Facsímil 01

Los cimientos

Capítulo 07

Funciones de pérdida y optimizadores

686f6c61 · 2026

Capítulo 07: Funciones de pérdida y optimizadores

Entrando en el tema

En el capítulo anterior construiste el motor: la retropropagación. Sabe calcular gradientes. Sabe en qué dirección ajustar cada peso para reducir el error. Pero le faltan dos piezas para funcionar: **qué error medir** y **cómo aplicar los ajustes**.

La primera pieza es la función de pérdida: la regla que decide qué cuenta como «error». La segunda es el optimizador: el algoritmo que decide cómo traducir los gradientes en actualizaciones concretas de los pesos.

Elegir mal cualquiera de las dos es como tener un coche con un motor excelente pero sin volante o sin frenos. El motor hace su trabajo, pero el coche no va a ninguna parte.

Funciones de pérdida: cómo medir el error

Una función de pérdida toma dos cosas (la predicción del modelo y la respuesta real) y devuelve un número: el error.[1] Cuanto mayor es el número, peor es la predicción. El objetivo del entrenamiento es hacer ese número lo más pequeño posible.

Hay tres funciones que cubren la inmensa mayoría de casos prácticos:

FUNCIÓN	SE USA EN	QUÉ MIDE
Cross-entropy	Clasificación, LLMs	Diferencia entre la distribución de probabilidad predicha y la real. Penaliza fuertemente las predicciones confiadas pero equivocadas.
MSE (error cuadrático medio)	Regresión	Media de los errores al cuadrado. Penaliza más los errores grandes que los pequeños (por el cuadrado).
Contrastive loss	*Embeddings*, búsqueda semántica	Acerca representaciones de ejemplos similares y aleja las de ejemplos diferentes.

Las fórmulas importan porque te dicen qué está castigando exactamente el entrenamiento:

PÉRDIDA	FÓRMULA SIMPLIFICADA	LECTURA
Binary cross-entropy	$-[y \log(p) + (1 - y) \log(1 - p)]$	Para una salida binaria p. Castiga mucho estar seguro y equivocarse.
Cross-entropy multiclase	$-\log(p_y)$	Para k clases. Solo mira la probabilidad asignada a la clase correcta.
MSE	$\frac{1}{n}\sum_i (\hat{y}_i - y_i)^2$	Para valores continuos. Los errores grandes pesan más por el cuadrado.
Triplet loss (contrastive)	$\max(0,\ d(a,p) - d(a,n) + m)$	Para representaciones. Acerca el ancla a un ejemplo similar y la aleja de uno distinto, con un margen m de separación.

Los símbolos que aparecen en esas fórmulas:

SÍMBOLO	SIGNIFICADO	EJEMPLO
y	Etiqueta real. En clasificación binaria vale 0 o 1	1 (la clase positiva)
p	Probabilidad que el modelo predice para la clase positiva	0,90
p_y	Probabilidad que el modelo asigna a la clase correcta (multiclase)	0,75 para la clase `alta`
n	Número de ejemplos sobre los que se promedia	128
$\hat{y}_i$	Predicción del modelo para el ejemplo i	19,4 (precio estimado)
y_i	Valor real del ejemplo i	21,0 (precio real)
a	Ancla: el ejemplo de referencia	una foto de un gato
$d(a,p)$	Distancia entre el ancla y un ejemplo similar (positivo)	pequeña si el modelo va bien
$d(a,n)$	Distancia entre el ancla y un ejemplo distinto (negativo)	grande si el modelo va bien
m	Margen: separación mínima exigida entre ambas distancias	0,2

En palabras: las tres primeras comparan lo que el modelo predice con lo que debería salir. La binaria y la multiclase trabajan con probabilidades y castigan asignar poca probabilidad a la respuesta correcta; la MSE trabaja con números y castiga la distancia al cuadrado. La *triplet*

loss no compara con una etiqueta: empuja a que el ancla quede más cerca de lo similar que de lo distinto, al menos por un margen m.

Ejemplo concreto: si la clase correcta es `alta` y el modelo le da probabilidad $p_y = 0{,}01$, la cross-entropy es $-\log(0{,}01) \approx 4{,}61$. Si le da $p_y = 0{,}90$, baja a $-\log(0{,}90) \approx 0{,}105$. No es una penalización lineal: fallar con mucha confianza duele mucho más. Por eso esta pérdida encaja tan bien con clasificación y predicción de siguiente token.

Cross-entropy es la reina indiscutible de la IA moderna. Cada vez que un LLM predice el siguiente token, está minimizando una *cross-entropy* entre la distribución que predice y la distribución real (que asigna probabilidad 1 al token correcto y 0 al resto).[2] Funciona especialmente bien porque su gradiente tiene una forma sorprendentemente simple. Para softmax seguido de cross-entropy, el gradiente respecto a la puntuación z_i de cada clase es exactamente la probabilidad predicha menos la real:

$$\frac{\partial E}{\partial z_i} = p_i - y_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
z_i	Puntuación (logit) que el modelo da a la clase i antes de softmax	3,1
p_i	Probabilidad que el modelo asigna a la clase i tras softmax	0,99
y_i	Valor real: 1 para la clase correcta, 0 para el resto	1

En palabras: el gradiente es el error directo, lo que el modelo predijo menos lo que debía predecir. Si el modelo asigna un 99 % de probabilidad al token correcto, $p_i - y_i$ es casi cero (ya lo hace bien). Si asigna un 1 %, el gradiente es grande (tiene mucho que aprender). Esa simplicidad, que el gradiente sea literalmente el error, es una de las razones de su éxito.

Esta pérdida tiene un nombre cuando se usa para evaluar modelos de lenguaje: la **perplejidad** (*perplexity*). Es la exponencial de la cross-entropy media:

$$\text{PPL} = e^{H}, \qquad H = -\frac{1}{N}\sum_{i=1}^{N} \log p(\text{token}_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
PPL	Perplejidad: la cross-entropy llevada a una escala intuitiva	12,5
H	Cross-entropy media sobre todos los tokens evaluados	2,53
N	Número de tokens evaluados	1.000.000
$p(\mathrm{token}_i)$	Probabilidad que el modelo asignó al token correcto en la posición i	0,21

Se interpreta como «entre cuántas opciones, de media, duda el modelo en cada token». Una perplejidad de 1 sería un modelo perfecto, que siempre acierta el siguiente token; una de 50 significa que, de media, está tan indeciso como si eligiera entre 50 opciones igual de probables. Cuando leas que un modelo de lenguaje tiene una perplejidad de X, estás leyendo su cross-entropy traducida a una escala más intuitiva.

