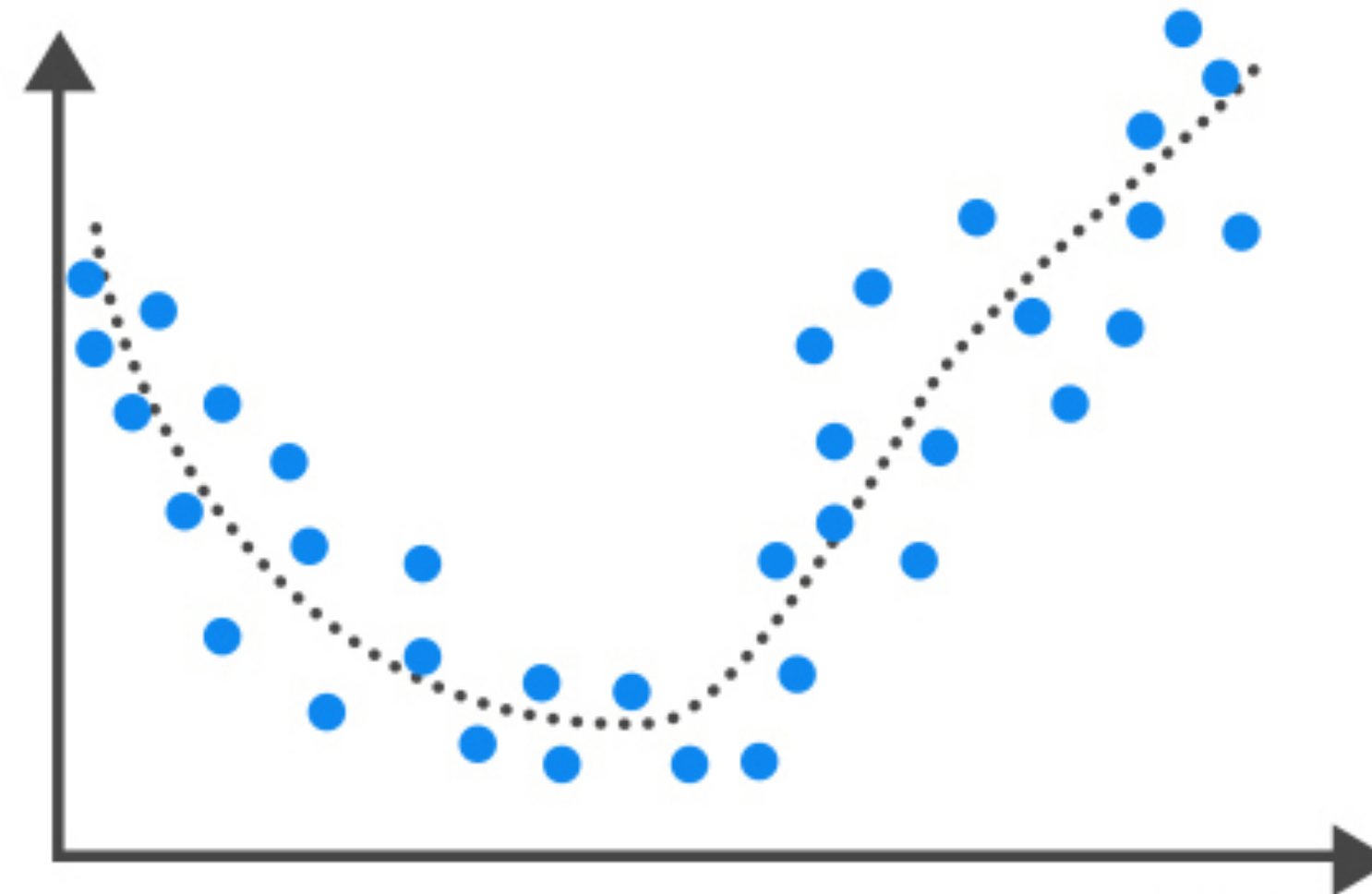


Sub-ajustado

(Alto error de sesgo)

Pérdida baja (Loss)

Exactitud alta

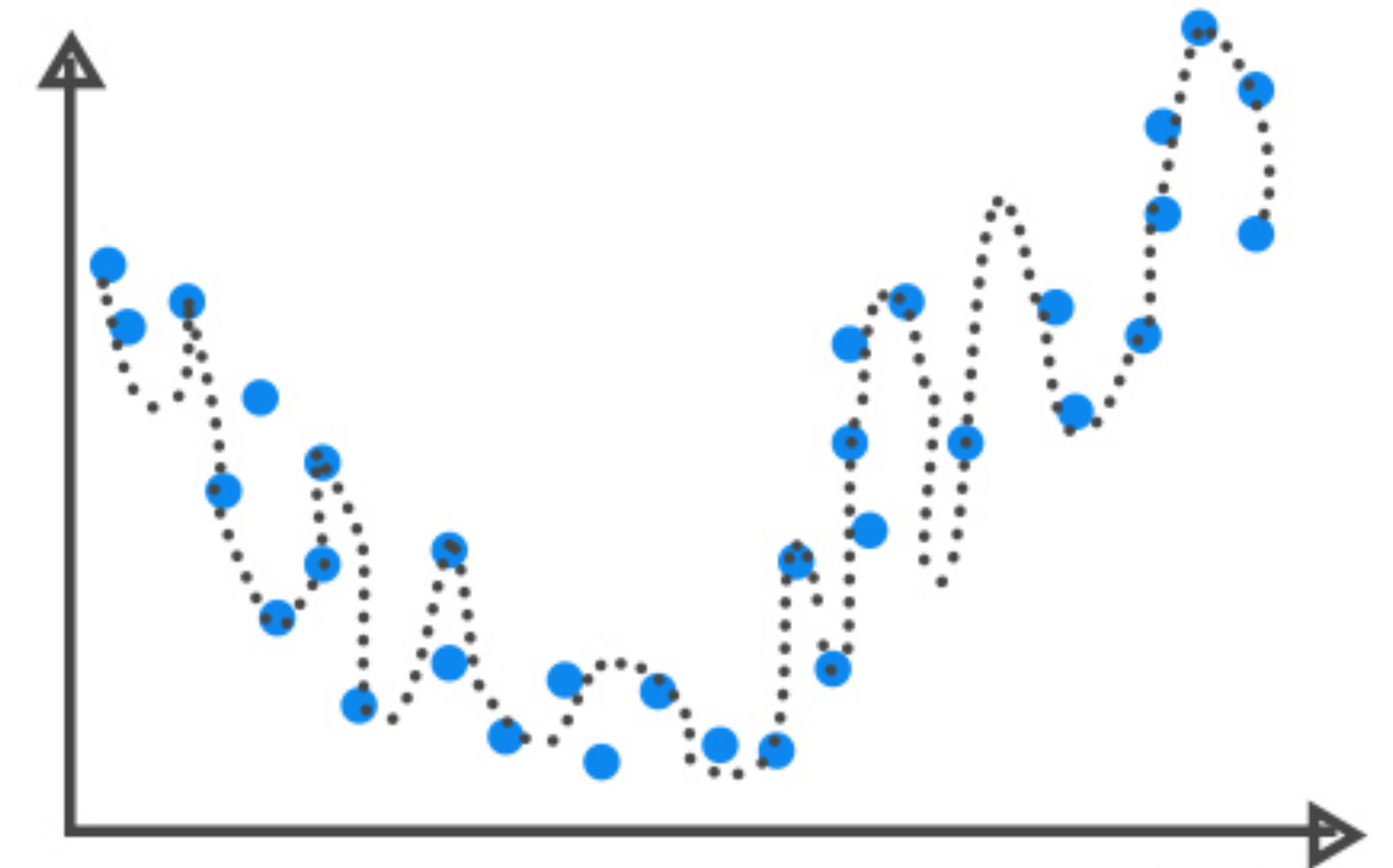


Buen ajuste/Robusto

(Balance entre sesgo y  
varianza)

Pérdida baja (Loss)

Exactitud baja



Sobre-ajustado

(Alto error de varianza)

# Ejemplo

## Predecir el cambio para el Euro

- Basado en 50 indicadores comunes
- El entrenamiento da un 99.99% de exactitud
- El modelo se lanza al mercado y pierde todo su dinero
- Problema:
  - Modelo se ajustó demasiado a los datos, incluyendo el ruido introducido por los inversionistas de la bolsa de valores de ese día

**La primera regla de la  
programación:**

**La computadora no se equivoca,  
somos nosotros los que erramos!**

**Las gráficas simples podrían llevarnos a pensar que podemos encontrar el problema fácilmente.**

**Pero hay que recordar que en este ejemplo hay 50 variables...una gráfica de 50 dimensiones escapa nuestra capacidad de visualizar.**

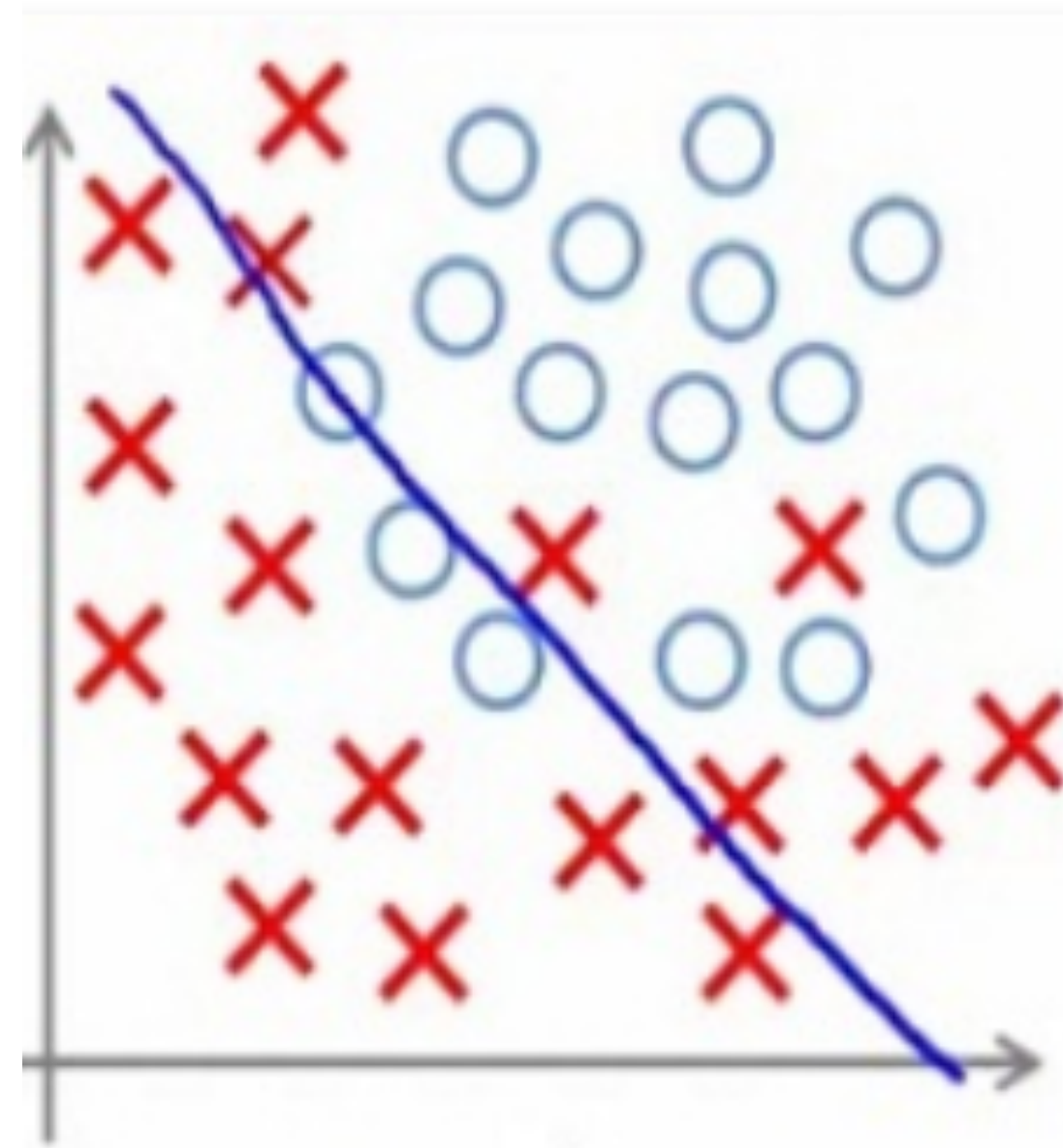
**Más adelante veremos cómo manejar esto de una forma más fácil.**



# Sub y sobre ajuste

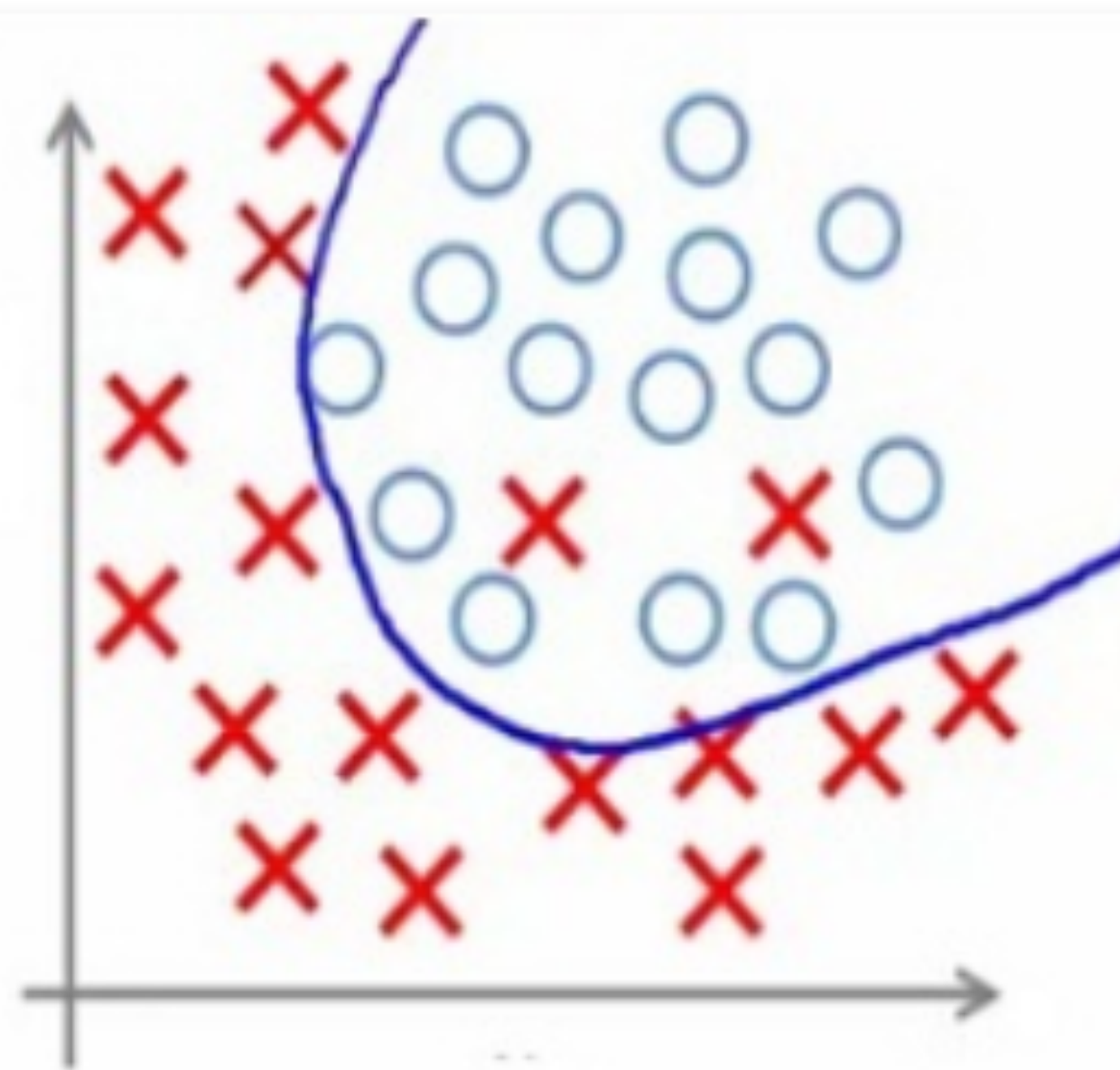
## Ejemplo de clasificación

X Perros  
O Gatos



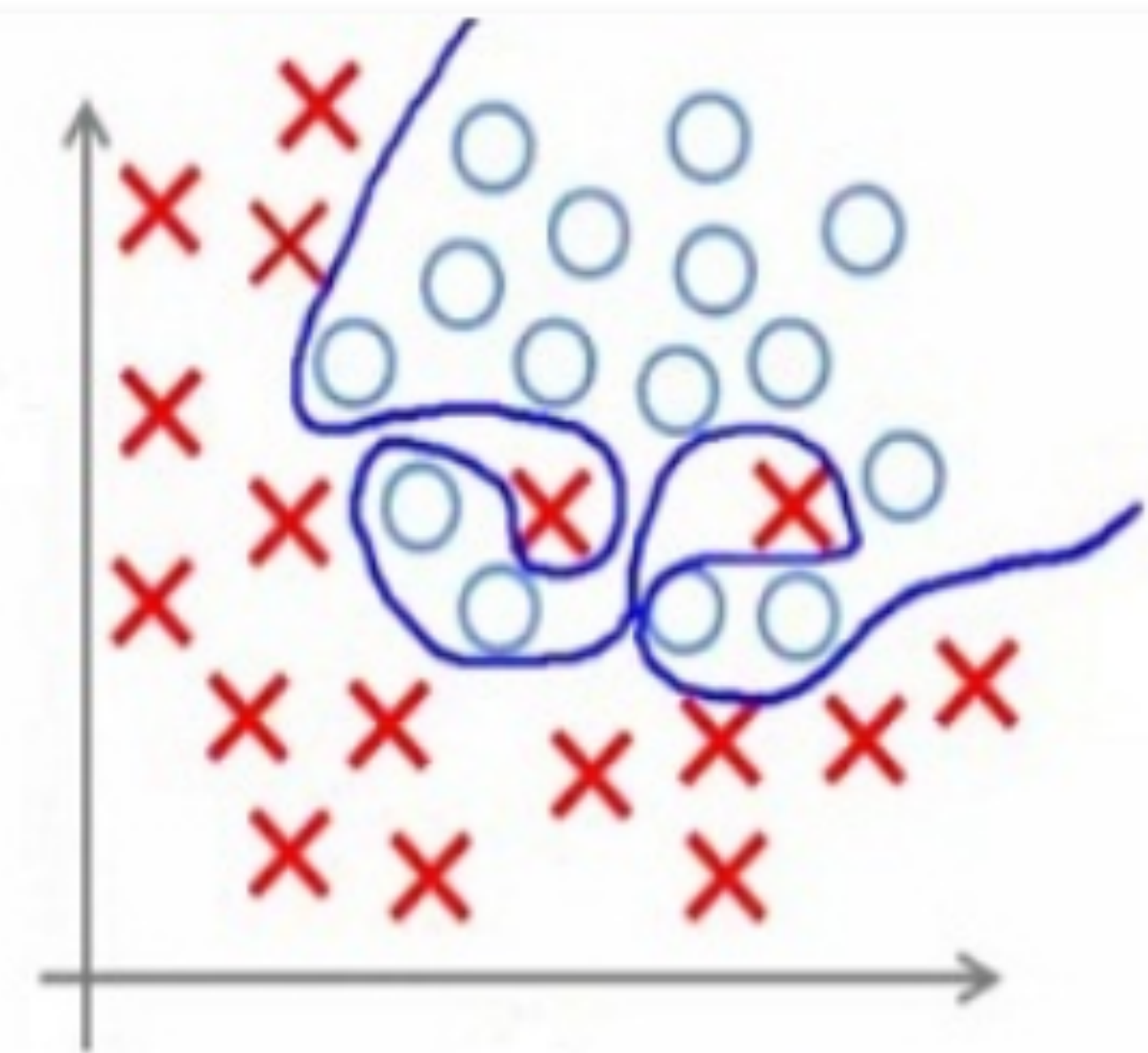
Sub-ajustado

- No captura la lógica subyacente
- Perdida alta (Loss)
- Exactitud baja



Bien ajustado

- Captura la lógica subyacente
- Perdida baja (Loss)
- Exactitud alta



Sobre-ajustado

- Captura todo el ruido
- Perdida baja (Loss)
- Exactitud baja

**El balance afinado entre el sub y sobre  
ajuste se denomina la compensación  
Sesgo - Varianza**



# Un meme popular



Se ajusta perfectamente a la persona....  
pero pierde totalmente lo que es el concepto de una cama

¿Qué es validación?

