ZOFF.TECH

GUÍA DE ESTUDIO · ESPAÑOL

Redes neuronales desde cero en C#

Una red neuronal completa y funcional con *backpropagation*, escrita en C# con SIMD, junto con una explicación de **por qué existe cada pieza**, partiendo de cero conocimientos previos.

2 152	98,02 %	27
LÍNEAS EN ESPAÑOL	MNIST EN 37 S	SECCIONES + GLOSARIO

Fabricio Zoff · Zoff.tech

github.com/fabbrik/neural-network-csharp

Mediciones: Apple M3 Pro (11 núcleos) · macOS 26.3 · .NET 10.0.10 · Arm64 RyuJIT

Una red neuronal completa y funcional con backpropagation, escrita en C# con SIMD, junto con una explicación de **por qué existe cada pieza**, partiendo de cero conocimientos previos.

Cómo usar esta guía. La Parte I explica las ideas sin código — léela primero, aunque tengas prisa. La Parte II traslada cada idea al C# real y explica la ingeniería. La Parte III es práctica, depuración y ejercicios. Los ejemplos resueltos de §7 y §8 han sido calculados y verificados; síguelos con una calculadora y entenderás backpropagation de verdad: el concepto del que depende todo lo demás.

> **Sobre el idioma.** El código, los nombres de identificadores, las banderas de línea de comandos y la salida de los programas se conservan en inglés a lo largo de toda la guía: son los artefactos reales del repositorio, y traducirlos describiría un programa que no existe. La prosa, los comentarios explicativos y las tablas están en español.
>
> **Sobre términos técnicos.** Esta guía evita traducir literalmente términos que programadores y estudiantes de IA suelen conocer en inglés. Por eso verás `forward pass`, `backward pass`, `backpropagation`, `mini-batch`, `learning rate`, `dataset`, `features`, `benchmark`, `buffer`, `cache`, `logit`, `softmax`, `cross-entropy` y `one-hot`. Cuando el español ayuda, aparece como explicación, no como reemplazo forzado.

Archivo	Contenido
`Activations.cs`	`IActivation` y las funciones de activación
`Perceptron.cs`	El perceptrón de una unidad de 1958 y su regla de actualización
`Dense.cs`	La capa densa (totalmente conectada): `forward pass`, `backward pass` y actualización de gradientes
`SimdOps.cs`	`Dot` y `AddScaled` vectorizados
`ILayer.cs`	Interfaz de capa no genérica, para que una red mezcle tipos de activación
`Network.cs`	Pila de capas, bucle de entrenamiento con SGD por mini-batches, `Summary()`
`Sequential.cs`	Constructor fluido estilo Keras con inferencia del tamaño de entrada
`Loss.cs`	`ILoss`, MSE y softmax con cross-entropy
`ModelIO.cs`	Guardado y carga de modelos entrenados
`GradientCheck.cs`	Verificación de backpropagation por diferencias finitas
`Datasets.cs`	Dataset «dos lunas» generado, para experimentos train/test
`Program.cs`	Demos: perceptrón, XOR, guardar/cargar, gradiente, dos lunas
`Idx.cs`	Lector del formato IDX de MNIST (big-endian, etiquetas one-hot)
`MnistData.cs`	Descarga MNIST, guarda el dataset en cache y se salta la demo limpiamente si no hay conexión
`ImageFile.cs`	Decodificación de PNG y Netpbm, sin dependencias
`DigitPreprocessor.cs`	Lleva cualquier imagen a las convenciones de MNIST — §22
`NN.Mnist/Program.cs`	El reconocedor de dígitos: entrenar, evaluar, leer una imagen
`models/`	Un reconocedor entrenado, incluido para que leer dígitos funcione en un clon nuevo
`bench/`	Los benchmarks detrás de cada afirmación de velocidad de abajo

Contenidos

Parte I — Los conceptos *(sin código; lee esto primero)*

> **Si solo vas a leer tres secciones:** §7 y §8 (backpropagation resuelto a mano) y §21 (cómo sabes que es correcta).
>
> **Dónde se comprueban las afirmaciones.** Cada afirmación de rendimiento de la Parte II enlaza a una medición en `bench/README.es.md`, incluida una que resultó ser falsa y que se corrige en §11. Los números de esta guía provienen de un Apple M3 Pro con .NET 10; véase la nota al principio de §22 sobre por qué los tuyos pueden diferir en los últimos dígitos.

Parte I — Los conceptos

1. ¿Qué problema resuelve realmente una red neuronal?