Un matiz práctico: obligar al modelo a predecir exactamente 1 para la clase correcta y 0 para el resto puede volverlo demasiado confiado. El **label smoothing** suaviza un poco esas etiquetas, por ejemplo 0,9 para la correcta y un pequeño resto repartido entre las demás. Esto regulariza, mejora la calibración (que la confianza del modelo se parezca a su acierto real) y es habitual al entrenar Transformers.

MSE es la elección natural cuando predices valores numéricos: el precio de una casa, la temperatura de mañana, los ingresos del próximo trimestre. El cuadrado en la fórmula (`(predicción - realidad)²`) hace que los errores grandes pesen mucho más que los pequeños: equivocarse por 100 es cien veces peor que equivocarse por 10, pero la penalización es diez mil veces mayor. Esto empuja al modelo a evitar errores graves a toda costa.

Contrastive loss no se usa para predecir, sino para **representar**. Su objetivo es que dos imágenes del mismo objeto tengan vectores de *embedding* cercanos, y dos imágenes de objetos distintos los tengan lejanos. Es la base de los sistemas de búsqueda por similitud y del aprendizaje de representaciones.

El margen y las pérdidas de clasificación

Las tres pérdidas anteriores son las que más vas a ver en la IA moderna, pero esconden una idea más antigua y muy iluminadora que conviene conocer, porque organiza casi todo el aprendizaje supervisado clásico y reaparece, disfrazada, dentro de la *cross-entropy*. La idea es el **margen**, y nace de una pregunta sencilla: en una clasificación, ¿cuándo una predicción es buena, no solo correcta?

Para verlo, coloquemos la clasificación binaria en su forma más cruda. El modelo calcula una **puntuación** (en inglés *score*), un número que es positivo cuando apuesta por la clase positiva y negativo cuando apuesta por la negativa; cuanto más lejos de cero, más confiado está.[3] Si

codificamos la etiqueta real como $+1$ para la clase positiva y -1 para la negativa, el **margen** es el producto de la puntuación por la etiqueta:

$$\text{margen} = \text{puntuación} \cdot y$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
puntuación	Número que produce el modelo: positivo apuesta por la clase $+1$, negativo por la -1	2,3
y	Etiqueta real codificada como $+1$ o -1	$+1$
margen	Puntuación por etiqueta: positivo si acierta, negativo si falla; su tamaño es la confianza	2,3

En palabras: el margen junta en un solo número las dos cosas que importan de una predicción. Su **signo** dice si el modelo acierta (margen positivo, la puntuación va en el sentido de la etiqueta) o falla (margen negativo). Su **tamaño** dice con cuánta holgura, es decir, la confianza. Un margen de $+2{,}3$ es un acierto cómodo; uno de $+0{,}1$ es un acierto por los pelos, a punto de equivocarse; uno de $-1{,}5$ es un error claro y confiado, el peor caso. Geométricamente, si los pesos están normalizados, el margen es la distancia con signo del ejemplo a la **frontera de decisión**, la línea (o el plano) que separa una clase de la otra.

Con el margen en la mano, las pérdidas de clasificación se vuelven fáciles de comparar, porque todas son funciones de ese único número: castigan los márgenes negativos y premian los positivos, y solo se diferencian en *cómo* de severas son en la zona de transición. Las tres clásicas son:

PÉRDIDA	FÓRMULA (EN FUNCIÓN DEL MARGEN M)	IDEA
0-1 (*zero-one*)	$1[m \leq 0]$	Vale 1 si hay error, 0 si hay acierto. Es lo que de verdad quieres minimizar.
Bisagra (*hinge*, SVM)	$\max(0,\ 1 - m)$	Cero en cuanto el margen supera 1; si no, crece de forma lineal. Exige un acierto con holgura.
Logística	$\log_2(1 + e^{-m})$	Nunca llega a cero del todo: siempre empuja a aumentar el margen, cada vez con menos fuerza.

El problema de la pérdida **0-1**, la que mide lo que realmente nos importa (cuántas veces nos equivocamos), es que es inservible para entrenar con gradientes: es plana a ambos lados y da un escalón vertical en cero, así que su derivada es cero en casi todas partes y no señala ninguna dirección de mejora. El motor que montaste en el capítulo anterior se quedaría

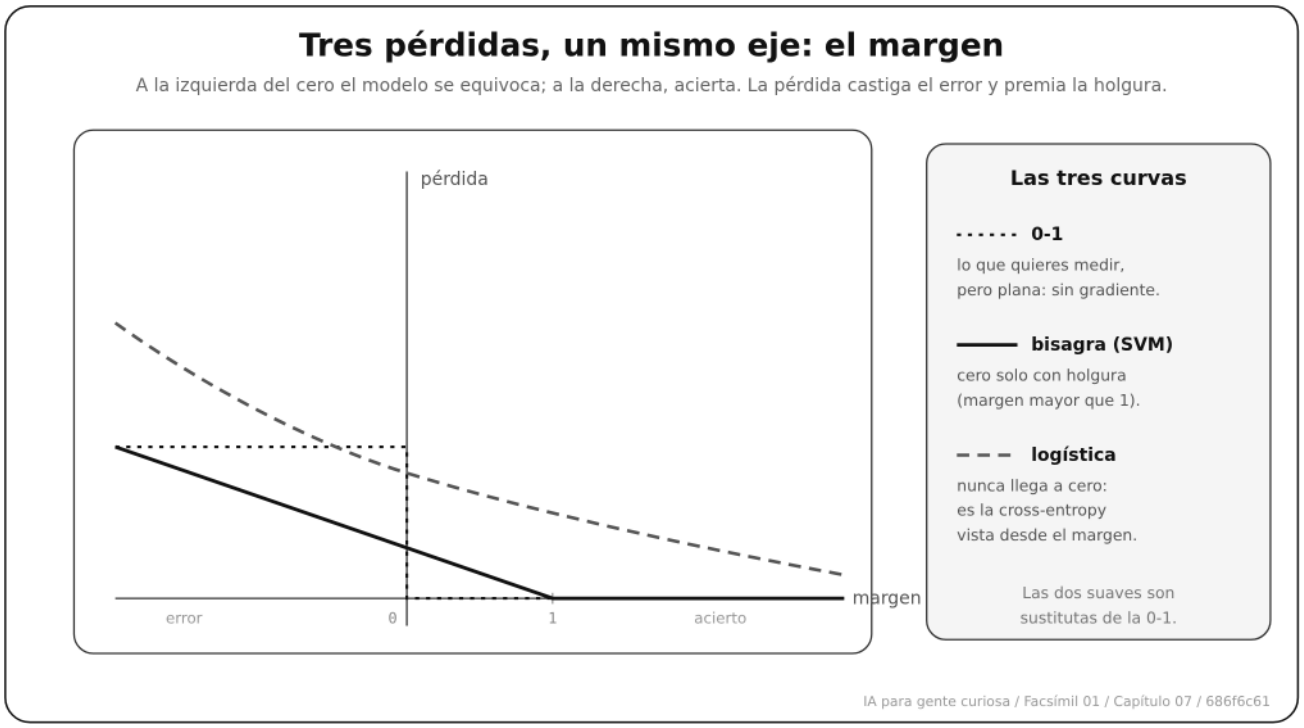
parado. Por eso, en la práctica, se sustituye por una pérdida **sustituta** (en inglés *surrogate*) que sí tenga pendiente: una que se parezca a la 0-1 pero baje de forma suave, de modo que el descenso de gradiente tenga siempre hacia dónde tirar.