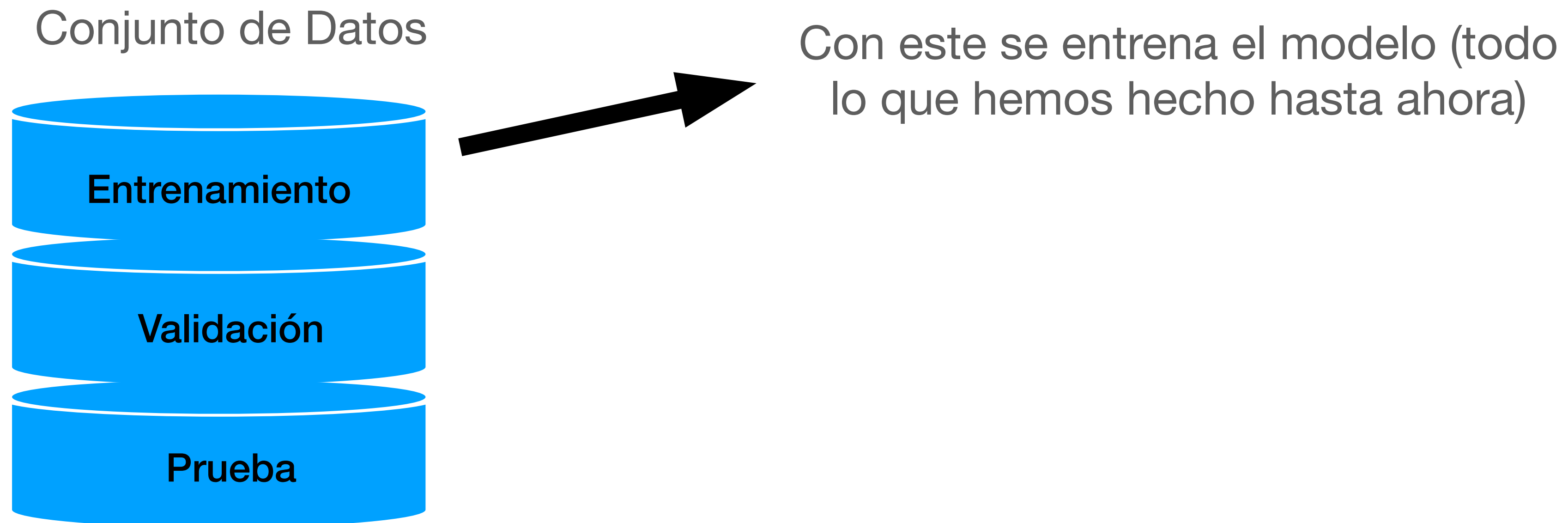
**El verdadero problema en ML es el sub o sobre  
ajuste !**

**Ahora veremos cómo se pueden evitar.**

# Entrenamiento, validación y prueba

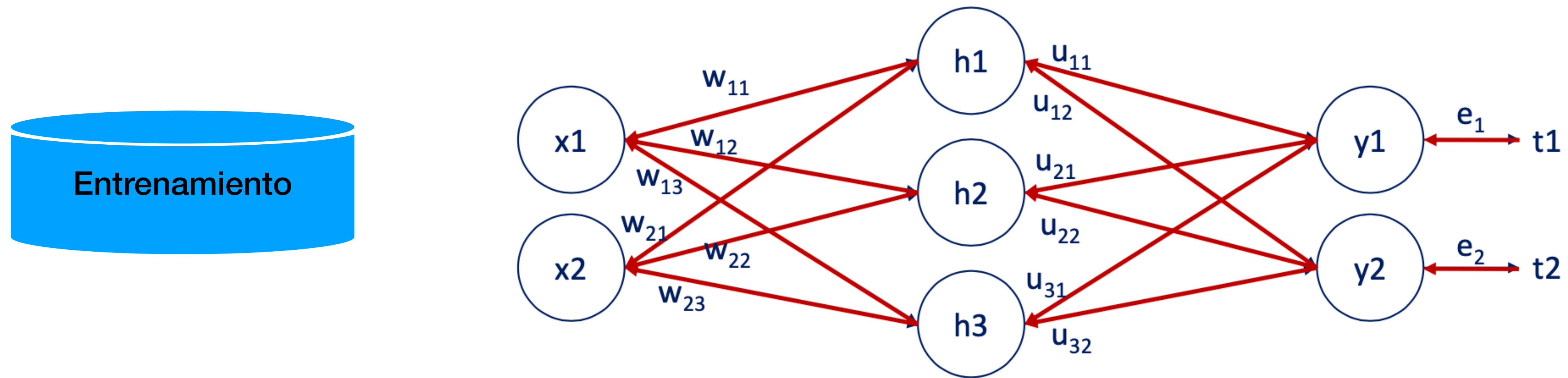
**Para evitar el sobre o sub-ajuste, hay que identificarlo**

Generalmente se puede detectar el sobre-ajuste, por medio de dividir nuestro conjunto de datos en tres subconjuntos:





# Entrenamiento

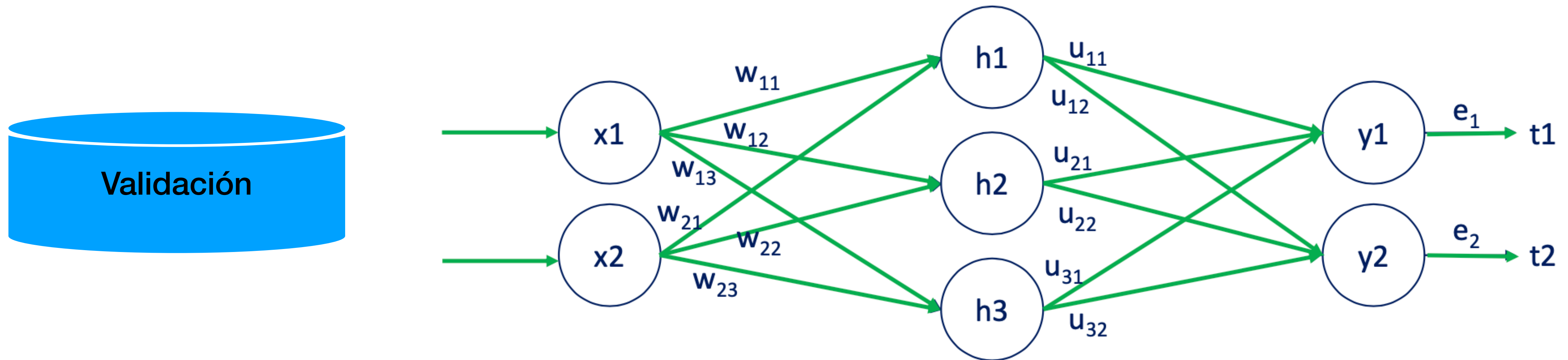


Con este conjunto se entrena el modelo, hay propagación hacia adelante y hacia atrás, que es cuando se reajustan los pesos y sesgos.

En algún momento se detiene el proceso: el modelo está “algo” entrenado y se pasa a la siguiente fase:



# Validación



Con este conjunto solo se corre el modelo, propagación hacia adelante. (No se modifican los pesos y sesgos). Se calcula la función de pérdida.

En promedio la pérdida de validación debiera ser igual a la pérdida del entrenamiento. Es lógico ya que los dos subconjuntos vinieron del mismo conjunto.

**Normalmente se repiten estos procesos muchas veces:  
entrenamiento y validación.**

**Como la función de pérdida del entrenamiento es  
basado en el descenso por gradiente, cada valor  
subsecuente debiera ser menor que la anterior.**

**La pérdida de entrenamiento se está minimizando!**

En el caso de las pérdidas de validación, como no se están cambiando los pesos y los sesgos, en algún momento empieza a incrementarse.

Esa es una señal de alerta “bandera roja”:

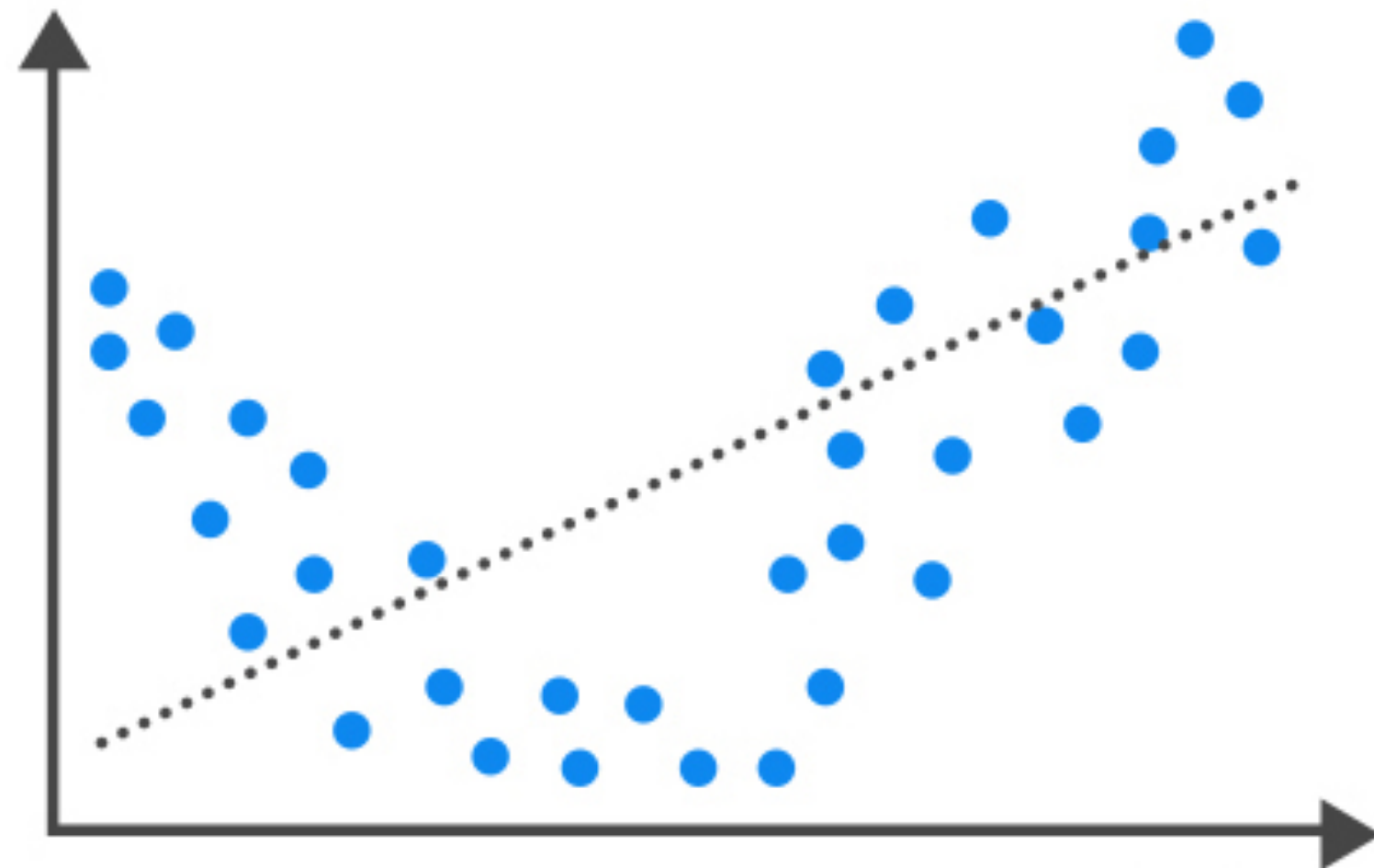
Estamos mejorando en la predicción del conjunto de entrenamiento, pero nos estamos alejando de los datos lógicos generales!

Eso es sobre-ajustar y debemos detener el proceso del entrenamiento del modelo!

# Gráficamente

Pérdida alta (Loss)

Exactitud baja

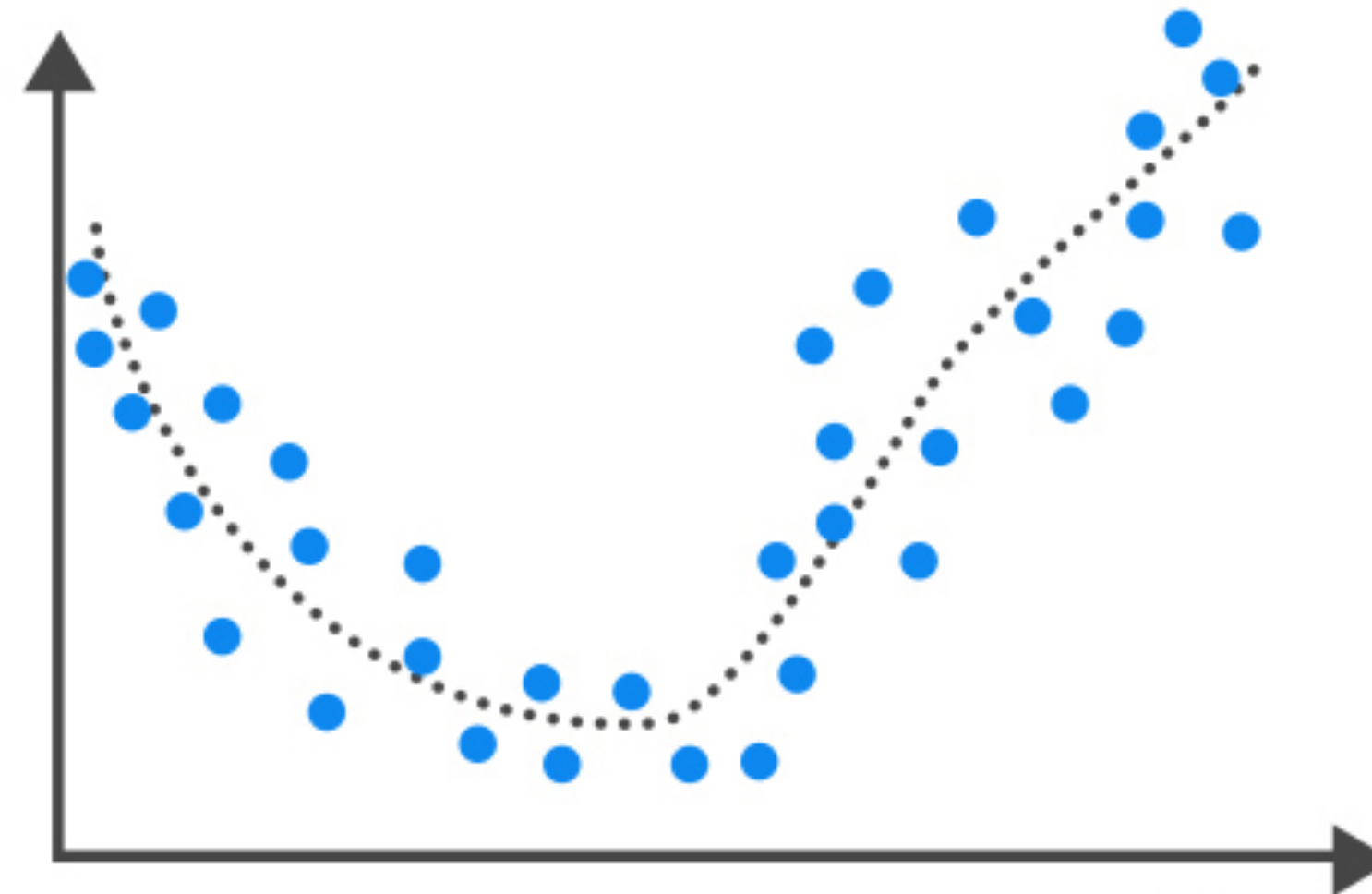


Sub-ajustado

(Alto error de sesgo)

Pérdida baja (Loss)

Exactitud alta

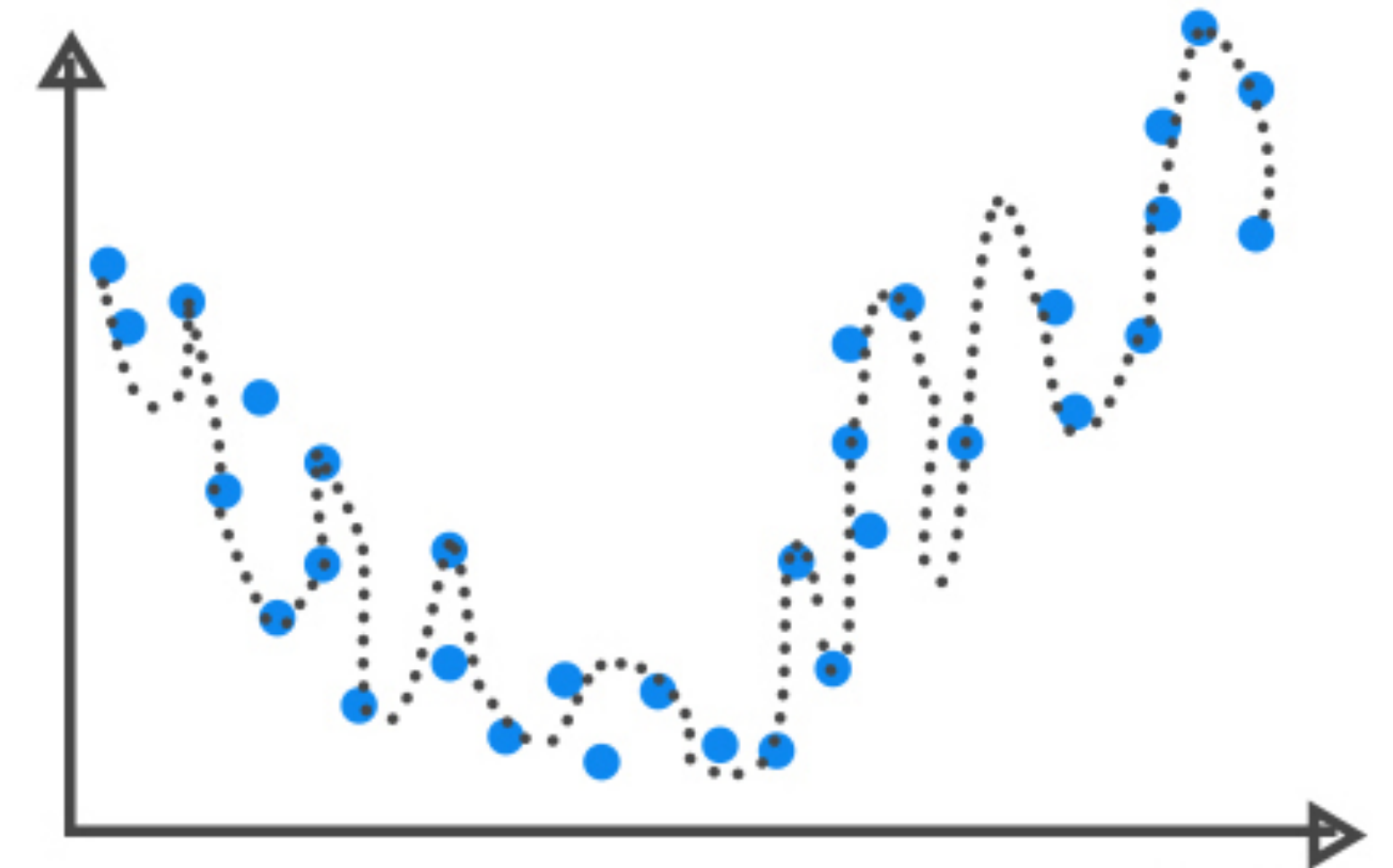


Buen ajuste/Robusto

(Balance entre sesgo y  
varianza)

Pérdida baja (Loss)