$$\text{Bisagra}(m) = \max(0,\ 1 - m) \qquad \text{Logística}(m) = \log_2(1 + e^{-m})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
m	Margen del ejemplo (puntuación por etiqueta)	0,4
$\max(0,\ \cdot)$	Recorta a cero los valores negativos: si ya hay holgura, no penaliza	$\max(0,\ 0{,}6) = 0{,}6$
e^{-m}	Exponencial del margen cambiado de signo: enorme si el margen es muy negativo	$e^{-0{,}4} \approx 0{,}67$

En palabras: la pérdida **bisagra** dibuja una rampa que toca el suelo justo cuando el margen llega a 1, no a 0. Esa exigencia de pasar de largo, de acertar con un colchón de seguridad, es la idea central de las **máquinas de vectores de soporte** (SVM), que estudiarás como modelo en el capítulo de *machine learning* clásico: no basta con caer del lado correcto de la frontera, hay que hacerlo con margen.[4] La pérdida **logística**, en cambio, nunca se conforma: aunque el margen sea ya grande y positivo, sigue empujando un poquito más, con una fuerza que se desvanece poco a poco. Es la pérdida de la **regresión logística**, y no es una desconocida: es exactamente la *binary cross-entropy* de antes, reescrita en función del margen en lugar de la probabilidad. La reina de la IA moderna y este clásico de los años cincuenta son, por debajo, la misma pérdida.

Un ejemplo numérico fija las diferencias. Para un acierto justo, con margen $m = 0{,}4$: la pérdida 0-1 vale 0 (acertó, punto), la bisagra vale $\max(0,\ 1 - 0{,}4) = 0{,}6$ (acertó, pero sin la holgura que exige, así que sigue penalizando), y la logística vale $\log_2(1 + e^{-0{,}4}) \approx 0{,}76$ (también insiste en mejorar). Para un error confiado, con $m = -1{,}5$: la 0-1 vale 1, la bisagra $\max(0,\ 1 + 1{,}5) = 2{,}5$ y la logística $\log_2(1 + e^{1{,}5}) \approx 1{,}82$. Las dos sustitutas castigan mucho más el error confiado que el acierto ajustado, que es justo el comportamiento que se busca.



La lección que deja el margen va más allá de estas tres curvas. Casi cualquier pérdida de clasificación se entiende como una forma distinta de penalizar márgenes negativos, y elegir una u otra es elegir cuánto castigas la duda frente al error franco: la bisagra deja de insistir en cuanto hay holgura suficiente, lo que produce fronteras que dependen solo de los ejemplos difíciles, los más cercanos al borde; la logística nunca suelta, lo que da probabilidades mejor calibradas a cambio de no ignorar nunca a los ejemplos fáciles. No hay una mejor en abstracto; hay una mejor para lo que necesitas.

Optimizadores: cómo aplicar los ajustes

Una vez que la retropropagación ha calculado los gradientes, el optimizador decide **cómo** actualizar los pesos. La regla básica es siempre la misma:

$$w \leftarrow w - \eta \cdot \frac{\partial E}{\partial w}$$

Pero los optimizadores modernos añaden memoria estadística y adaptación a este proceso.[5]

OPTIMIZADOR	IDEA CLAVE	CUÁNDO USARLO
SGD	Gradiente descendente con mini-lotes. El más simple.	Cuando necesitas control total sobre el *learning rate* y el *momentum*.
Adam	*Learning rate* adaptativo por parámetro. Funciona bien sin ajuste.	El estándar para la mayoría de tareas. Buen punto de partida.
AdamW	Adam + *weight decay* correcto (desacoplado).	El estándar para entrenar LLMs y Transformers.
Muon	Ortogonaliza el *momentum* de las matrices de las capas ocultas. Más eficiente en cómputo.	Modelos grandes donde el cómputo aprieta; solo en capas ocultas, con Adam para el resto.

SGD (*Stochastic Gradient Descent*) es el abuelo de todos los optimizadores. Actualiza los pesos en la dirección del gradiente, con un tamaño de paso fijo (η). Es simple, funciona, pero requiere ajustar el *learning rate* con cuidado: si es muy alto, el entrenamiento diverge; si es muy bajo, tarda una eternidad.

Antes de llegar a Adam hace falta una idea intermedia: el **momentum** (momento). El SGD básico da cada paso mirando solo el gradiente actual, así que en zonas ruidosas va dando bandazos. El momentum le añade memoria: acumula una media de los gradientes recientes y se mueve en esa dirección acumulada, como una bola que rueda cuesta abajo y gana velocidad donde la pendiente es consistente, sin frenarse en cada pequeño bache. Acelera el avance en las direcciones estables y amortigua las oscilaciones. En la práctica casi nunca se usa SGD sin momentum.

Adam (*Adaptive Moment Estimation*) mantiene una estimación del *momentum* (media móvil de los gradientes pasados) y una estimación de la varianza. Con estas dos, adapta el *learning rate* para cada parámetro individualmente. Un parámetro que recibe gradientes grandes y consistentes recibe pasos grandes. Uno que recibe gradientes pequeños o ruidosos recibe pasos pequeños. Esto hace que Adam funcione sorprendentemente bien sin necesidad de ajustar el *learning rate*.