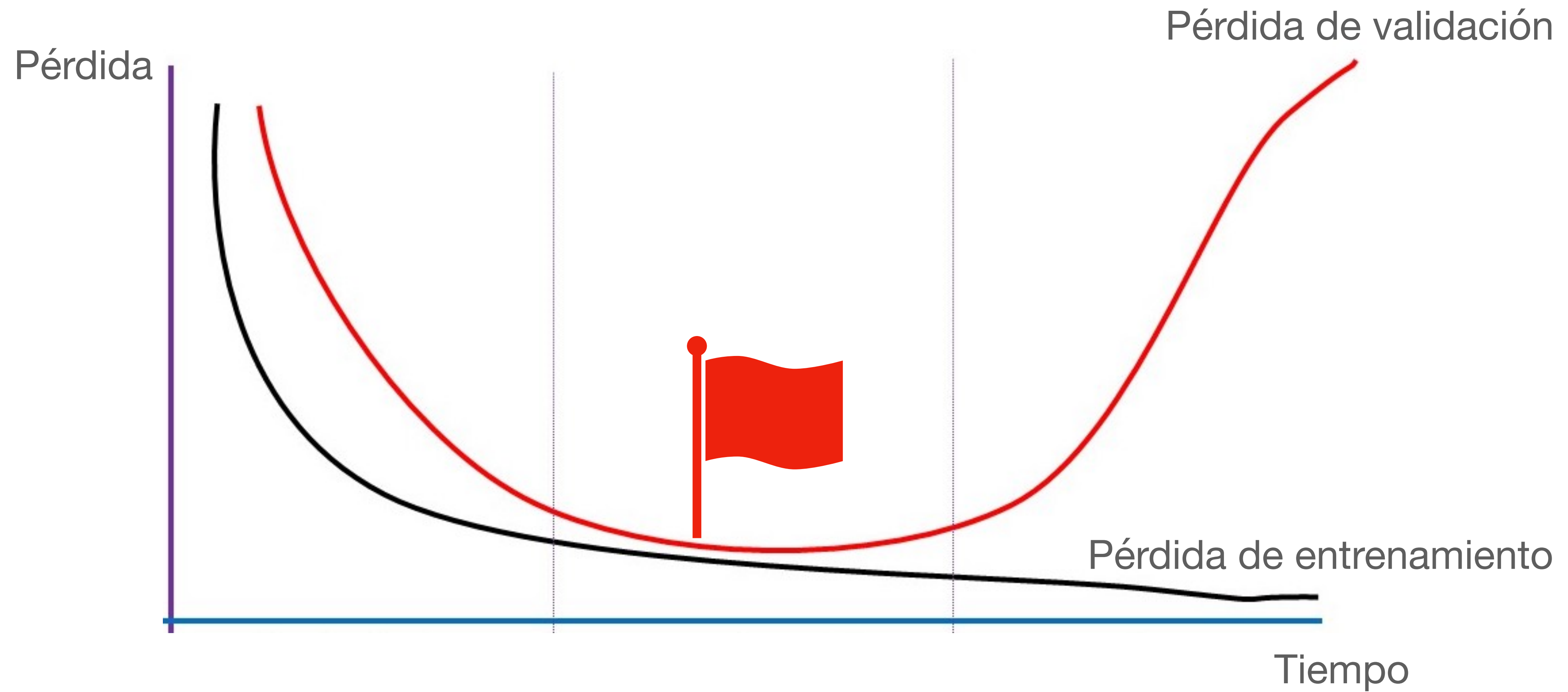
Exactitud baja



Sobre-ajustado

(Alto error de varianza)

# Sobre-ajuste



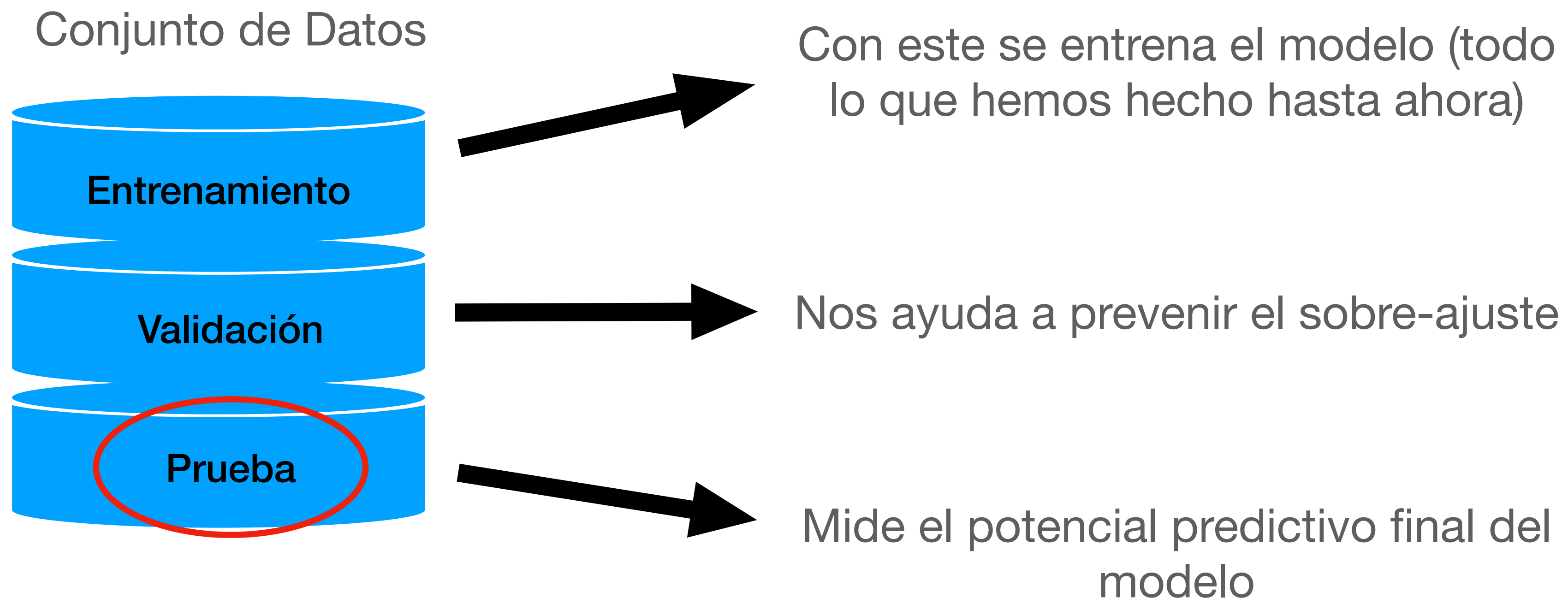


# Sobre-ajuste

- Es sumamente importante que el modelo no entrene con las muestras de validación
- Solo se actualizan los pesos y sesgos con los datos de entrenamiento

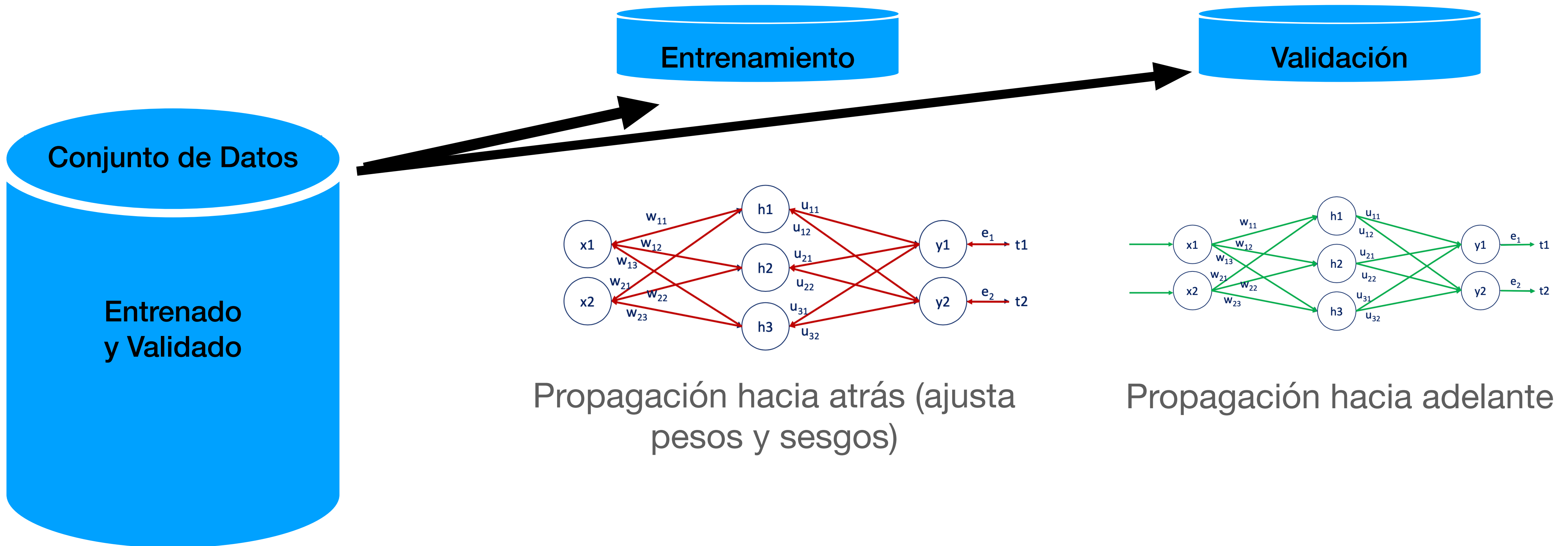
# Entrenamiento, validación y prueba

Luego de entrenar el modelo y validarlo,  
es hora de medir su potencial predictivo

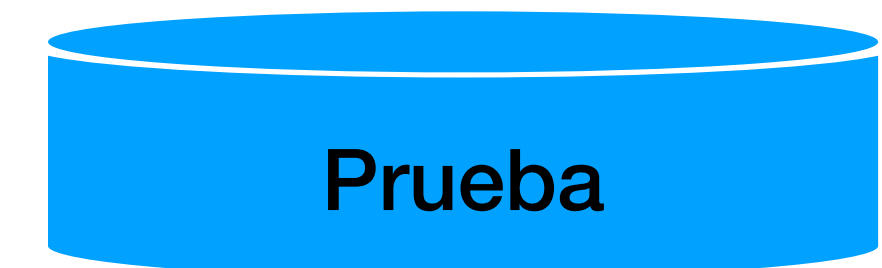


# Entrenamiento, validación y prueba

Tenemos la versión final de la “caja negra” de ML



# ¿Cómo se deben dividir estos sub-conjuntos?



No hay una respuesta estándar, pero las divisiones más comunes son:

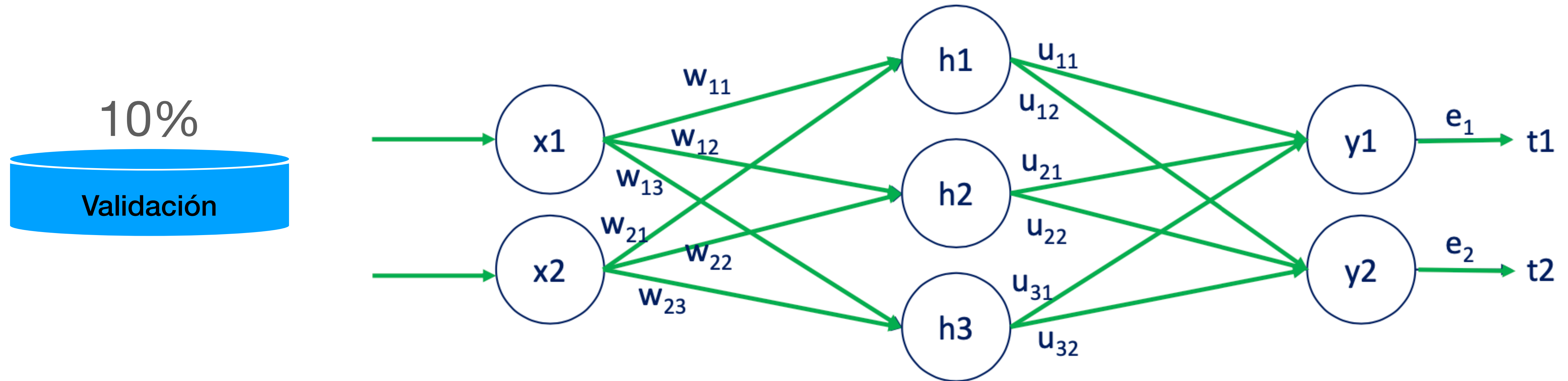
80%  
70%

10%  
20%

10%  
10%

Mencionamos que “de vez en cuando” debemos validar el modelo.

¿Qué quiere decir “de vez en cuando”?



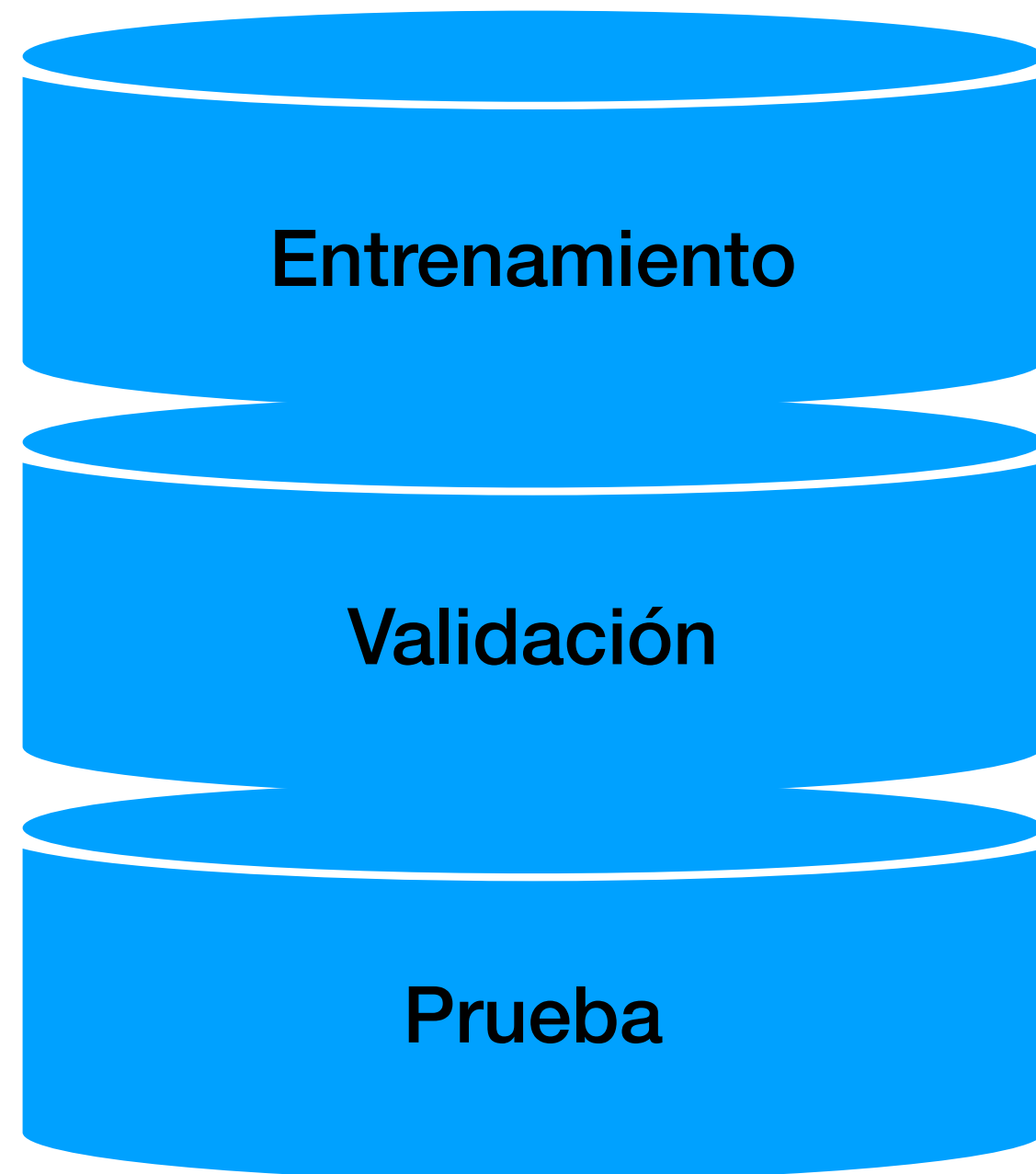
Se valida al terminar cada época (cada vez que se ajustan los pesos y los sesgos)



```
if (pérdida_entrenamiento va de la mano con la pérdida_validación):  
    print("no hay problema, continuar")  
elif (pérdida_validación va en aumento):  
    print("sobre-ajuste, sobre-ajuste, sobre-ajuste")
```

# Sobre-ajuste

## Datos poco voluminosos



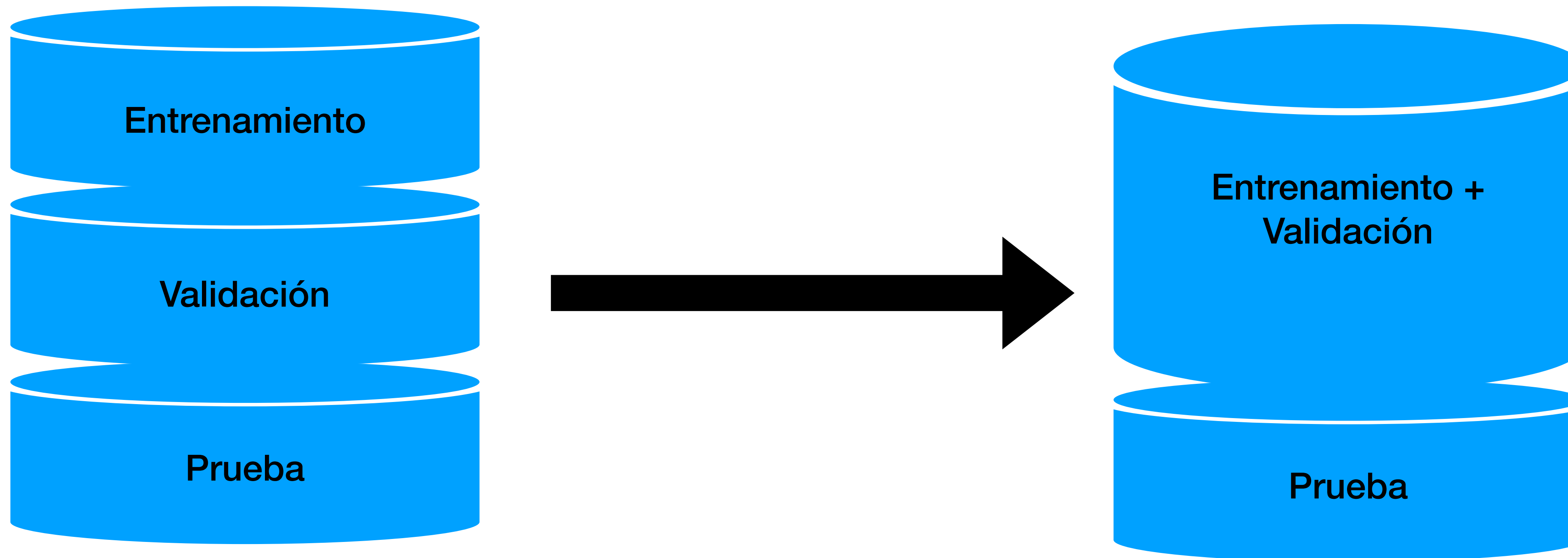
- No podemos darnos el lujo de dividir el conjunto de datos en 3
- Podrían ser tan pocos datos que el algoritmo no “aprende” nada

Solución:

**N-FOLD CROSS-VALIDATION**

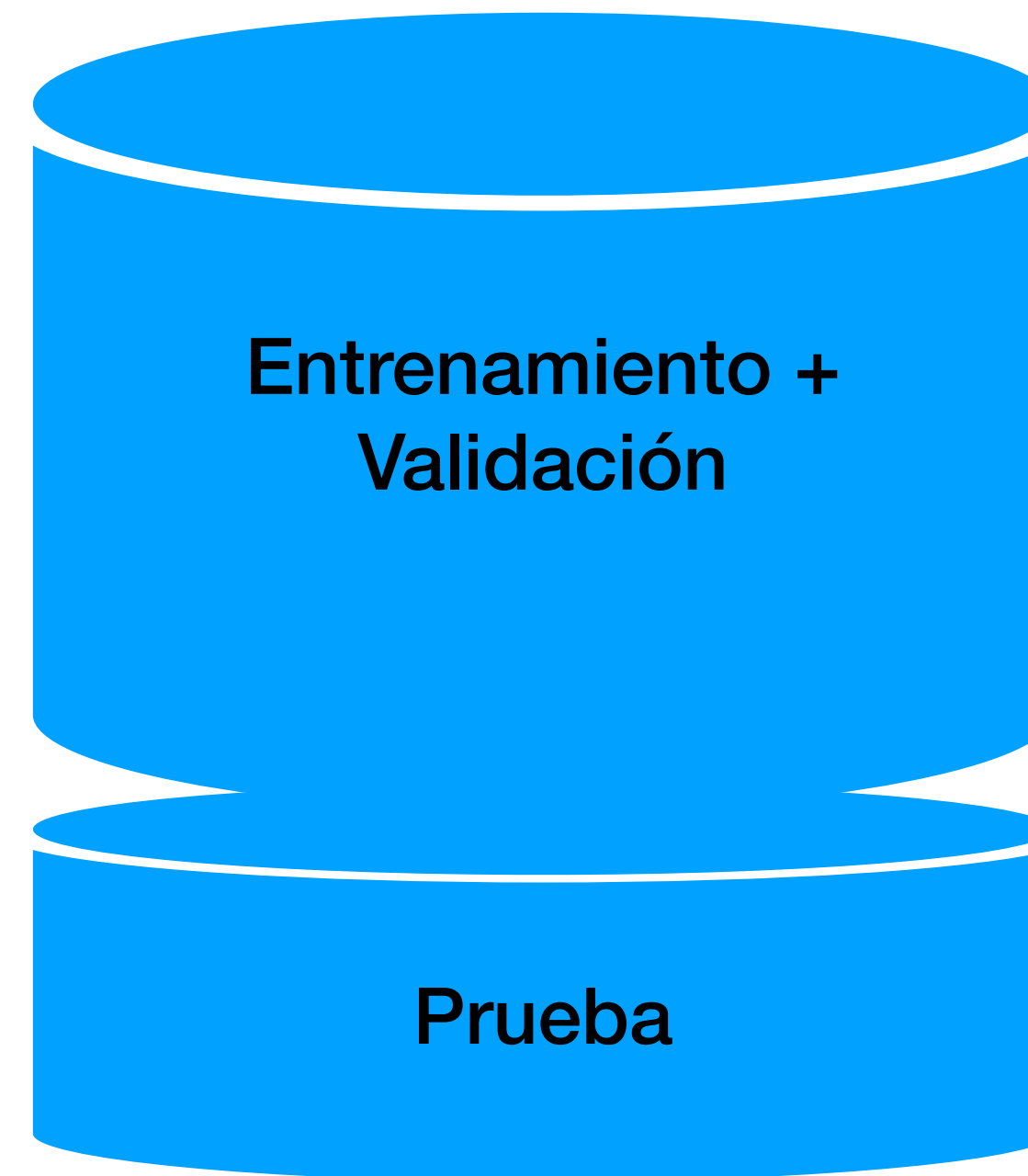
(Validación cruzada sobre N “dobletes”

# N-fold Cross-Validation



# N-fold Cross-Validation

Conjunto completo  
11,000 muestras



10,000 muestras

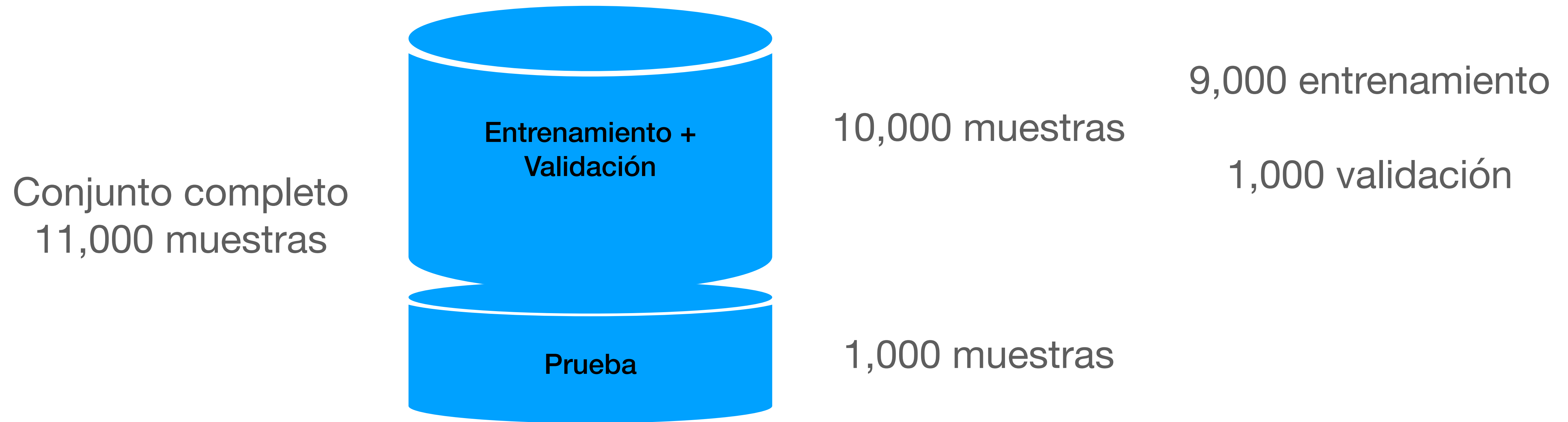
1,000 muestras



**Nótese que esto es un conjunto de datos pequeños. En ML se acostumbra trabajar con conjuntos de “sapotocientas” muestras**

**Los conjuntos de datos enormes suelen tener sus propios problemas. Por ejemplo: datos faltantes**

# N-fold Cross-Validation



# 10-fold Cross-Validation



9,000

1,000



Entrenamiento



Validación

Epoca 1



Epoca 2



Epoca 3



Epoca 4



Epoca 5



Epoca 6



Epoca 7



Epoca 8



Epoca 9



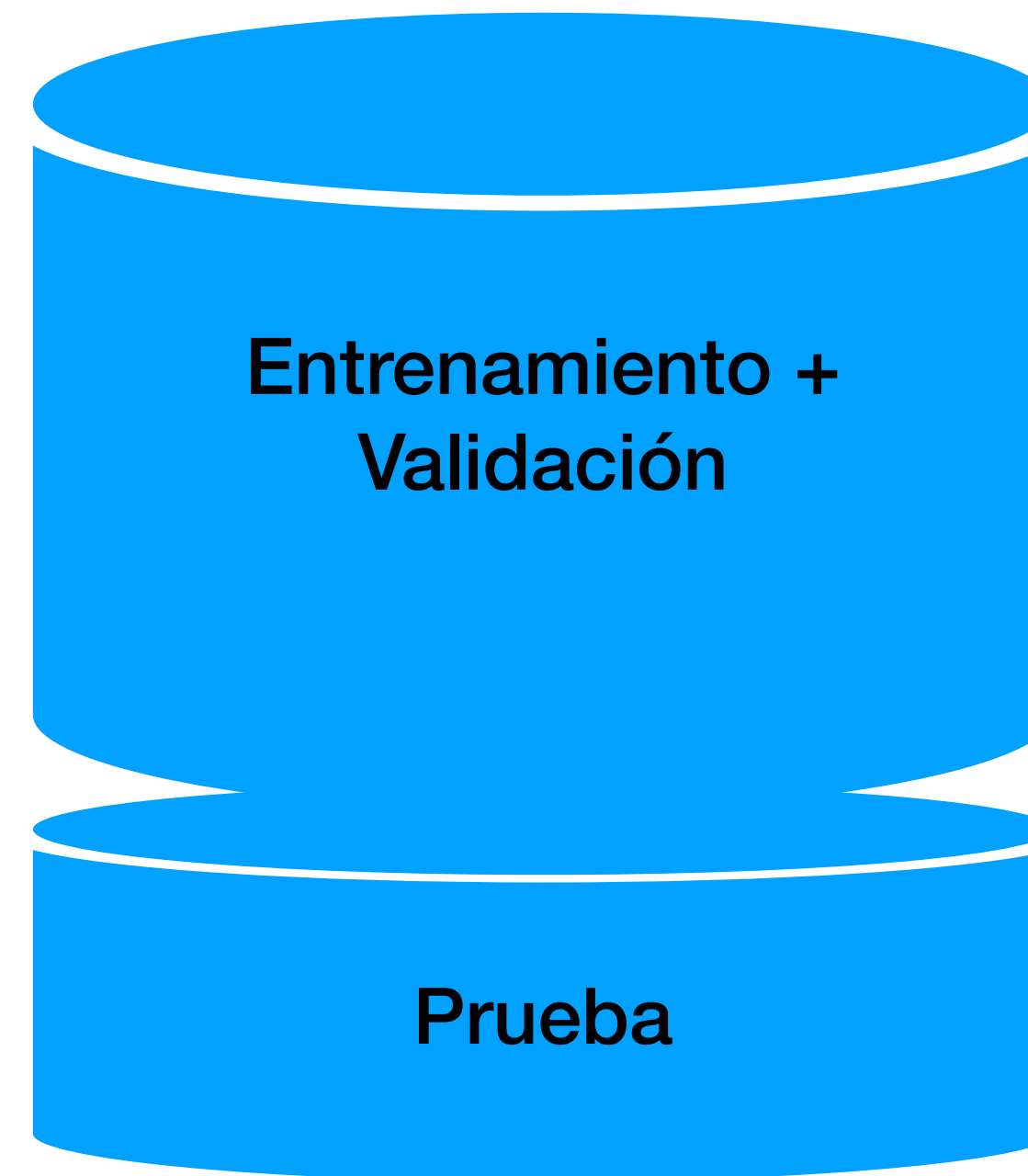
Epoca 10



# N-fold Cross-Validation

## Pros

- Aprovechamos más datos
- Tenemos un modelo

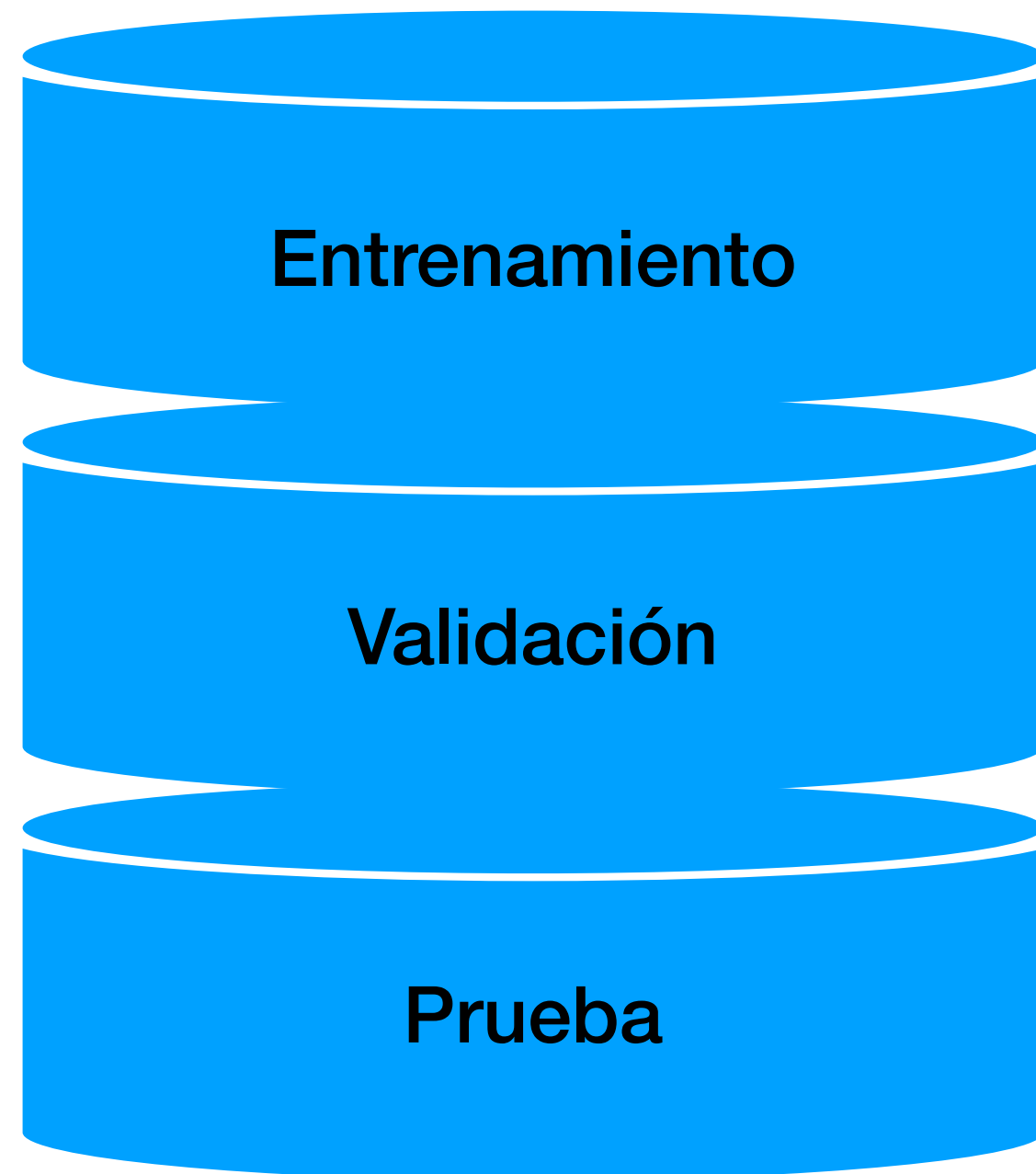


## Cons

- Se han utilizados los datos de validación para entrenar
- Posiblemente haya algo de sobre-ajuste



# Entrenamiento, validación y prueba



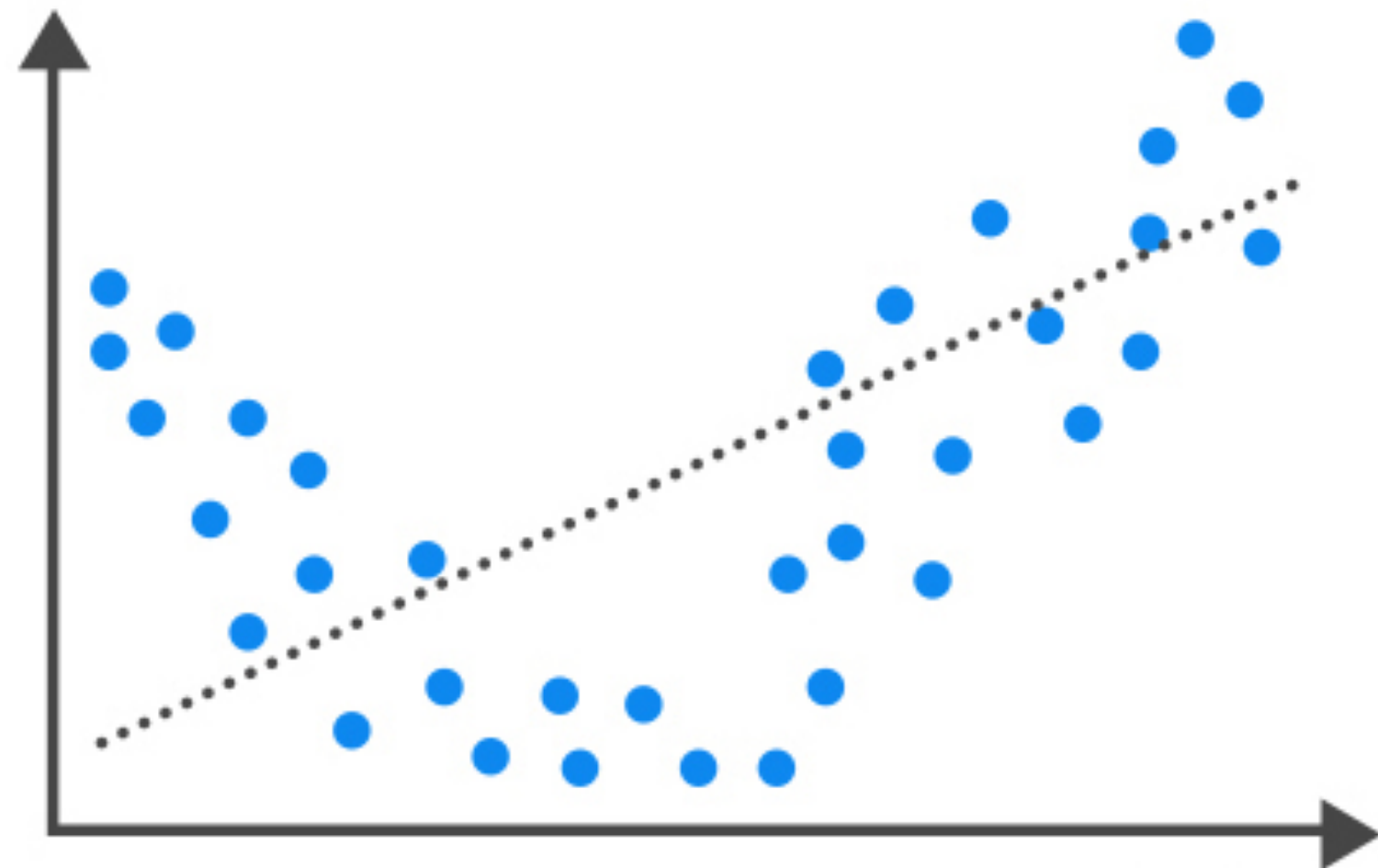
- Siempre debemos procurar hacer la división de 3
- Solo si con este modelo, no parece “aprender” bien, se debe acudir al método N-FOLD CROSS-VALIDATION

Terminación temprana ó...  
¿Cuándo dejar de entrenar?