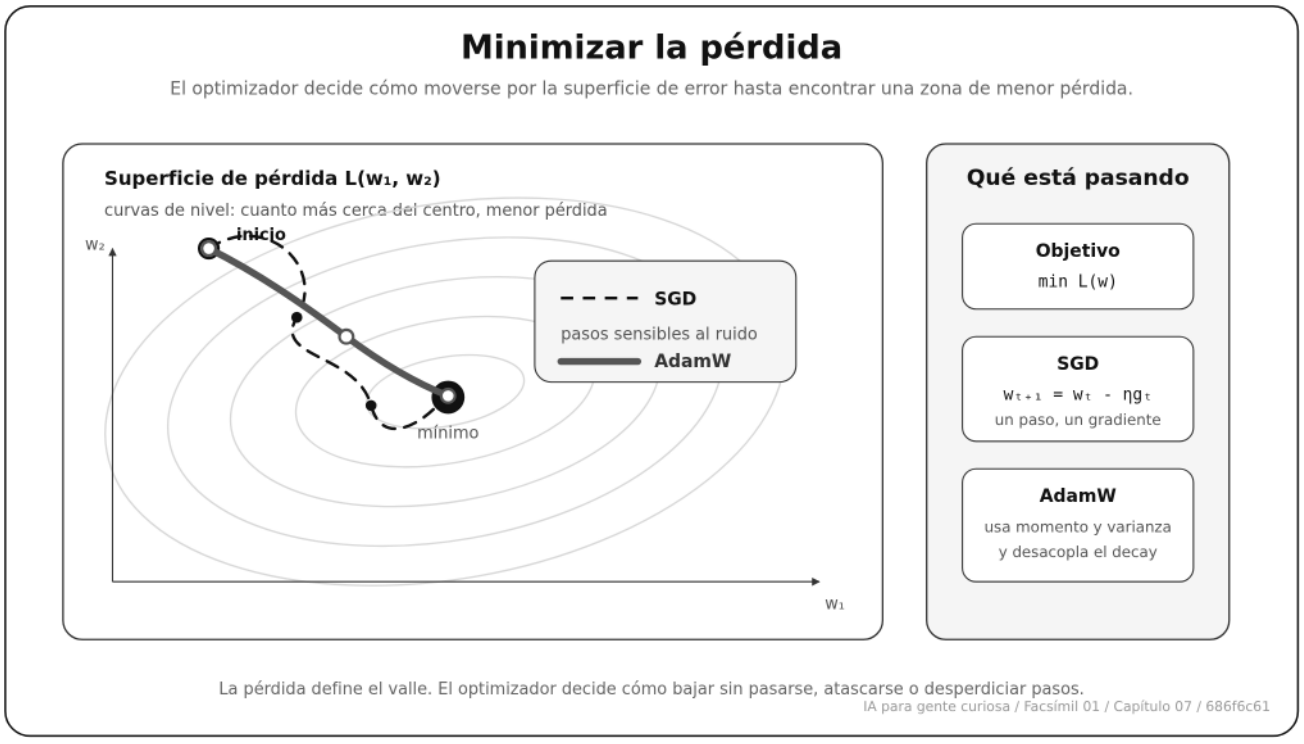
AdamW corrige un error sutil en la implementación original de Adam: la forma en que aplica la regularización *weight decay*. En Adam clásico, la *weight decay* está acoplada a la tasa de aprendizaje adaptativa, lo que reduce su efectividad.[6] AdamW la desacopla, aplicando la regularización directamente a los pesos. Es el optimizador estándar para entrenar LLMs y Transformers.

Un detalle que marca la diferencia al entrenar modelos grandes: el *learning rate* casi nunca es constante, sino que sigue un **calendario** (*schedule*). Lo habitual es un *warmup* al principio, empezar con un *learning rate* muy bajo y subirlo poco a poco durante las primeras iteraciones, para que el modelo no dé bandazos cuando los pesos todavía están desordenados,

seguido de un *decay,* bajarlo de forma progresiva a medida que el entrenamiento avanza, para afinar sin pasarse del mínimo. Todos los LLMs se entrenan con alguna variante de warmup más decay.

Una forma práctica de leer los optimizadores:

SEÑAL OBSERVADA	QUÉ PROBAR	POR QUÉ
La pérdida baja, pero muy lenta	Subir un poco el *learning rate* o usar Adam/AdamW.	El paso puede ser demasiado pequeño.
La pérdida oscila o explota	Bajar el *learning rate*, aplicar *gradient clipping* o revisar escalado de datos.	Los pasos son demasiado grandes o los gradientes son inestables.
Entrenamiento mejora y validación empeora	Añadir *weight decay*, *dropout*, más datos o *early stopping*.	Hay sobreajuste: el modelo memoriza.
Las clases minoritarias fallan	Ponderar la pérdida, cambiar métrica o revisar muestreo.	La pérdida media puede esconder que una clase casi no aprende.
La regresión castiga demasiado valores extremos	Probar MAE, Huber o revisar outliers.	MSE amplifica mucho errores grandes.



Cómo funciona por dentro

El capítulo ha dicho que Adam mantiene una estimación del *momentum* y otra de la varianza, y que con ellas adapta el paso de cada parámetro. Conviene abrir esa caja y ver el mecanismo concreto, porque entender cómo se calcula el paso explica por qué Adam funciona tan bien sin apenas ajuste manual.[7] Para cada peso del modelo, Adam guarda dos números que se actualizan en cada iteración: el primer momento, una media de los gradientes recientes que cumple el papel del *momentum*, y el segundo momento, una media de los gradientes al cuadrado que actúa como medida de la varianza. Ambos son medias móviles exponenciales, es decir, dan más peso a lo reciente y olvidan poco a poco lo antiguo.

El primer momento acumula la dirección reciente del gradiente. Se calcula así:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
m_t	Primer momento en el paso t: media móvil del gradiente	0,12
m_{t-1}	Primer momento del paso anterior	0,10
β_1	Factor de olvido del primer momento. Cuanto más cerca de 1, más memoria	0,9
g_t	Gradiente actual del peso, calculado por la retropropagación	0,30

En palabras: el nuevo m_t es casi todo lo que ya había acumulado (un 90 % con $\beta_1 = 0{,}9$) más una pizca del gradiente nuevo. Así la dirección se suaviza y no salta con cada gradiente ruidoso, igual que el *momentum* clásico.