# Hemos reiterado muchas veces que se entrena el modelo hasta que se minimice la función de pérdida

Pérdida alta (Loss)

Exactitud baja

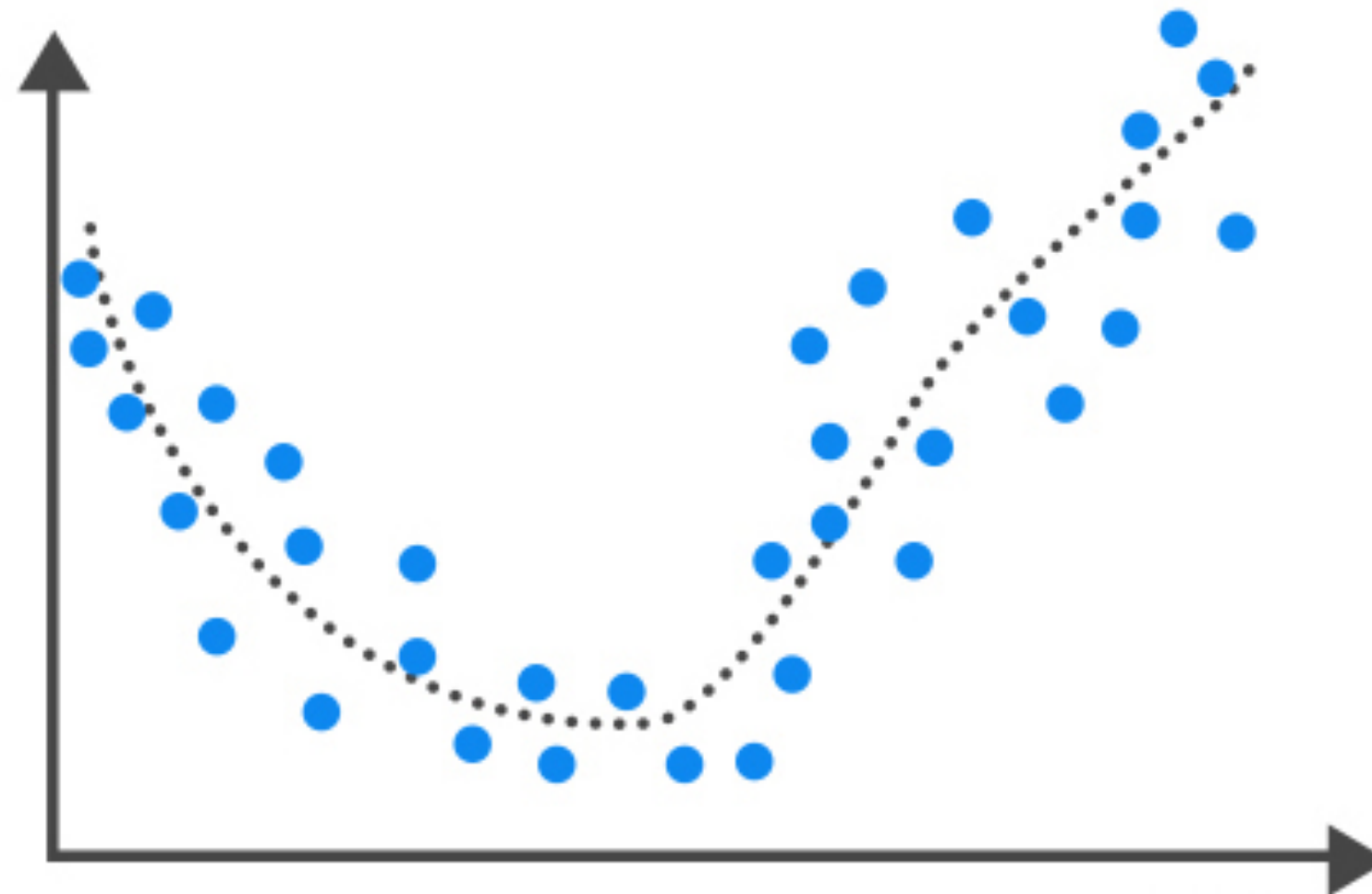


Sub-ajustado

(Alto error de sesgo)

Pérdida baja (Loss)

Exactitud alta

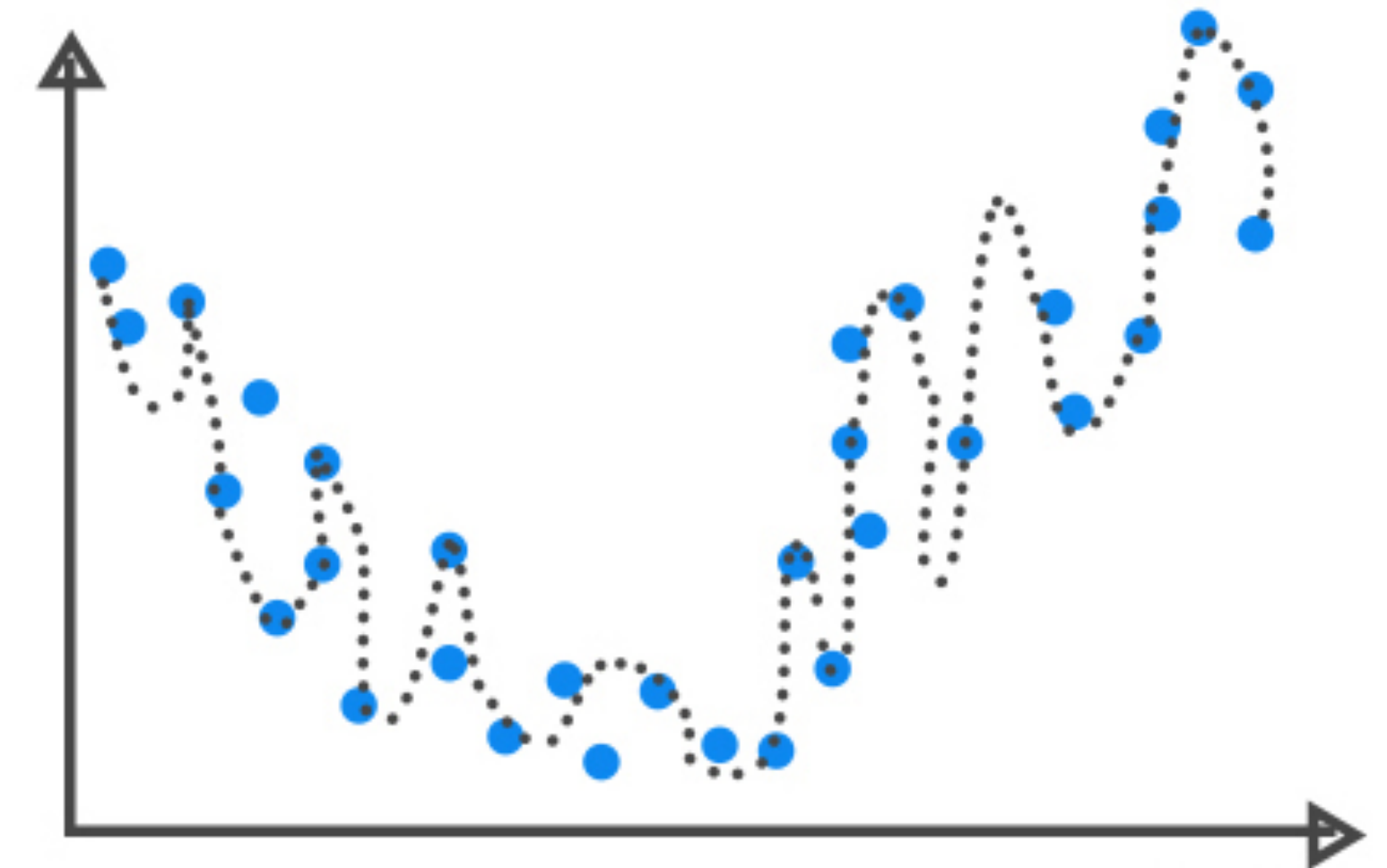


Buen ajuste/Robusto

(Balance entre sesgo y  
varianza)

Pérdida baja (Loss)

Exactitud baja



Sobre-ajustado

(Alto error de varianza)



# Terminación temprana

## Una técnica para evitar el sobre-ajuste

¿Cómo hacerlo?

1. Predefinir el número de épocas (es la forma más simple)

Pros:

1. Eventualmente, resuelve el problema

Cons:

1. No hay garantía que se llegue al mínimo
2. Posiblemente no lo logre y diverge al infinito

# Terminación temprana

## Una técnica para evitar el sobre-ajuste

Ya conocemos bastante más! Una técnica un poco más sofisticada es:

2. Detener el proceso cuando los cambios sean demasiado pequeños

Lo mencionamos hace algunas clases.  $X_{i+1} - X_i = 0.001$

Pros:

1. Aseguramos que la pérdida se minimice
2. Hay ahorro de potencia computacional al reducir las iteraciones

Cons:

# Terminación temprana

	Definir el No. De Epocas	Actualizaciones muy pequeñas	
Resuelve el problema	✓	✓	
Seguridad de que se minimiza la pérdida	✗	✓	
No hay iteraciones en balde	✗	✓	
Previene sobre-ajuste	✗	✗	

# Terminación temprana

## Una técnica para evitar el sobre-ajuste

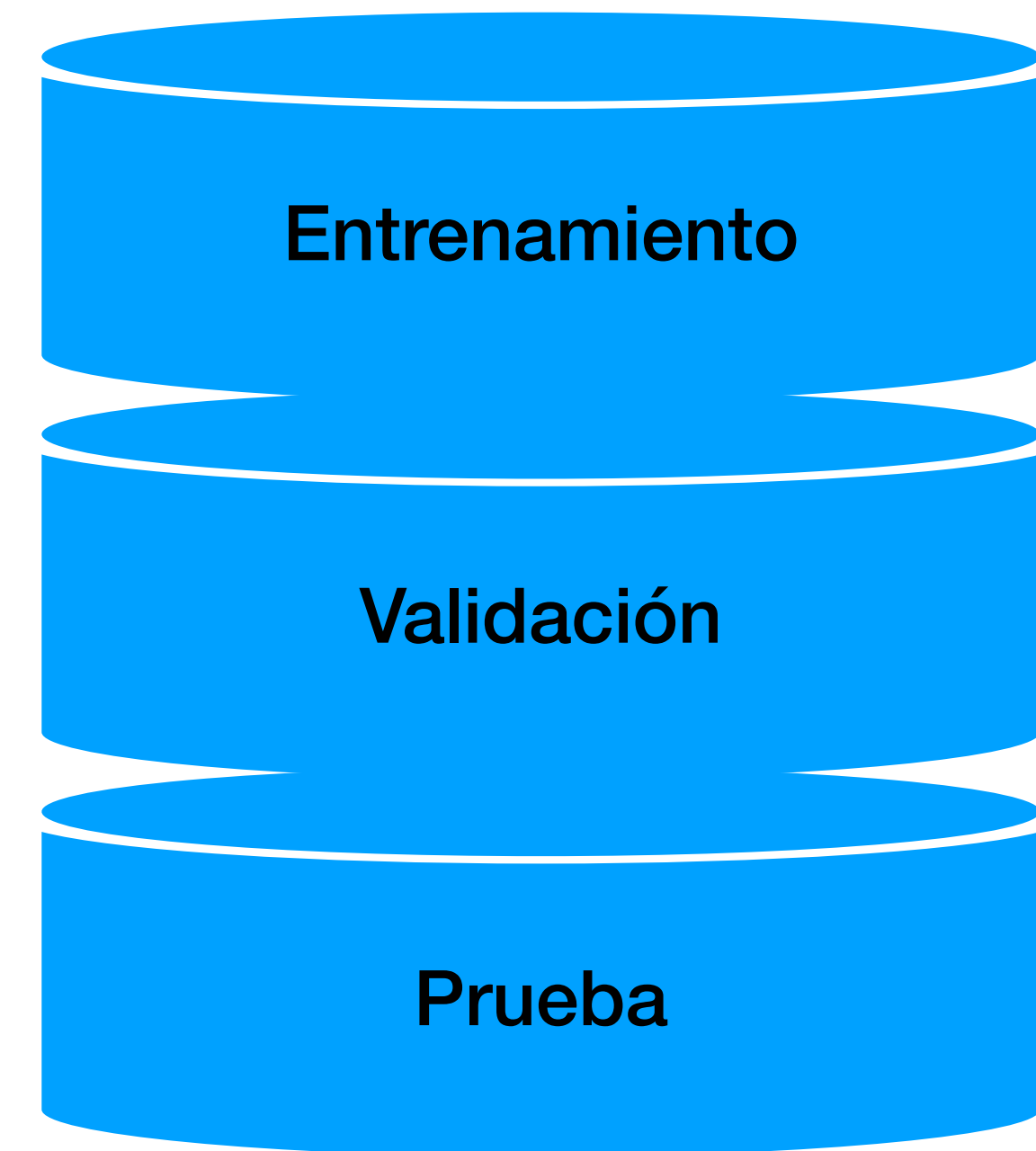
Necesitamos una técnica más avanzada:

### 3. La estrategia del conjunto de validación

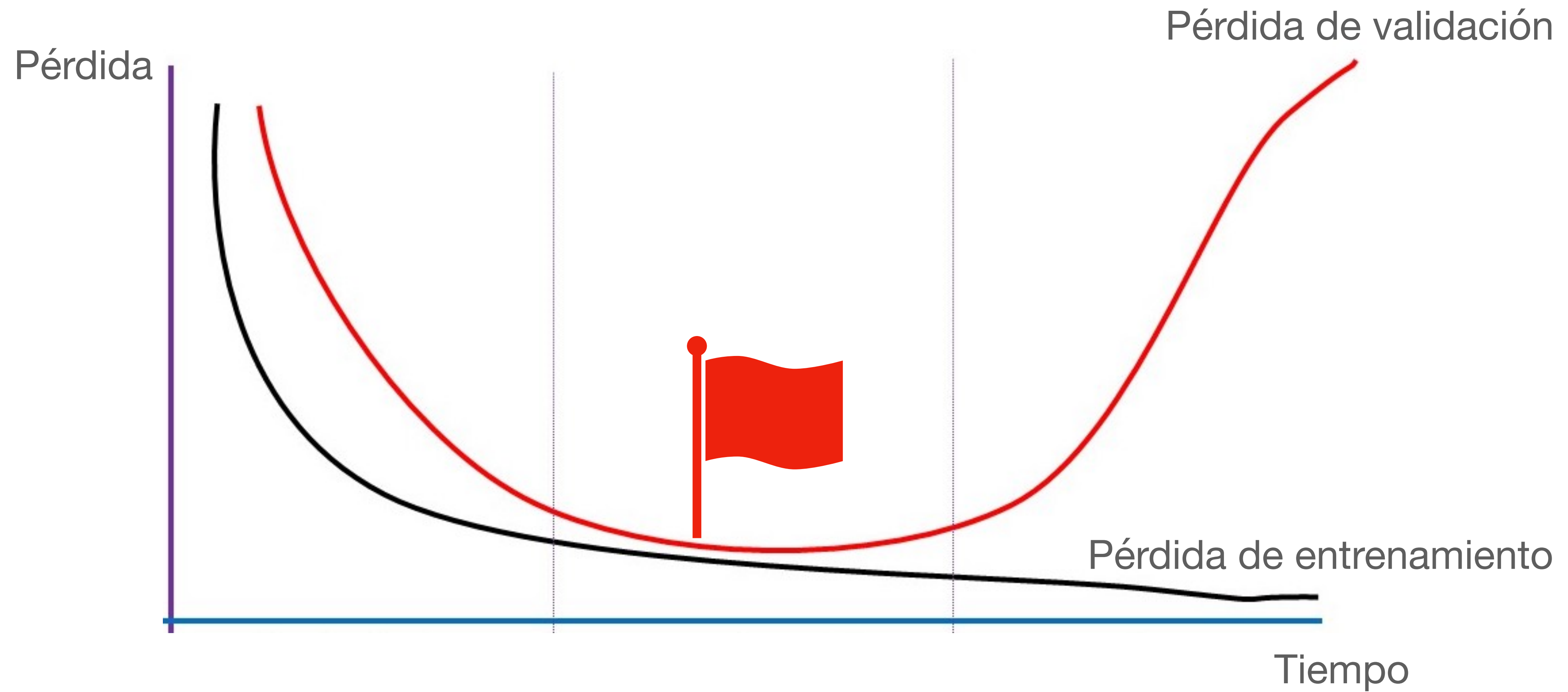
#### Pros:

1. Aseguramos que la pérdida se minimice
2. Hay ahorro de potencia computacional al reducir las iteraciones
3. Evita el sobre-ajuste

#### Cons:



# Terminación temprana



# Terminación temprana

	Definir el No. De Epocas	Actualizaciones muy pequeñas	Conjunto de validación
Resuelve el problema	✓	✓	✓
Seguridad de que se minimiza la pérdida	✗	✓	✓
No hay iteraciones en balde	✗	✓	✗
Previene sobre-ajuste	✗	✗	✓

Puede ser que el método del conjunto de validación esté presentando ya cambios muy pequeños de los pesos y sesgos y aún no esté sobre-ajustando

**Mejor práctica:**

**Detener el proceso cuando ocurra uno de los siguientes:**

- **La pérdida de validación empieza a incrementarse, ó**
- **La pérdida de entrenamiento se torna muy pequeña**



# Inicialización

**Inicialización es el proceso por medio del cual fijamos los valores iniciales de los pesos**

**Es importante porque una  
asignación de pesos no  
adecuados puede conducir a un  
modelo que no se puede  
optimizar.**

# Inicialización

Recuerdan cuando empezamos con nuestro ejemplo simple:

$$f(x) = 5x^2 + 3x - 4$$

Generamos los datos así:

$$\text{eta} = 0.01$$

i	$x_i$	$f'(x_i)$
69	-0.29700634598073716	0.02993654019262859
70	-0.29730571138266343	0.02694288617336582
71	-0.2975751402443971	0.02424859755602915
72	-0.2978176262199574	0.021823737800426013
73	-0.29803586359796164	0.019641364020383634
74	-0.2982322772381655	0.017677227618345448
75	-0.29840904951434893	0.015909504856510548
76	-0.29856814456291403	0.014318554370859715
77	-0.29871133010662265	0.012886698933773477
78	-0.2988401970959604	0.011598029040396085
79	-0.2989561773863643	0.01043822613635692
80	-0.2990605596477279	0.009394403522721362
81	-0.2991545036829551	0.008454963170449137
82	-0.2992390533146596	0.007609466853403912
83	-0.29931514798319364	0.006848520168063477
84	-0.29938363318487426	0.006163668151257351
85	-0.29944526986638687	0.005547301336131127
86	-0.29950074287974815	0.004992571202518725

# Inicialización

Arbitrariamente seleccionamos 4 como valor de  $x_0$ :

```
[2]: x = [4]
     f_prima = [43]
     eta = 0.01
     print(f"{0}    {x[0]}    {f_prima[0]}")
```

0 4 43

# Inicialización

Cuando vimos nuestra primera red neuronal, también inicializamos:

```
[44]: rango_inicial = 0.1

weights = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=(2, 1))

biases = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=1)

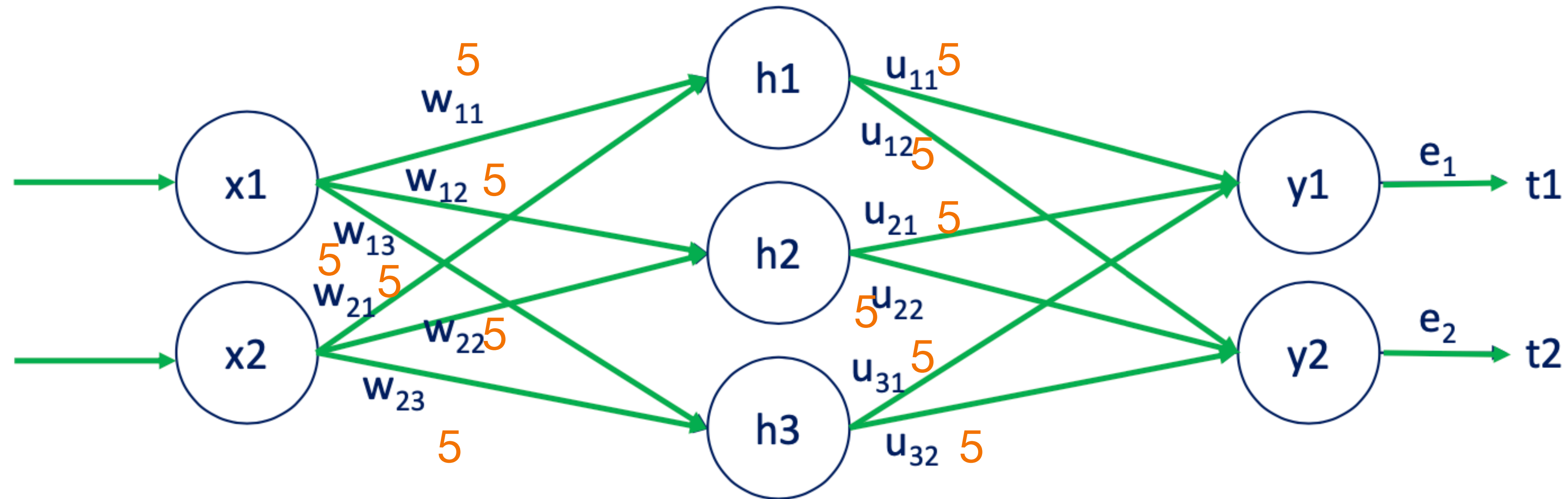
#Veamos cómo fueron inicializados.
print (weights)
print (biases)
```

```
[[ 0.09369914]
 [-0.08781476]]
[-0.06697437]
```



**¿Realmente importa cuáles son los valores iniciales de los pesos?**

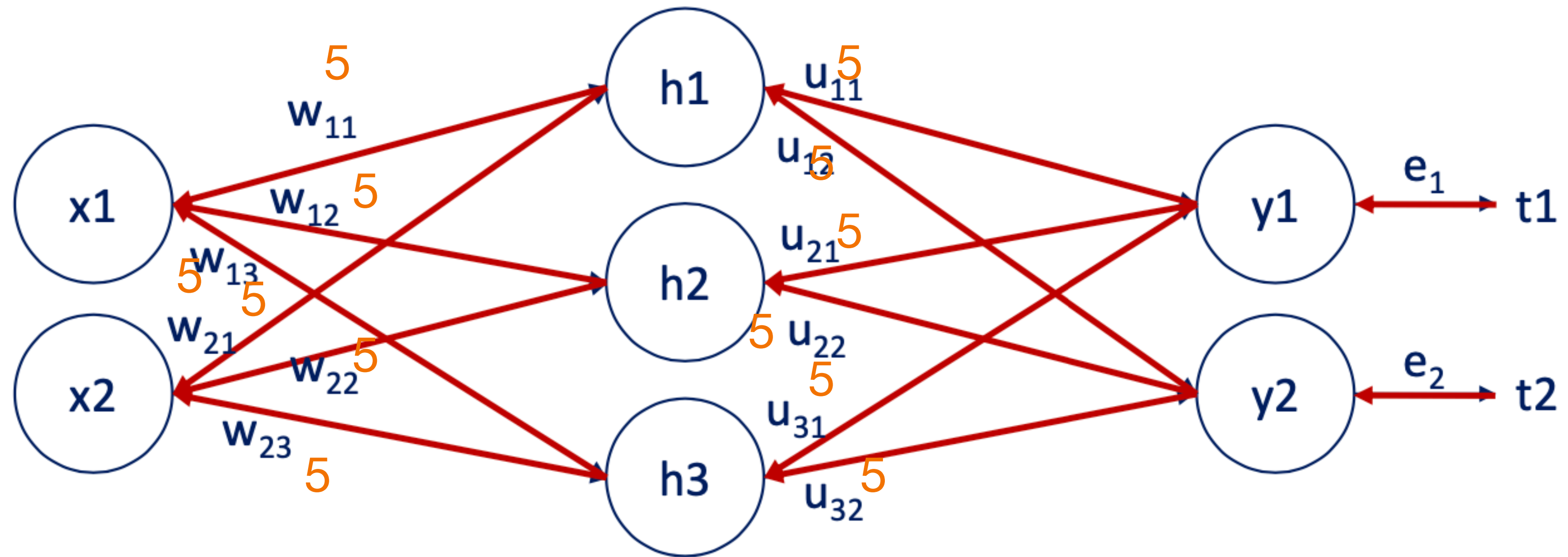
# Inicialización



¿Que pasa si asignamos una constante a todos los pesos?