El segundo momento acumula el tamaño típico del gradiente, no su dirección, porque eleva el gradiente al cuadrado y pierde el signo:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
v_t	Segundo momento en el paso t: media móvil del gradiente al cuadrado	0,090
v_{t-1}	Segundo momento del paso anterior	0,089
β_2	Factor de olvido del segundo momento. Suele ser muy cercano a 1	0,999
g_t^2	Gradiente actual elevado al cuadrado. Siempre positivo	0,090

En palabras: v_t registra cómo de grandes han sido los gradientes últimamente, sin importar si empujaban hacia arriba o hacia abajo. Un peso con gradientes grandes o muy variables acumula un v_t alto; uno con gradientes pequeños y estables acumula un v_t bajo.

Con esos dos números, Adam actualiza el peso dividiendo el primer momento por la raíz del segundo. Antes aplica una corrección de sesgo, porque en los primeros pasos m_t y v_t arrancan en cero y quedan artificialmente bajos; los símbolos $\hat{m}_t$ y $\hat{v}_t$ son esas versiones corregidas que compensan el arranque frío.

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\hat{v}_t + \epsilon}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
θ_t	Valor del peso tras la actualización	0,481
θ_{t-1}	Valor del peso antes de la actualización	0,500
η	*Learning rate*: tamaño base del paso	0,001
$\hat{m}_t$	Primer momento corregido de sesgo	0,12
$\hat{v}_t$	Segundo momento corregido de sesgo	0,090
ϵ	Número diminuto que evita dividir por cero	0,00000001

En palabras: el paso es proporcional a la dirección acumulada $\hat{m}_t$ dividida por cuánto se mueve ese peso de costumbre, $\hat{v}_t$. Aquí está la clave de la adaptación por parámetro. Un peso con gradientes consistentes y grandes tiene un $\hat{m}_t$ alto frente a un $\hat{v}_t$ moderado, así que el cociente es grande y recibe un paso amplio. Un peso con gradientes ruidosos o muy variables tiene un $\hat{v}_t$ alto, el denominador crece y el paso se encoge. Adam, en la práctica, le da a cada peso su propio *learning rate* efectivo sin que nadie lo configure peso a peso, y por eso funciona razonablemente bien con los valores por defecto.

AdamW añade una última pieza al mecanismo. El *weight decay*, esa regularización que empuja los pesos hacia valores pequeños, no entra dentro del cociente adaptativo, sino que se aplica aparte, restando directamente una fracción del propio peso en cada paso. Esa separación es lo que significa el término desacoplado: en Adam clásico el *weight decay* viajaba mezclado con la actualización adaptativa y su efecto quedaba diluido de forma desigual entre parámetros; en AdamW se resta limpio, con la misma fuerza para todos, y la regularización recupera el efecto que se espera de ella.[8] Ese matiz, aparentemente menor, es la razón de que AdamW se haya convertido en el estándar para entrenar Transformers y modelos de lenguaje.

Más allá de Adam: la nueva generación

AdamW lleva años siendo el estándar, pero desde 2023 ha aparecido una familia de optimizadores que le disputa el trono, y conviene conocerla porque ya se usa para entrenar modelos de primera línea.

El más comentado es **Muon**, de *MomentUm Orthogonalized by Newton-Schulz*, propuesto por Keller Jordan a finales de 2024.[9] La idea, en una frase: antes de dar el paso, Muon endereza el momento. Coge la dirección acumulada (el mismo *momentum* que ya conoces) y la ortogonaliza, es decir, la convierte en una matriz cuyas direcciones tienen todas una fuerza parecida, mediante una operación iterativa llamada Newton-Schulz que basta con repetir unas cinco veces. El efecto práctico es que ningún sentido del espacio de pesos se come a los demás, así que el entrenamiento avanza más equilibrado y llega a la misma calidad con menos pasos que AdamW. Muon se aplica solo a las matrices de pesos de las capas ocultas; los *embeddings*, la capa de salida y los parámetros sueltos (escalares y *bias*) se siguen entrenando con Adam, porque ahí la ortogonalización no aporta.

Que no es un experimento de laboratorio lo demostró Moonshot AI en 2025: entrenó **Kimi K2**, un modelo de un billón de parámetros, de los que se activan unos 32.000 millones por cada token porque es una mezcla de expertos, con una variante de Muon llamada **MuonClip**, y completó 15,5 billones de tokens de preentrenamiento sin un solo pico de inestabilidad.[10] El "Clip" tapa el talón de Aquiles de Muon a gran escala: cada cierto número de pasos reescala las matrices de consulta y clave de la atención para que sus valores no se disparen, que es justo lo que solía descarrilar estos entrenamientos.

Muon no está solo. **Lion**, de *EvoLved Sign Momentum*, lo descubrió Google en 2023 dejando que un buscador automático explorara algoritmos de optimización.[11] Lion se queda solo con el **signo** del momento, de modo que todos los pesos dan un paso del mismo tamaño y lo único que cambia es la dirección; a cambio de algo menos de finura, gasta la mitad de memoria que Adam, porque no necesita guardar el segundo momento. En esa misma búsqueda de ahorrar memoria está **Adafactor**, que en vez del segundo momento entero almacena una versión factorizada, y en el extremo opuesto está **Shampoo**, un método de segundo orden que estima la curvatura de la superficie por bloques para dar pasos mejor informados, a costa de más cálculo.

¿Quiere decir esto que AdamW está acabado? No. Sigue siendo la opción por defecto, la más probada y la que mejor se porta sin tocar nada, y para la inmensa mayoría de proyectos es la elección correcta. Pero cuando se entrenan modelos enormes y cada hora de GPU cuenta, esta nueva generación es la frontera donde hoy se pelea por exprimir el cómputo. La lección de fondo no cambia: el optimizador no aprende por ti, solo decide cómo moverte por la superficie de error, y elegirlo bien puede ahorrar semanas de entrenamiento.

Problemas comunes del entrenamiento

Incluso con la función de pérdida y el optimizador correctos, el entrenamiento puede fallar. Dos problemas aparecen una y otra vez:

Overfitting (sobreajuste). El modelo memoriza los datos de entrenamiento pero falla estrepitosamente con datos nuevos. Es como un estudiante que se aprende las respuestas del examen de memoria sin entender la materia: saca un 10 en el simulacro y suspende el examen real.[12] Soluciones: *dropout* (apagar neuronas aleatoriamente durante el entrenamiento), regularización (penalizar pesos grandes), más datos (la mejor medicina), *data augmentation* (generar variaciones de los datos existentes).

Vanishing gradients (desvanecimiento del gradiente). En redes profundas, el gradiente puede hacerse tan pequeño en las primeras capas que dejan de aprender. Es como si el profesor susurrara la corrección al alumno de la última fila, y el mensaje se fuera atenuando fila a fila hasta que los de delante no oyen nada.[13] Soluciones: ReLU en lugar de sigmoide (su gradiente es 1 para valores positivos), *skip connections* (atajos que permiten al gradiente saltar capas), *batch normalization* (normalizar las activaciones de cada capa).

El contrato mínimo de una run

Una *run* de entrenamiento no debería ser «he lanzado un modelo y ya veremos». Debería tener un contrato mínimo antes de empezar:

ELEMENTO	DECISIÓN	EJEMPLO
Tarea	Clasificación binaria, multiclase, regresión o representación.	Priorizar tickets: multiclase.
Salida	Número de salidas y activación final.	3 clases → softmax.
Pérdida	Función que coincide con la tarea.	Cross-entropy multiclase.
Métrica	Qué miras para decidir si sirve.	F1 macro si hay desbalance; accuracy si las clases están equilibradas.
Optimizador	Algoritmo y sus hiperparámetros.	AdamW, `lr=1e-3`, `weight_decay=1e-2`.
Validación	Partición que no actualiza pesos.	80/20 estratificado o validación temporal si hay fechas.
Criterio de parada	Cuándo dejar de entrenar.	Parar si validación no mejora en 5 épocas.
Observabilidad	Qué registras por época.	Train loss, val loss, métrica, norma del gradiente, learning rate.

Este contrato es una defensa contra el autoengaño. Si cambias pérdida, métrica y partición cada vez que algo sale mal, no estás optimizando: estás persiguiendo resultados. Una run seria deja claro qué cuenta como mejora antes de mirar el resultado.

En el día a día

En la práctica, casi nunca eliges la función de pérdida desde cero. Los *frameworks* ya tienen las implementaciones optimizadas. Pero sí tomas decisiones que dependen de entenderlas, y cada una es una decisión de ingeniería concreta, no una preferencia estética.

¿Cross-entropy o MSE? La pregunta no es de gusto, sino de qué supone cada pérdida sobre tu salida. La cross-entropy asume que el modelo emite una distribución de probabilidad y mide cuánta sorpresa le causa la respuesta correcta; encaja con clasificar (spam o no, gato o perro) y con predecir el siguiente token, donde la última capa es un softmax. La MSE asume una salida continua y penaliza la distancia al cuadrado, así que solo tiene sentido cuando predices un número real (un precio, una temperatura). Equivocarse aquí es más común de lo que parece y no siempre rompe el entrenamiento de forma visible: si usas MSE sobre probabilidades, el modelo aprende algo, pero su gradiente deja de ser el error limpio $p_i - y_i$ y la convergencia se vuelve lenta y peor calibrada. La regla operativa es mirar la activación final: softmax o sigmoide piden cross-entropy; una salida lineal pide MSE (o MAE/Huber si hay valores extremos que no quieres que dominen).

¿Qué optimizador y con qué calendario? Empieza con Adam o AdamW: su *learning rate* adaptativo por parámetro hace que funcionen bien sin afinar mucho, y cubren la inmensa mayoría de los casos. AdamW es preferible cuando vas a usar *weight decay*, porque lo desacopla de la actualización adaptativa y la regularización vuelve a tener el efecto esperado; por eso es el estándar al entrenar Transformers. La decisión no termina en el optimizador: en modelos grandes el *learning rate* casi nunca es constante. Un *warmup* corto al arranque evita que los pasos den bandazos cuando los pesos todavía están desordenados, y un *decay* posterior afina sin pasarse del mínimo. SGD con momentum sigue siendo una opción válida cuando quieres control total y estás dispuesto a buscar el *learning rate* a mano, algo habitual en visión por sus efectos de generalización.

¿Overfitting o underfitting? El diagnóstico se lee comparando dos curvas, no una. Si la pérdida de entrenamiento es baja pero la de validación es alta (y la brecha entre ambas crece con las épocas), tienes *overfitting*: el modelo memoriza en lugar de generalizar. La respuesta es regularizar (*weight decay*, *dropout*), conseguir más datos o aplicar *data augmentation*, y cortar con *early stopping* cuando la validación deja de mejorar. Si en cambio ambas pérdidas son altas y se estancan, tienes *underfitting*: el modelo es demasiado simple, el *learning rate* es inadecuado o no has entrenado lo suficiente; ahí toca subir capacidad, ajustar el optimizador o entrenar más. Confundir los dos lleva a aplicar el remedio contrario: regularizar un modelo que ya se queda corto solo empeora el resultado.

Por qué debería importarte

La función de pérdida es lo que le dices al modelo que quieres. No es un detalle técnico: es la especificación formal de tu objetivo.

Si usas MSE para clasificar, el modelo intentará minimizar la distancia numérica entre probabilidades, no maximizar la probabilidad de la clase correcta. Aprenderá algo, pero no lo que quieres. Si usas *cross-entropy* para regresión, el modelo asumirá que la salida es una distribución de probabilidad, no un valor continuo. Los resultados serán inconsistentes.

El optimizador, por su parte, determina si el modelo converge, a qué velocidad y a qué solución. Adam con un *learning rate* por defecto funciona en la mayoría de casos. Pero si tu modelo no converge, cambiar de optimizador o ajustar sus hiperparámetros puede ser la diferencia entre el éxito y el fracaso.