Como es una red totalmente simétrica, en la propagación hacia adelante, no hay razón alguna para que cada unidad de la capa escondida aprenda algo diferente.

# Inicialización



En la propagación hacia atrás, tampoco...aunque si habría un poco de optimización por los errores introducidos.

Los pesos no serían de utilidad!

# Tipos de inicializaciones simples

## Opción 1

- Inicializar dentro de un rango pequeño

```
[44]: rango_inicial = 0.1

weights = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=(2, 1))

biases = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=1)

#Veamos cómo fueron inicializados.
print (weights)
print (biases)

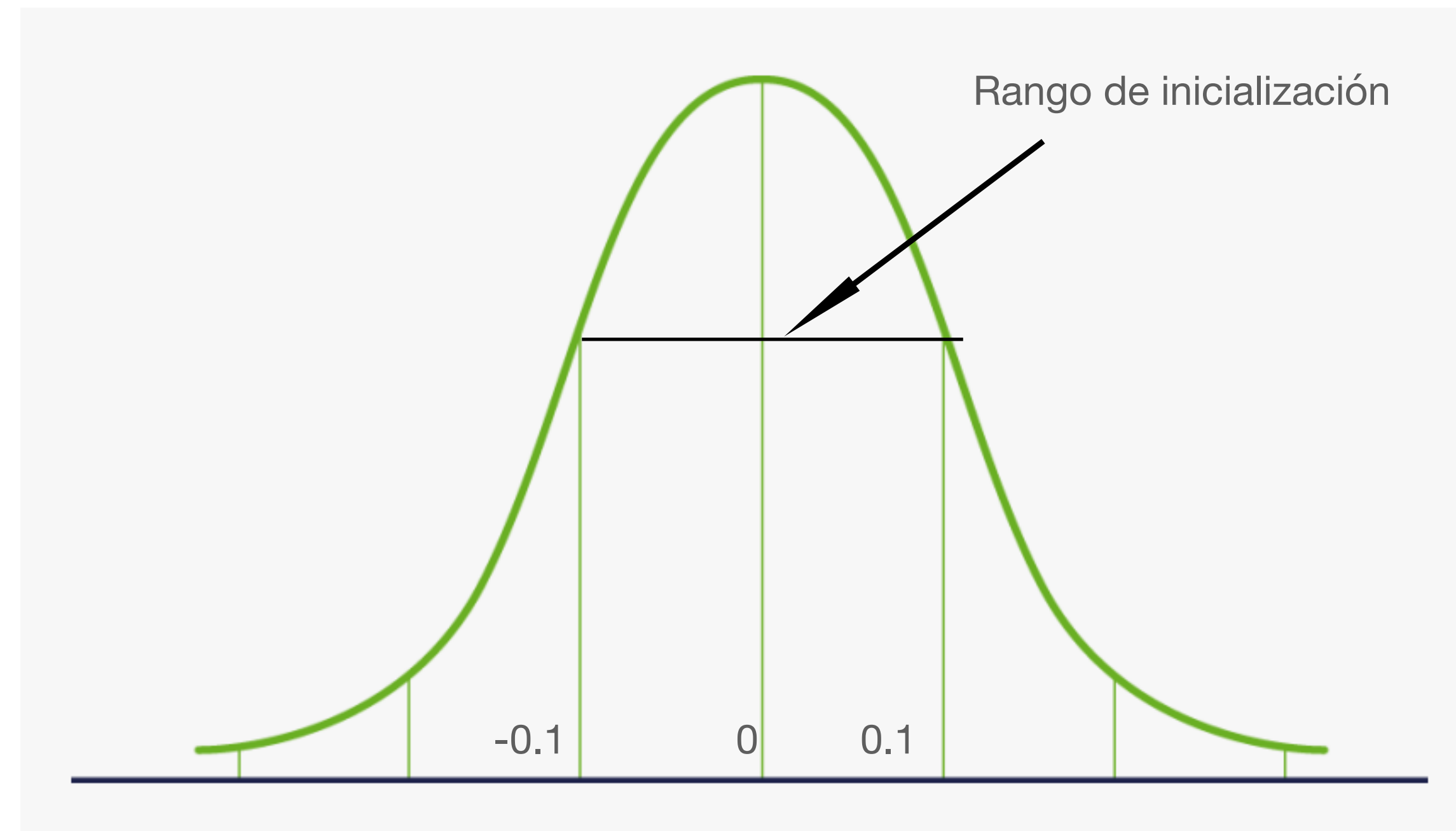
[[ 0.09369914]
 [-0.08781476]]
[-0.06697437]
```

- Se escogen los valores al azar pero en una forma uniforme (todos los valores tienen la misma probabilidad de salir)

# Tipos de inicializaciones simples

## Opción 2

- Inicializar dentro de un rango pequeño



- Se escogen los valores al azar pero dentro de una distribución normal (los valores cercanos a cero tienen más probabilidad de salir)

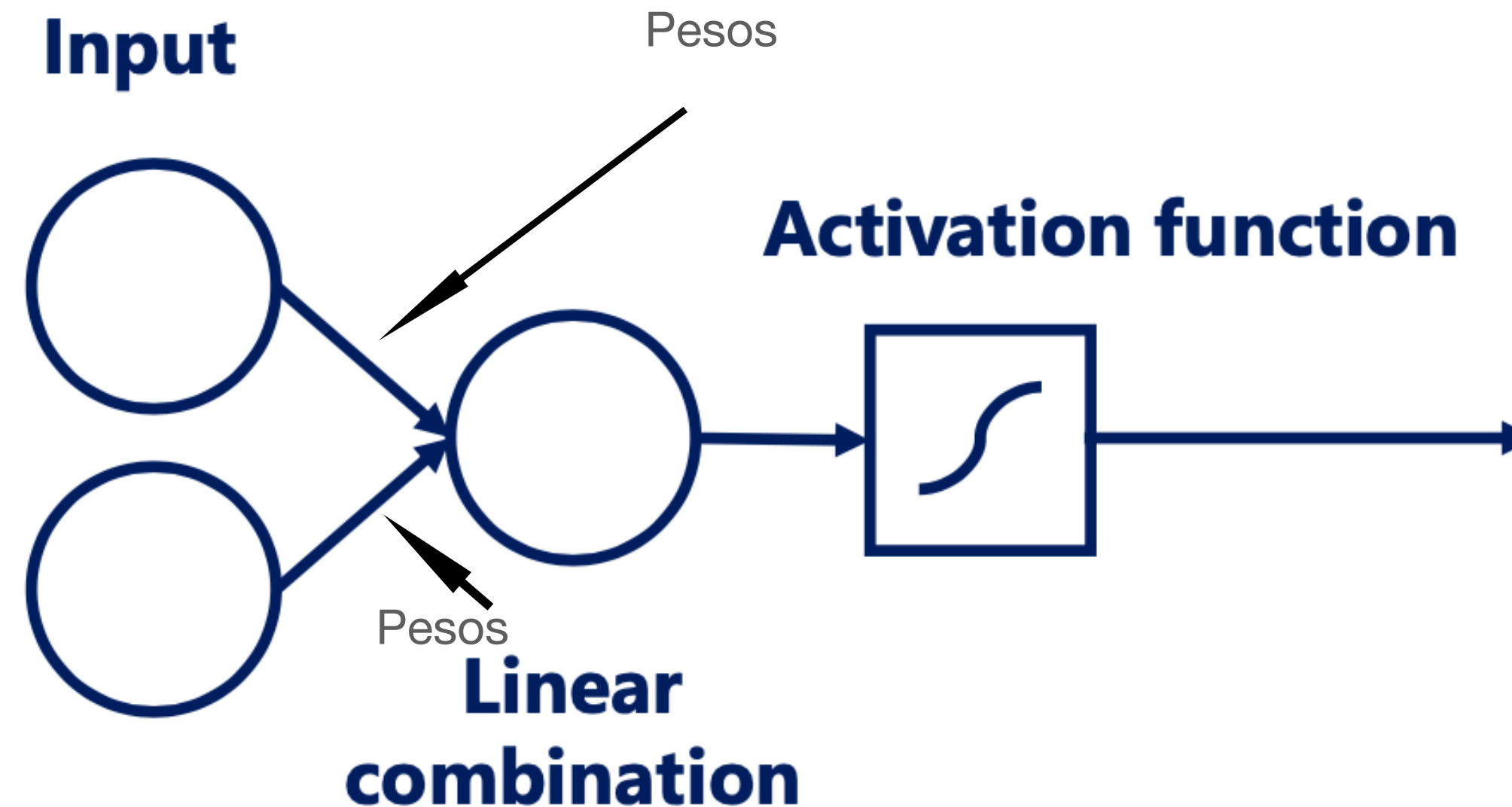
**Hasta el 2010 estas fueron las dos  
opciones que todo mundo utilizaba**



# Tipos de inicializaciones simples

## El problema

- Se utiliza la función sigmoide como función de activación

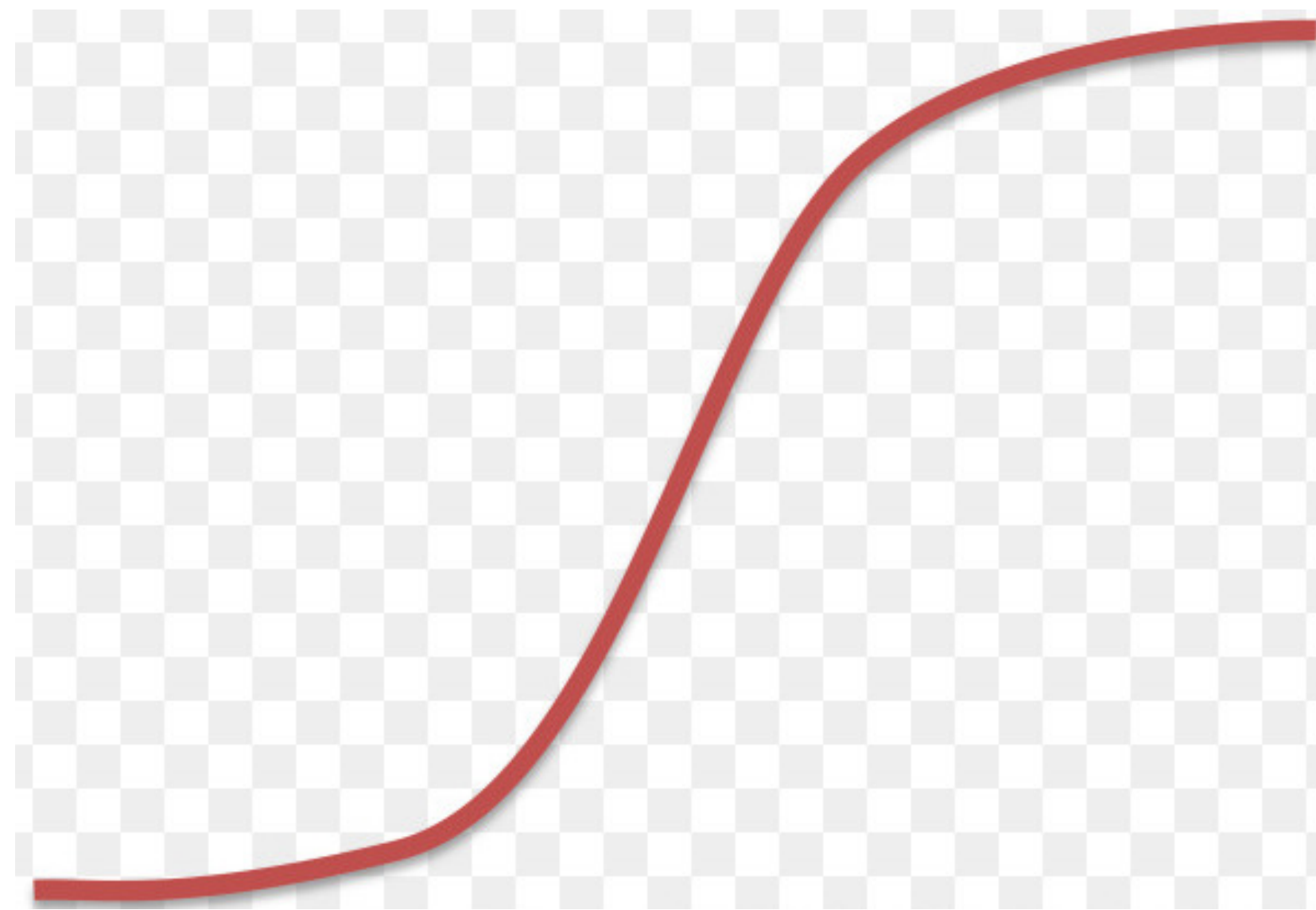
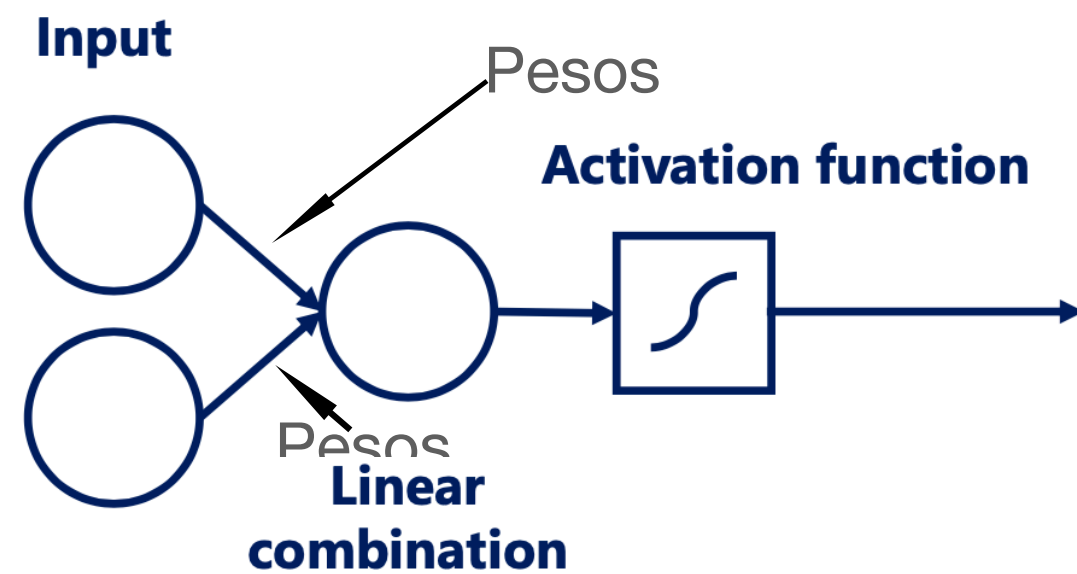


- El sigmoide es algo peculiar ya en su centro y en los extremos prácticamente es lineal...de nuevo caemos en un proceso totalmente lineal!

# Tipos de inicializaciones simples

## El problema

- La entrada al sigmoide son los pesos



- Si los pesos son muy pequeños, o muy grandes, no ocurre una transformación lineal

# Tipos de inicializaciones simples

## Opción 3 - Inicialización Xavier ó Inicialización Glorot

En el 2010 se publicó un artículo que cambió totalmente la forma de inicializar

**Understanding the difficulty of training deep feedforward neural networks**

***Xavier Glorot, Yosua Bengio; Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010***

# Tipos de inicializaciones simples

## Opción 3 - Inicialización Xavier ó Inicialización Glorot

**Inicialización Xavier:** obtener, al azar, cada peso de una distribución:

$$\text{Uniform distribution } \left(-\sqrt{\frac{6}{n_i + n_o}}, \sqrt{\frac{6}{n_i + n_o}}\right)$$

$$\text{Normal Distribution with } \mu = 0 \text{ and } \sigma = \sqrt{\frac{2}{n_i + n_o}}$$

$n_i$  - Número de entradas

$n_o$  - Número de salidas

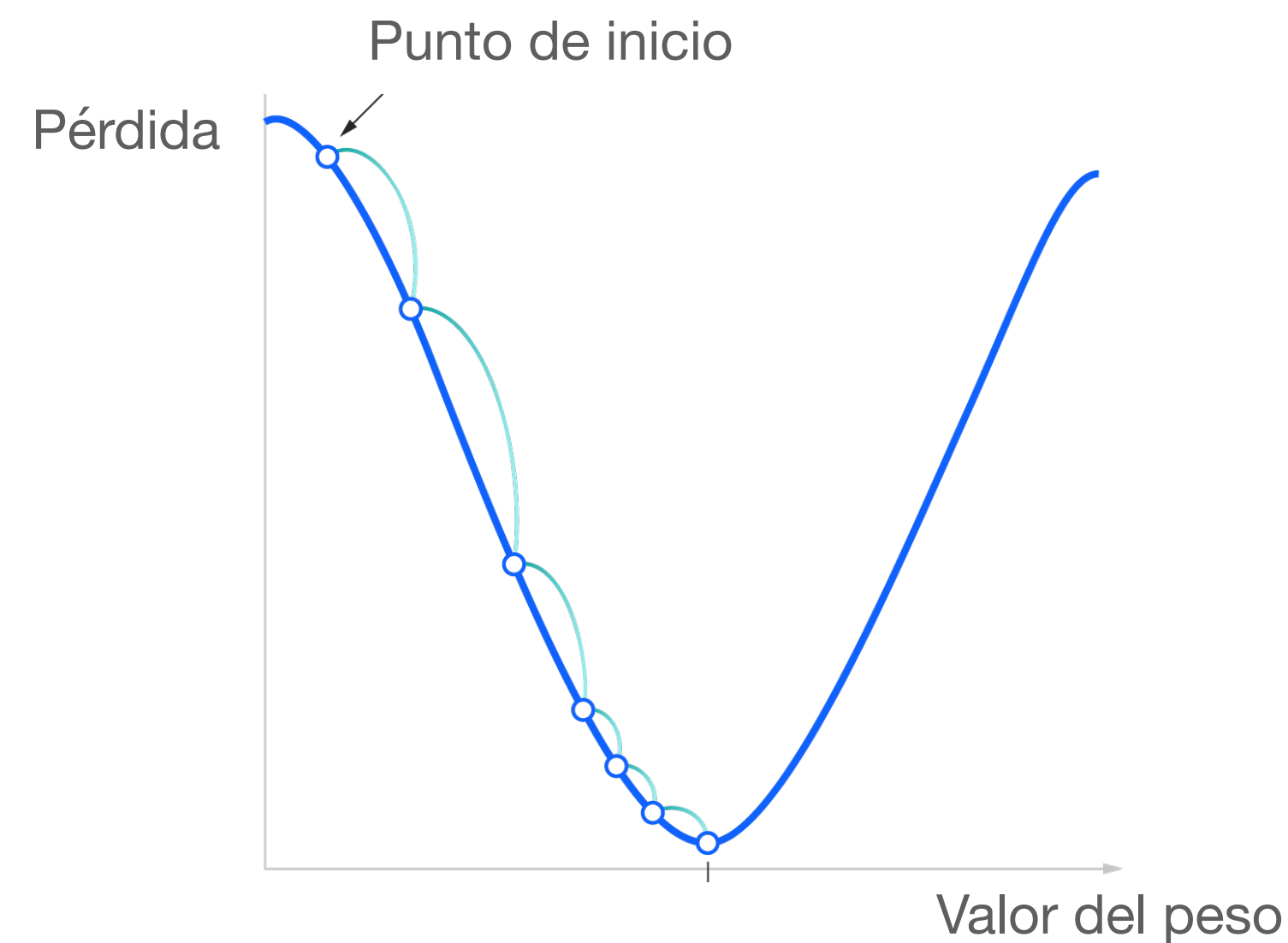
**En TensorFlow, si no se especifica el inicializador de variables, el default es:**

**glorot\_uniform\_initializer**

# Profundización sobre el método de Descenso de Gradiente (Gradient Descent)

# Optimización

- Algoritmos que usaremos para variar los parámetros de nuestro modelo
- Hasta ahora solamente hemos visto el algoritmo de Descenso de Gradiente



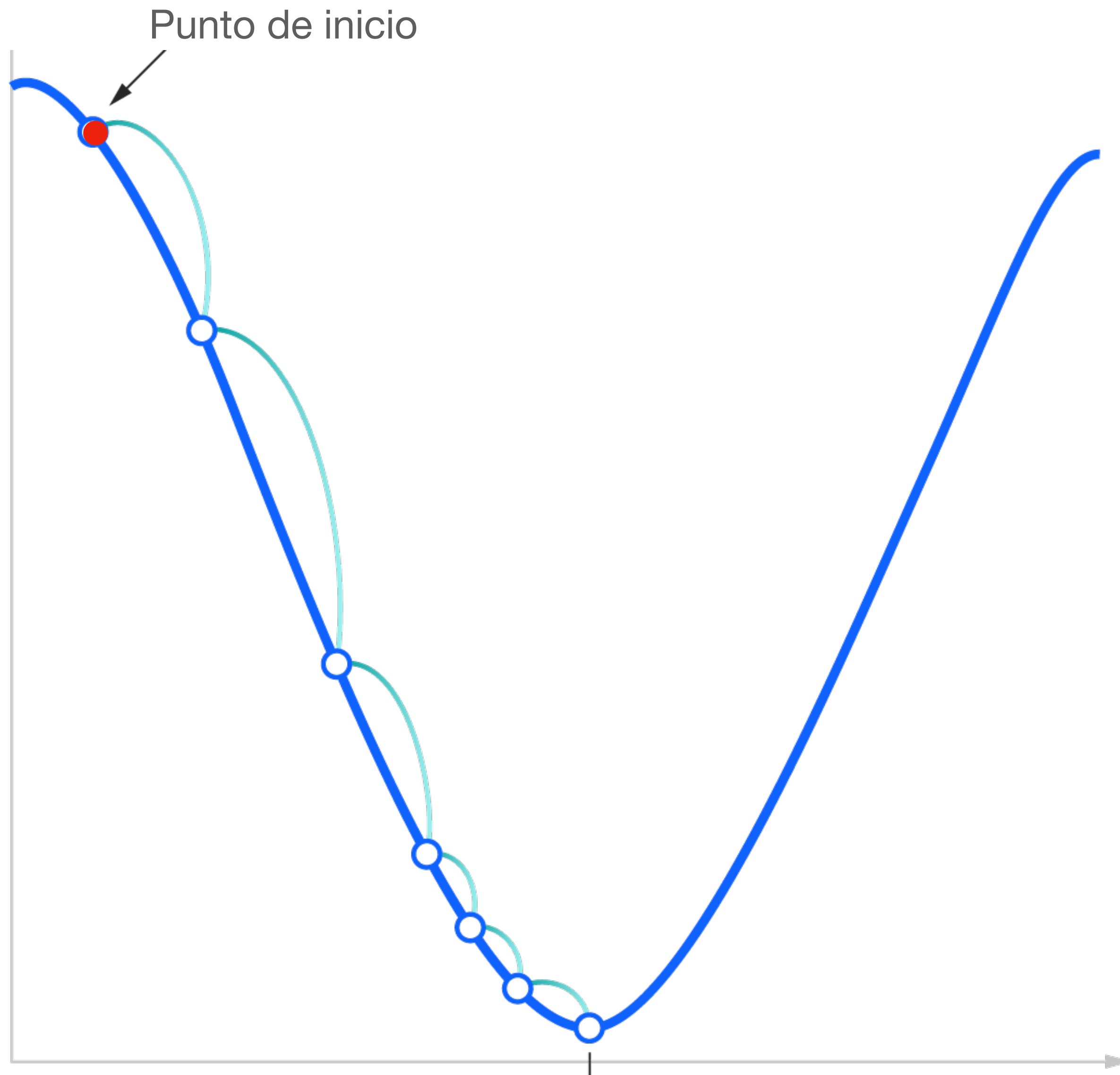
- Hablaremos de ajustes para obtener algoritmos mejorados



# Optimización

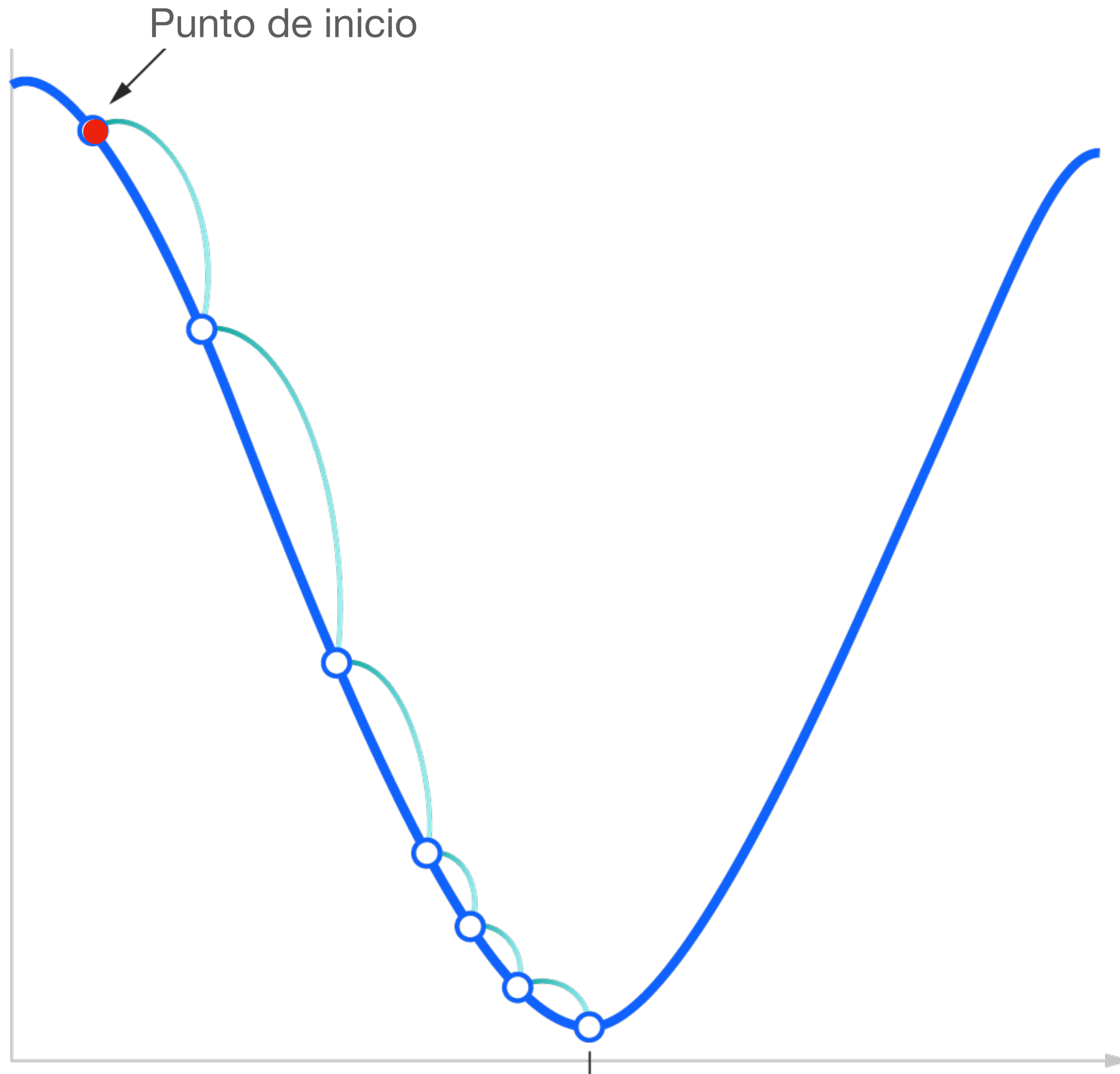
- La mayor parte de lo que hemos visto es invalorable desde el punto de vista teórico
- Sin embargo, desde el punto de vista de ejecución, no es práctico
- Con unos cuantos ajustes, podremos darle vuelta a esta situación

# Descenso de gradiente



- Hace iteraciones sobre la totalidad del conjunto de datos antes de modificar el peso
- Cada actualización es muy pequeño
- Resulta en:
  - Muchas épocas
  - Sobre todo el conjunto de datos
  - Usando una baja tasa de aprendizaje

# Descenso de gradiente estocástico (SGD)



- Actualizar el valor muchas veces dentro de cada época
- Es un método relacionado al concepto de tandas “batching”
- Proceso de dividir los datos en un conjunto de  $n$  tandas

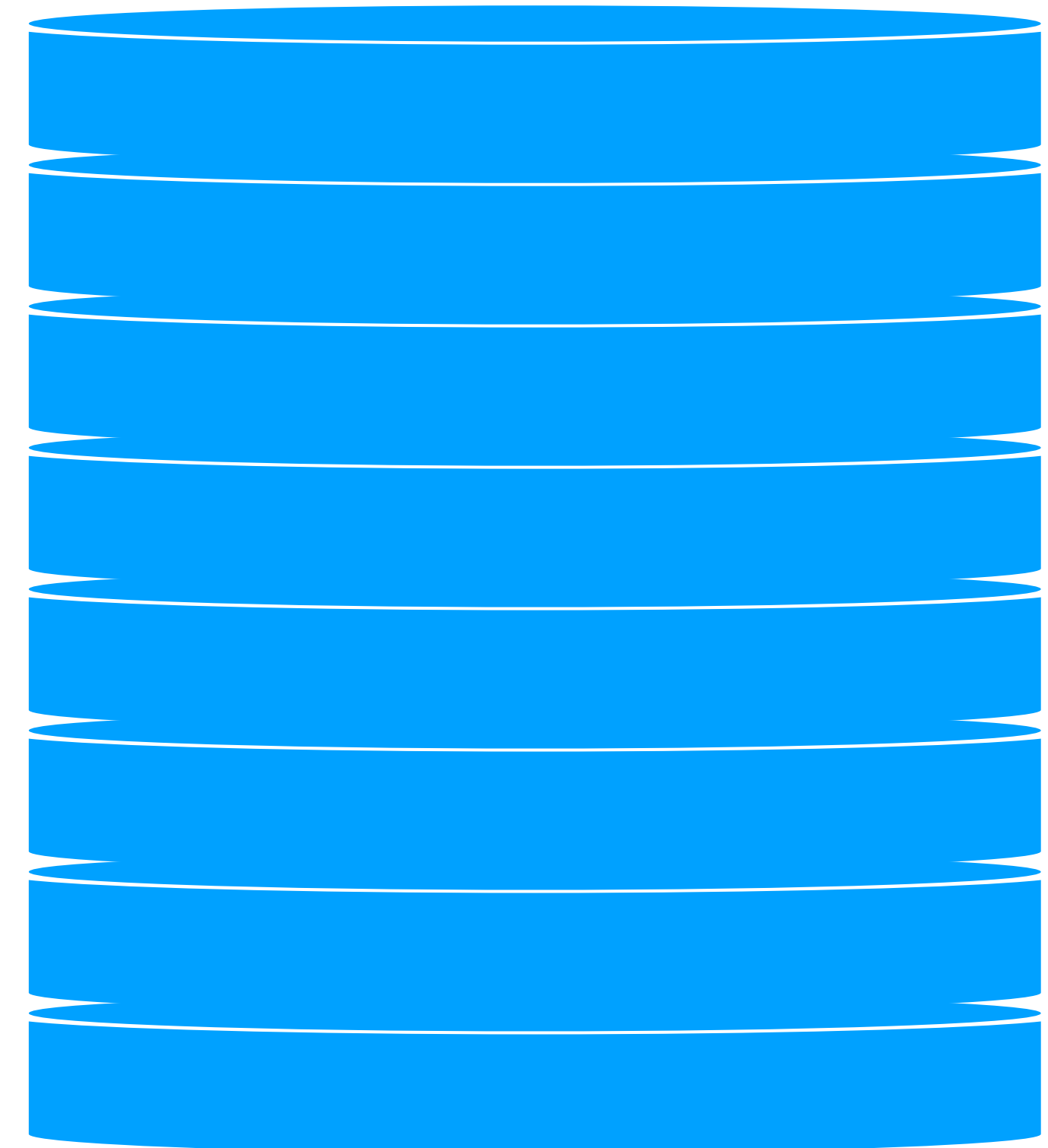
# Tandas o “batching”

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & & & & \\ a_{31} & & & a_{ij} & \\ \vdots & & & & \\ a_{m1} & & & & a_{mn} \end{bmatrix}$$

Si  $m = 10,000$  y escogemos tandas de 1,000, entonces tendremos 10 tandas por época

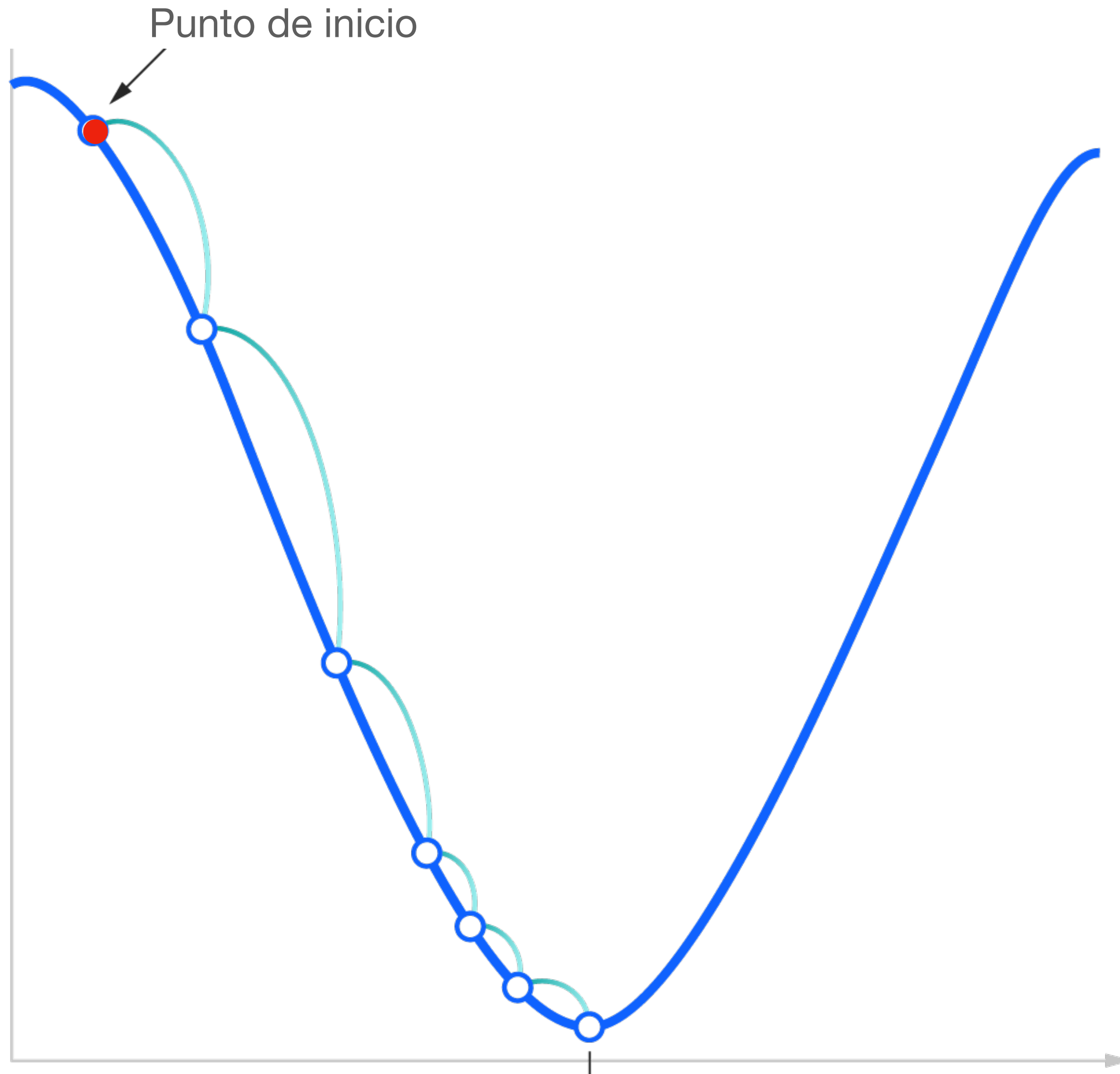
Para cada iteración completa del conjunto de datos, los pesos se actualizarían 10 veces

Tandas



Se actualizan los pesos después de cada tanda

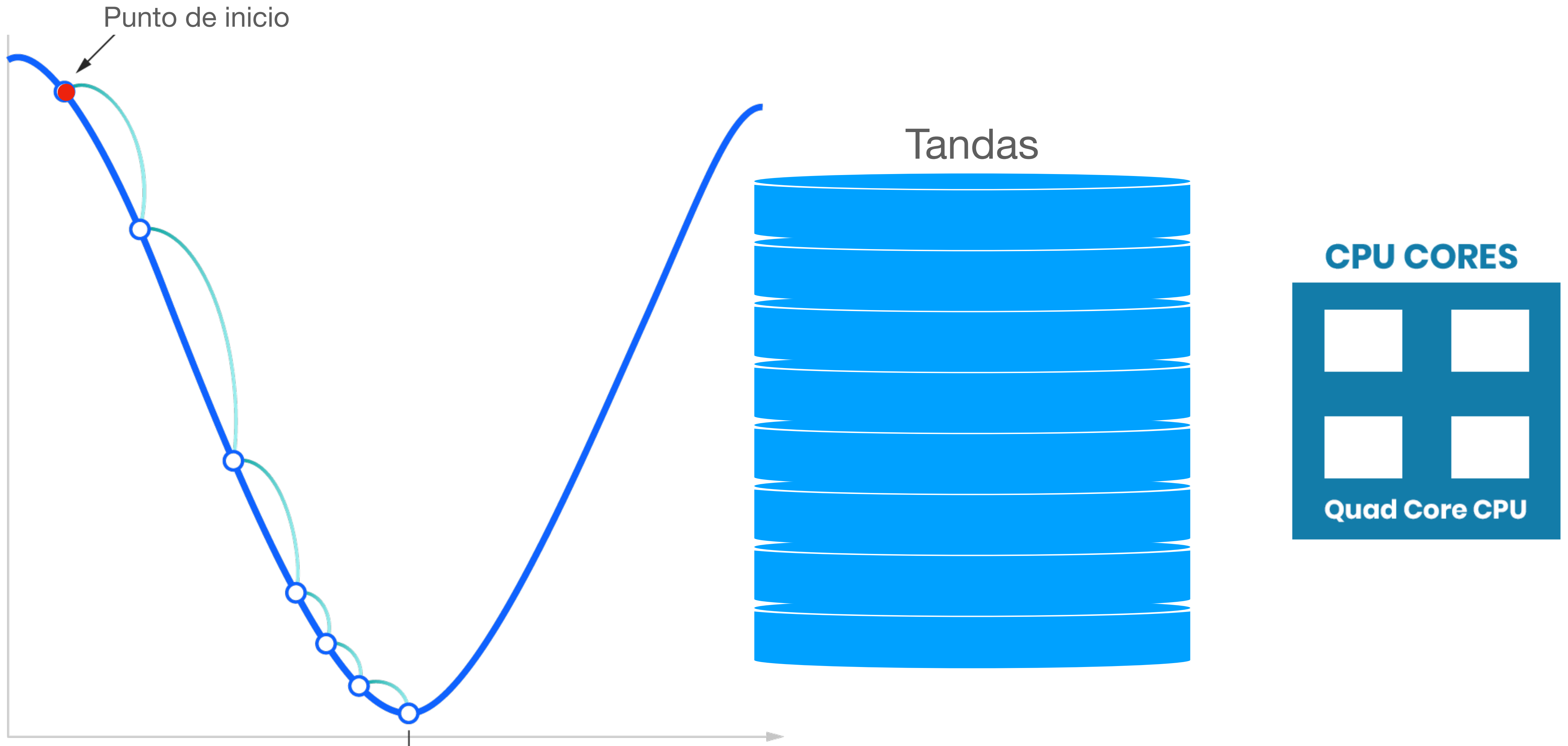
# Descenso de gradiente estocástico (SGD)



- No hay nada nuevo en el algoritmo, pero es mucho más rápido
- Pero.... Tiende a aproximar más

Perdemos algo de exactitud, pero vale la pena a cambio de la mayor velocidad!