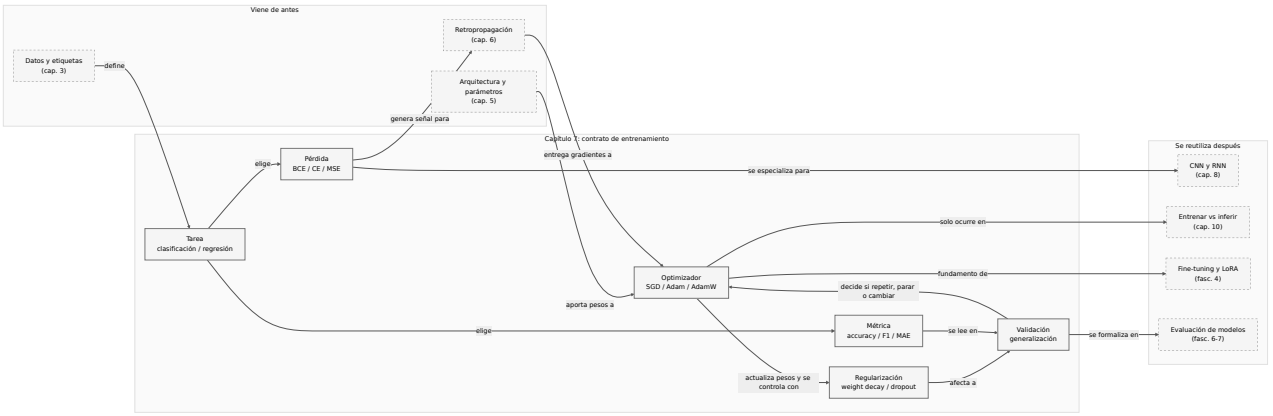
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Usar MSE para clasificación	MSE asume errores continuos y simétricos. La clasificación necesita comparar distribuciones de probabilidad. MSE penaliza igual una predicción de 0,4 cuando la respuesta es 1 que una predicción de 0,9 cuando la respuesta es 0.	*Cross-entropy* para clasificación. Siempre.
No monitorizar la pérdida de validación	Si solo miras la pérdida de entrenamiento, no detectas el *overfitting* hasta que es demasiado tarde. La pérdida de entrenamiento siempre baja; la de validación es la que importa.	Separa un conjunto de validación y monitoriza ambas curvas. Si divergen, estás sobreajustando.
Usar el *learning rate* por defecto sin comprobarlo	El valor por defecto de Adam (0,001) funciona en muchos casos, pero no en todos. Un *learning rate* demasiado alto hace que la pérdida oscile; uno demasiado bajo hace que el entrenamiento sea innecesariamente lento.	Prueba con 0,01, 0,001 y 0,0001. Observa la curva de pérdida: si oscila, baja el *learning rate*; si baja muy lentamente, súbelo.
Olvidar el *weight decay*	Sin regularización, los pesos pueden crecer descontroladamente, llevando a *overfitting*. AdamW incluye *weight decay* por defecto, pero SGD no.	Añade *weight decay* (típicamente 1e-4 a 1e-2). Observa si mejora la pérdida de validación.
Elegir métrica que contradice la pérdida	Puedes optimizar cross-entropy y celebrar accuracy mientras la clase minoritaria queda destrozada. La pérdida entrena; la métrica decide si te vale.	Define pérdida y métrica juntas. En desbalance, mira F1 macro, recall por clase o coste de error.
Comparar runs con particiones distintas	Si cada experimento usa otro split, no sabes si mejoró el modelo o si tuvo una validación más fácil.	Fija partición, semilla y protocolo antes de comparar optimizadores.

Cómo encaja todo

Este capítulo convierte el gradiente del capítulo 6 en una decisión de entrenamiento completa. La pérdida define qué significa fallar; el optimizador decide cómo moverse; la validación decide si esa mejora sirve fuera de los ejemplos que actualizan pesos.

También prepara una idea que volverá muchas veces: entrenar no es solo bajar una métrica interna. Es diseñar un contrato medible, ejecutarlo de forma reproducible y mirar si generaliza.


Viene de antes
Datos y etiquetas (cap. 3)
define
Retropropagación (cap. 6)
Arquitectura y parámetros (cap. 5)
genera señal para
Capítulo 7: contrato de entrenamiento
elige
Pérdida BCE / CE / MSE
entrega gradientes a
se especializa para
Tarea clasificación / regresión
Optimizador SGD / Adam / AdamW
aporta pesos a
solo ocurre en
fundamento de
decide si repetir, parar o cambiar
elige
Métrica accuracy / F1 / MAE
se lee en
Validación generalización
se formaliza en
actualiza pesos y se controla con
Regularización weight decay / dropout
afecta a
Se reutiliza después
CNN y RNN (cap. 8)
Entrenar vs inferir (cap. 10)
Fine-tuning y LoRA (fasc. 4)
Evaluación de modelos (fasc. 6-7)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Función de pérdida	Función que mide el error entre la predicción y la realidad. El entrenamiento busca minimizarla.
Cross-entropy	Pérdida estándar para clasificación. Mide la diferencia entre la distribución predicha y la real.
MSE	Error cuadrático medio. Pérdida estándar para regresión. Penaliza fuertemente los errores grandes.
Margen	Puntuación por etiqueta ($+1$/-1). Su signo dice si se acierta; su tamaño, con cuánta confianza.
Pérdida 0-1	Vale 1 si hay error y 0 si hay acierto. Mide lo que importa, pero es plana: no sirve para entrenar con gradientes.
Pérdida bisagra (*hinge*)	Sustituta de la 0-1 que solo deja de penalizar con holgura (margen mayor que 1). Es la pérdida de las SVM.
Pérdida sustituta (*surrogate*)	Pérdida suave y con pendiente que reemplaza a la 0-1 para que el descenso de gradiente tenga dirección de mejora.
Optimizador	Algoritmo que actualiza los pesos a partir de los gradientes. Controla cómo y cuánto se ajusta cada parámetro.
Adam	Optimizador con *learning rate* adaptativo por parámetro. El más usado en la práctica.
AdamW	Variante de Adam con *weight decay* desacoplado. Es el estándar habitual para Transformers y muchos entrenamientos modernos.
Muon	Optimizador que ortogonaliza el *momentum* de las matrices de las capas ocultas (con iteración de Newton-Schulz). Más eficiente en cómputo; se combina con Adam para el resto de parámetros.
Lion	Optimizador que actualiza con el signo del momento: pasos de magnitud uniforme y menos memoria que Adam.
Momentum	Memoria que acumula una media de los gradientes recientes para acelerar las direcciones estables y amortiguar las oscilaciones del SGD.
Calendario de *learning rate*	Plan que varía la tasa de aprendizaje durante el entrenamiento, normalmente con un *warmup* inicial seguido de un *decay* progresivo.
Warmup	Fase inicial en la que el *learning rate* empieza muy bajo y sube poco a poco para evitar bandazos mientras los pesos están desordenados.

TÉRMINO	DEFINICIÓN
Perplejidad	Exponencial de la cross-entropy media. Mide entre cuántas opciones equivalentes duda de media un modelo de lenguaje en cada token.
Label smoothing	Suavizado de las etiquetas (la correcta deja de ser un 1 exacto) que regulariza y mejora la calibración, habitual al entrenar Transformers.
Weight decay	Regularización que penaliza pesos grandes y ayuda a reducir sobreajuste.
Overfitting	El modelo memoriza los datos de entrenamiento pero no generaliza a datos nuevos.
Curva de validación	Evolución de pérdida o métrica en datos que no actualizan pesos. Permite detectar generalización, sobreajuste y parada temprana.
Early stopping	Detener el entrenamiento cuando la validación deja de mejorar durante varias épocas.