# Tandas ó “batching - vale la pena?



# Problemas con el descenso de gradiente

En la vida real, las funciones de pérdida no son tan regulares



Para evitar caer en el primer mínimo, podríamos incrementar la tasa de cambio. Sin embargo, como ya hemos visto podríamos caer en oscilación,



# Momento

## Una solución al problema



La regla que tenemos para el descenso de gradientes es:

$$w'_1 = w_1 - \eta \frac{\partial L}{\partial w_1}$$

La mejor forma de saber cuál es el momento, es ver cuál era el momento en la medición anterior:

$$w \leftarrow w(t) - \underbrace{\eta \frac{\partial L}{\partial w}(t)}_{\text{Actualización actual}} - \underbrace{\eta \frac{\partial L}{\partial w}(t-1)}_{\text{Actualización anterior}}$$

Actualización actual

Actualización anterior

# Momento

## Una solución al problema



Para no darle la misma importancia a las dos actualizaciones, se usa un coeficiente

$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \alpha \eta \frac{\partial L}{\partial w}(t-1)$$

Generalmente se usa un valor de:

$$\alpha = 0.9$$

$\alpha$  es un hiperparámetro

# Agenda de la tasa de aprendizaje

## Hiperparámetros

Fijados por nosotros

Ancho

Profundidad

Tasa de aprendizaje ( $\eta$ )

Coeficiente momento ( $\alpha$ )

vrs

## Parámetros

encontrados por optimización

Pesos ( $w$ )

Sesgos ( $b$ )

# Tasa de aprendizaje ( $\eta$ )

- **Suficientemente pequeño** para poder descender suavemente, en vez de oscilar o divergen a infinito
- **Suficientemente grande** para alcanzarlo en un tiempo racional

Pero acá estamos haciendo Ciencia de Datos! Estos términos son demasiado vagos. Una forma inteligente para hacerlo más objetivo es adoptar una **Agenda de Tasa de Aprendizaje**

# Agenda de tasa de aprendizaje ( $\eta$ )

Obtenemos lo mejor de los dos: **Suficientemente pequeño y Suficientemente grande**

Lo hacemos así:

1. Empezamos con una tasa de aprendizaje alta
2. En algún momento bajamos la tasa para evitar la oscilación
3. Cerca del final escogemos una tasa muy pequeña para llegar a una respuesta precisa



# Agenda de tasa de aprendizaje ( $\eta$ )

## Opción 1 de implementación

1. Empezamos con una tasa de aprendizaje alta	Primeras 5 épocas $\eta = 0.1$
2. En algún momento bajamos la tasa para evitar la oscilación	Siguientes 5 épocas $\eta = 0.01$
3. Cerca del final escogemos una tasa muy pequeña para llegar a una respuesta precisa	Hasta el final $\eta = 0.001$

**La opción anterior es demasiado  
simplísima y, además, requiere  
que nosotros establezcamos los  
valores**



# Agenda de tasa de aprendizaje ( $\eta$ )

## Opción 2 de implementación

1. Empezamos con una tasa de aprendizaje alta

2. En algún momento bajamos la tasa para evitar la oscilación

3. Cerca del final escogemos una tasa muy pequeña para llegar a una respuesta precisa

$$\eta_0 = 0.1$$

$n$  - la época de turno

$$\eta = \eta_0 e^{-n/c}$$

$c$  - alguna constante

# Agenda de tasa de aprendizaje ( $\eta$ )

## Ejemplo para $c = 20$

1. Empezamos con una tasa de aprendizaje alta

2. En algún momento bajamos la tasa para evitar la oscilación

3. Cerca del final escogemos una tasa muy pequeña para llegar a una respuesta precisa

$$\eta_0 = 0.1$$

$$\eta_1 = 0.0967$$

$$\eta_2 = 0.0905$$

$$\eta_3 = 0.0819$$

$$\eta_4 = 0.0717$$

$$\eta_5 = 0.0607$$

$$\eta_6 = 0.0497$$

$$\eta_7 = 0.0393$$

$$\eta_8 = 0.0301$$

$n$  - la época de turno

$$\eta = \eta_0 e^{-n/20}$$

$c$  - alguna constante

# Agenda de tasa de aprendizaje ( $\eta$ )

$$\eta_0 = 0.1$$

$$\eta_1 = 0.0967$$

$$\eta_2 = 0.0905$$

$$\eta_3 = 0.0819$$

$$\eta_4 = 0.0717$$

$$\eta_5 = 0.0607$$

$$\eta_6 = 0.0497$$

$$\eta_7 = 0.0393$$

$$\eta_8 = 0.0301$$

$n$  - la época de turno

$$\eta = \eta_0 e^{-n/20}$$

$c$  - alguna constante

No hay una regla fija para seleccionar  $c$ , pero debe ser del mismo orden de magnitud de las épocas

e.g. si necesitamos:

100 épocas,  $50 < c < 500$

1000 épocas,  $500 < c < 5000$

Generalmente necesitamos menos épocas y un valor de

$c \sim 20$

es bueno

**La experiencia muestra que el valor exacto de  $c$  no es importante. Lo que es importante es que haya una agenda de aprendizaje**

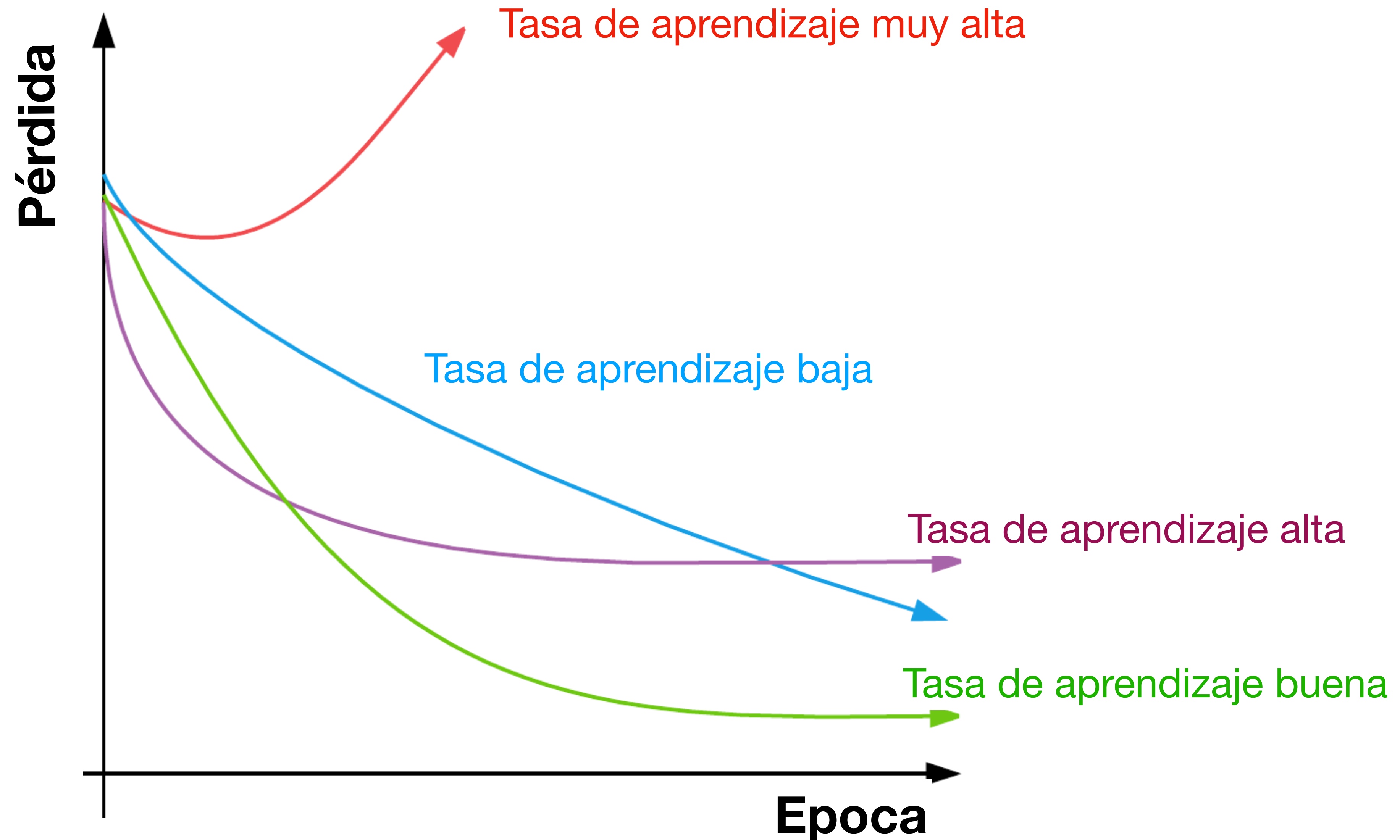
# Agenda de la tasa de aprendizaje

Hiperparámetros	vrs	Parámetros
Fijados por nosotros		encontrados por optimización
Ancho		
Profundidad		Pesos ( $w$ )
Tasa de aprendizaje ( $\eta$ )		
Tamaño de las tandas		Sesgos ( $b$ )
Coeficiente momento ( $\alpha$ )		
Coeficiente de decaimiento ( $c$ )		

Se paga un precio por buscar todas esas mejoras, tenemos más valores que fijar para los hiperparámetros

En general los valores popularmente aceptados funcionan bien, pero puede ser que haya un problema a resolver que requiera más afinación

# Tasas de aprendizaje





# Agendas avanzadas de tasa de aprendizaje

# Dos agendas bastante nuevas

**AdaGrad**

**RMSProp**

# AdaGrad

## Adaptive Gradient algorithm

- Propuesta en el 2011
- Actualiza dinámicamente la TA, para cada peso individualmente, en cada actualización

# AdaGrad

## Adaptive Gradient algorithm

La regla original era:

$$w(t + 1) = w(t) - \eta \frac{\partial L}{\partial w} (t)$$

$$w(t + 1) - w(t) = - \eta \frac{\partial L}{\partial w} (t)$$

$$\Delta w = - \eta \frac{\partial L}{\partial w} (t)$$

Nada nuevo hasta ahora

# AdaGrad

## Adaptive Gradient algorithm

$$\Delta w = -\eta \frac{\partial L}{\partial w} (t)$$

$$\Delta w_i(t) = -\frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i} (t)$$

Indice del peso,  
La regla de  
actualización es  
individual para  
cada peso

Iteración (tiempo o época) en el cuál se  
hace la actualización

# AdaGrad

## Adaptive Gradient algorithm

$$\Delta_{w_i}(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

G es la “magia”

$$G_i(t) = G_i(t - 1) + \left( \frac{\partial L}{\partial w_i}(t) \right)^2$$

con  $G_i(0) = 0$

# AdaGrad

## Adaptive Gradient algorithm

Tasa de aprendizaje efectiva

$$\Delta w_i(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

$$G_i(t) = G_i(t-1) + \left( \frac{\partial L}{\partial w_i}(t) \right)^2$$

con  $G_i(0) = 0$

$$G_i(0) = 0$$

$$G_i(1) = 0 + \text{no\_neg}$$

$$G_i(2) = [0 + \text{no-neg}] + \text{no-neg}$$

.

.

Eta / función creciente monotonícamente = función decreciente monotonica  
Epsilon es un número pequeño para evitar la división por cero cuando  $G = 0$



# AdaGrad

## Adaptive Gradient algorithm

- Es inteligente
- Agenda de tasa de aprendizaje adaptiva
- Basada en el entrenamiento mismo
- **Por peso**

# RMSQProp

**Root Mean Square Propagation (parecida a AdaGrad)**

$$\Delta w_i(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

G es la “magia”

$$G_i(t) = \beta G_i(t - 1) + (1 - \beta) \left( \frac{\partial L}{\partial w_i}(t) \right)^2$$

con  $G_i(0) = 0$

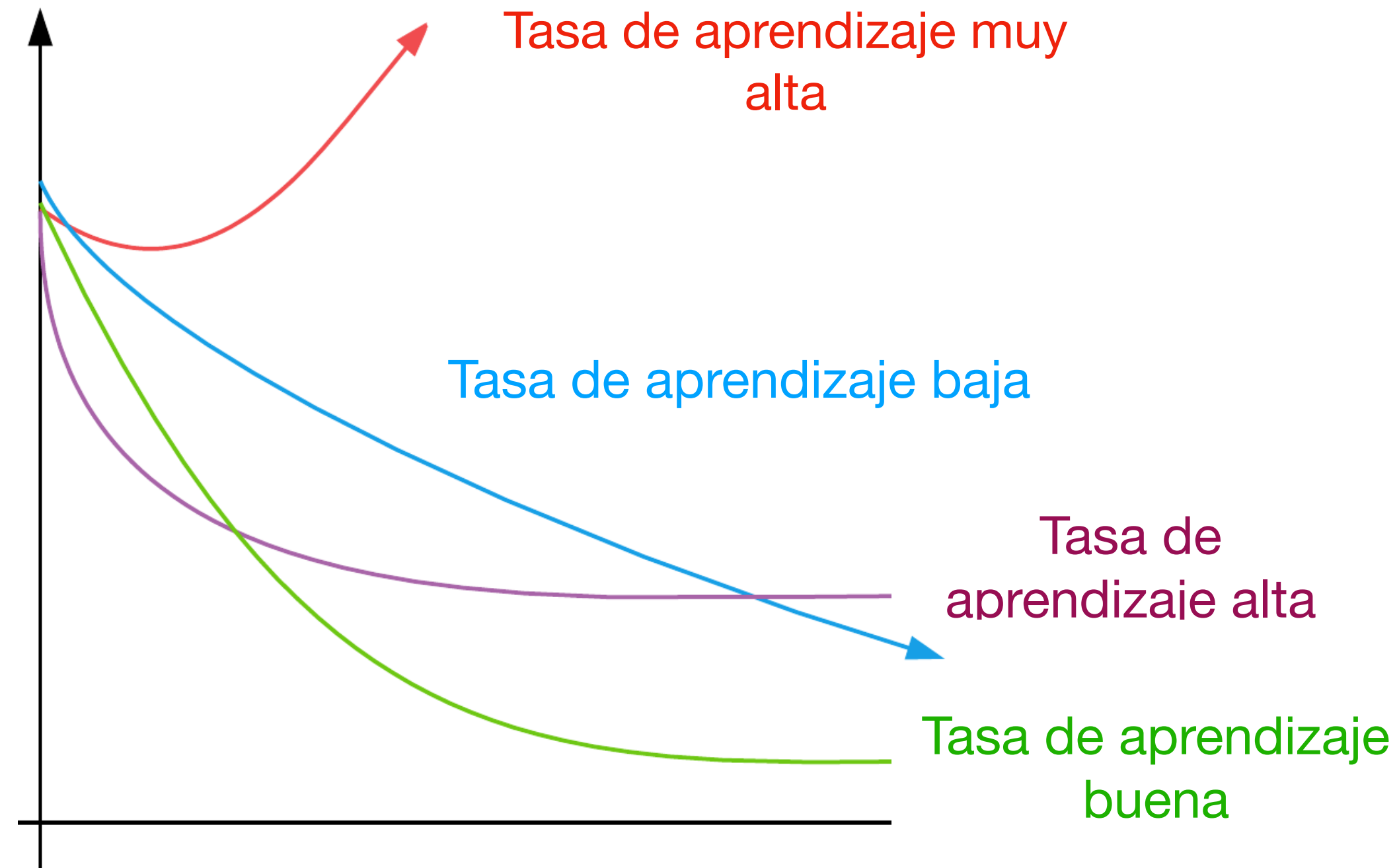
A los dos términos de  $G()$  se le asignan pesos.  $\beta$  es un nuevo hiperparámetro con valor entre 0 y 1. Generalmente se utiliza 0.9, o algo cercano. Este parámetro hace que la tasa efectiva de aprendizaje ya no decrezca monótonicamente y puede variar hacia arriba o abajo.

Los dos métodos son mejores y la evidencia empírica muestra que el ajuste es más eficiente.

¿Se podrá mejorar esto?

# Hasta ahora hemos visto

## Agendas de Tasa de Aprendizaje



$$\Delta w_i(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

AdaGrad, RMSProp

## Momento



$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \alpha \eta \frac{\partial L}{\partial w}(t-1)$$

# Adam

## Adaptive moment estimation

- El optimizador más avanzado (muy rápido y eficiente)
- Combinación de las dos anteriores
- Propuesta en el 2015

$$\Delta_{w_i}(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$

AdaGrad, RMSProp

$$w \leftarrow w(t) - \underbrace{\eta \frac{\partial L}{\partial w}(t)}_{\text{Actualización actual}} - \underbrace{\alpha \eta \frac{\partial L}{\partial w}(t-1)}_{\text{Actualización anterior}}$$

Actualización actual

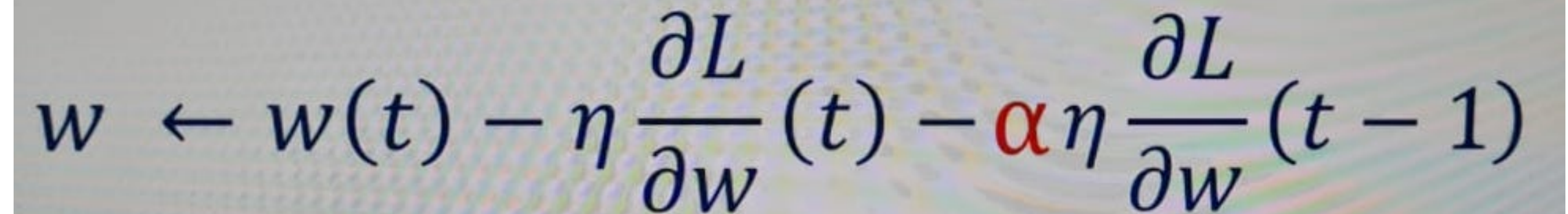
Actualización  
anterior




# Adam

- AdaGrad y RMSProp no incluyen nada del momento

$$\Delta w_i(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} \frac{\partial L}{\partial w_i}(t)$$


$$w \leftarrow w(t) - \eta \frac{\partial L}{\partial w}(t) - \alpha \eta \frac{\partial L}{\partial w}(t-1)$$


$$\Delta w_i(t) = - \frac{\eta}{\sqrt{G_i(t) + \epsilon}} M_i(t)$$

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha) \frac{\partial L}{\partial w_i}(t)$$

$$M_i(0) = 0$$

**Como en toda ciencia, la Ciencia de Datos es una cadena larga de investigación académica, acumulando el conocimiento nuevo sobre lo anterior**