Antes de pasar página

- [ ] ¿Puedo explicar cuándo usar *cross-entropy* y cuándo MSE? (Si no, vuelve a la tabla de funciones de pérdida.)
- [ ] ¿Entiendo la diferencia entre SGD, Adam y AdamW? (Si no, vuelve a la sección de optimizadores.)
- [ ] ¿Sé qué aporta Muon frente a AdamW y por qué empieza a usarse en los modelos grandes? (Si no, vuelve a «Más allá de Adam».)
- [ ] ¿Sé qué es el *overfitting* y cómo combatirlo? (Si no, vuelve a «Problemas comunes».)
- [ ] ¿Puedo escribir el contrato mínimo de una run antes de entrenar? (Si no, vuelve a «El contrato mínimo de una run».)
- [ ] ¿Sé comparar BCE, MSE, SGD y AdamW y elegir una combinación según el problema? (Si no, vuelve a «Optimizadores: cómo aplicar los ajustes».)
- [ ] ¿Entiendo por qué el *learning rate* no puede ser ni muy alto ni muy bajo? (Si no, vuelve a «Dónde solía tropezar yo».)

En resumen

IDEA FUERZA	DETALLE
La función de pérdida define qué significa «equivocarse».	*Cross-entropy* para clasificar, MSE para regresión, *contrastive* para representaciones. Elegir mal la pérdida es pedirle al modelo que optimice lo incorrecto.
El optimizador decide cómo aplicar los gradientes.	SGD es simple pero necesita ajuste. Adam es adaptativo y funciona en la mayoría de casos. AdamW es el estándar para Transformers, y una nueva generación (Muon, Lion) empieza a disputarle el sitio en los modelos más grandes.
El *overfitting* y el *vanishing gradient* son los dos grandes enemigos.	Monitoriza la pérdida de validación, usa *dropout* y regularización, y elige bien tus funciones de activación.
Una run necesita contrato antes de empezar.	Tarea, salida, pérdida, métrica, optimizador, validación y criterio de parada deben estar definidos antes de mirar resultados.

Para saber más

Chen, X. y otros (2023). Symbolic discovery of optimization algorithms. *arXiv*. https://arxiv.org/abs/2302.06675

Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018

Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org

Jordan, K. (2024). *Muon: an optimizer for hidden layers in neural networks* [entrada de blog]. https://kellerjordan.github.io/posts/muon/

Kimi Team (2025). *Kimi K2: open agentic intelligence*. *arXiv*. https://arxiv.org/abs/2507.20534

Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980

LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539

Liang, P. y Sadigh, D. *Artificial intelligence: principles and techniques* (notas del curso CS221). Stanford University. https://stanford-cs221.github.io

Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101

Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0

Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 1929-1958. http://jmlr.org/papers/v15/srivastava14a.html

---

Notas

1. Goodfellow, I., Bengio, Y. y Courville, A. (2016). *Deep learning*. MIT Press. https://www.deeplearningbook.org. El capítulo 6.2 aborda en profundidad las funciones de pérdida, su relación con la estimación de máxima verosimilitud y cómo la elección de la función de pérdida determina lo que el modelo aprende. ↩

2. Rumelhart, D. E., Hinton, G. E. y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. https://doi.org/10.1038/323533a0. Aunque el artículo se centra en la retropropagación con MSE, establece el marco general donde cualquier función de pérdida diferenciable puede usarse para entrenar redes multicapa. ↩

3. Liang, P. y Sadigh, D. *Artificial intelligence: principles and techniques* (notas del curso CS221). Stanford University. https://stanford-cs221.github.io. Las notas presentan el armazón puntuación → margen → pérdida como base unificada del aprendizaje supervisado lineal, del que la pérdida *hinge* (SVM) y la logística son dos casos. ↩

4. Cortes, C. y Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273-297. https://doi.org/10.1007/BF00994018. El artículo introduce las máquinas de vectores de soporte, que buscan la frontera de máximo margen y cuya pérdida asociada es la *hinge*. ↩

5. Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980. Kingma y Ba presentaron Adam, que combina las ventajas de AdaGrad (tasas de aprendizaje adaptativas) y RMSProp (media móvil del gradiente), convirtiéndose en el optimizador por defecto para la mayoría de aplicaciones de *deep learning*. ↩

6. Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101. Los autores demostraron que desacoplar la *weight decay* de la actualización del gradiente en Adam mejora significativamente la generalización. ↩

7. Kingma, D. P. y Ba, J. (2015). Adam: a method for stochastic optimization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980. El artículo deriva las dos medias móviles, la corrección de sesgo y la regla de actualización que se reproducen en esta sección. ↩

8. Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization. En *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101. Los autores muestran que aplicar el *weight decay* fuera de la actualización adaptativa, en lugar de sumarlo al gradiente, mejora la generalización y vuelve más predecible el ajuste del hiperparámetro. ↩

9. Jordan, K. (2024). *Muon: An optimizer for hidden layers in neural networks* [entrada de blog]. https://kellerjordan.github.io/posts/muon/. Introduce Muon y la ortogonalización del momento mediante iteración de Newton-Schulz como vía para entrenar las capas ocultas con menos pasos que AdamW. ↩

.0. Kimi Team (2025). *Kimi K2: Open Agentic Intelligence*. arXiv:2507.20534. https://arxiv.org/abs/2507.20534. Describe MuonClip, que añade a Muon un reescalado periódico de las matrices de consulta y clave de la atención (*QK-clip*) para estabilizar el entrenamiento a gran escala, y reporta el preentrenamiento de Kimi K2 sin picos de pérdida. ↩

.1. Chen, X. y otros (2023). *Symbolic discovery of optimization algorithms*. arXiv:2302.06675. https://arxiv.org/abs/2302.06675. Lion, hallado por búsqueda simbólica de programas, actualiza con el signo del momento y, al no guardar el segundo momento, usa menos memoria que Adam manteniendo un rendimiento comparable. ↩

.2. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), 1929-1958. http://jmlr.org/papers/v15/srivastava14a.html. El *dropout* (desactivar aleatoriamente neuronas durante el entrenamiento) es una de las técnicas más efectivas para prevenir el *overfitting*. ↩

.3. LeCun, Y., Bengio, Y. y Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444. https://doi.org/10.1038/nature14539. Los autores explican cómo las funciones de activación como ReLU y las arquitecturas con conexiones residuales mitigan el problema del desvanecimiento del gradiente. ↩