Tienes ejemplos de entradas emparejadas con las salidas correctas:

Entrada (x)	Salida correcta (y)
0, 0	0
0, 1	1
1, 0	1
1, 1	0

Quieres una función que reproduzca esta correspondencia — y, más importante, que generalice a entradas que no ha visto. No conoces la fórmula. Así que en lugar de escribir una:

1. Construyes una función con **miles de números ajustables** dentro (los *parámetros*).
2. Defines una medida de cuán equivocadas están sus salidas (la *pérdida*).
3. Averiguas, para cada parámetro, en qué dirección moverlo para reducir la pérdida.
4. Los mueves todos un poco en esa dirección, y repites.

Esa es toda la idea. Al final de esa lista se asocian dos nombres, y conviene distinguirlos desde el principio porque se confunden constantemente:

- Backpropagation es el paso 3 — *calcular* los gradientes de forma eficiente. Es la parte ingeniosa, y aquello que este código existe para demostrar.
- El **descenso de gradiente** es el paso 4 — *aplicarlos*.

Backpropagation te dice hacia dónde está la bajada; el descenso de gradiente da el paso. Todo lo demás en este repositorio es contabilidad y velocidad.

> La red no está «razonando». Es una fórmula grande y parametrizada que se ajusta a los datos, del mismo modo que ajustarías una recta a unos puntos — solo que con muchos más parámetros y una forma no lineal.

2. La neurona

Una sola neurona (aquí: una **unidad**) hace exactamente tres cosas.

Paso 1 — suma ponderada. Cada entrada recibe un peso que indica cuánto importa, positivo o negativo. Se suman todos, más un sesgo:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

Piénsalo como una **votación**. Cada entrada empuja el resultado hacia arriba o hacia abajo en proporción a su peso. El sesgo `b` es la línea base de la neurona — su opinión antes de ver ninguna entrada. Sin sesgo, `z` sería siempre 0 cuando todas las entradas son 0, lo cual es una restricción innecesaria.

Paso 2 — activación. Se pasa `z` por una función no lineal `g`:

$$a = g(z)$$

Paso 3 — salida `a`, que se convierte en entrada de la capa siguiente.

Por qué la activación no es opcional

Este es el punto que más se omite, y es importante.

Supón que te saltas la activación, de modo que cada capa es solo una suma ponderada. Apila dos capas: la capa 2 calcula una suma ponderada de las salidas de la capa 1, que a su vez son sumas ponderadas de las entradas. Una suma ponderada de sumas ponderadas es… **sigue siendo una simple suma ponderada**. Algebraicamente puedes colapsar las dos capas en una:

$$W_2 (W_1 x) = (W_2 W_1) x = W_{\text{combinada}}\ x$$

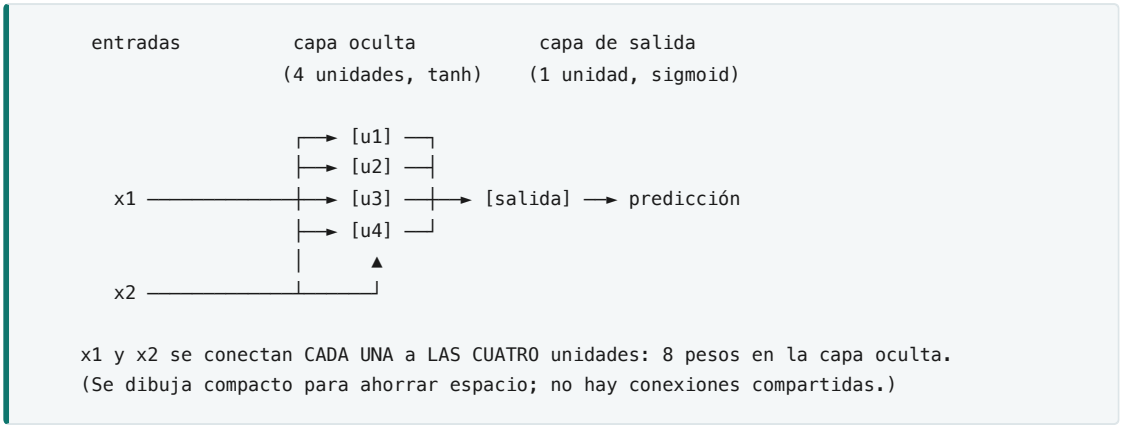
Así que cien capas lineales apiladas tienen exactamente el poder de una sola capa lineal — solo pueden trazar rectas. La `g` no lineal entre capas es lo que impide el colapso y es toda la razón por la que la profundidad te aporta algo.

La no linealidad es lo que hace profunda a una red «profunda». Quítala y la profundidad es decoración.

---

3. Capas y la red

Una **capa** es un grupo de unidades que ven todas las mismas entradas pero tienen sus propios pesos, de modo que cada una aprende a detectar algo distinto.



Esa es exactamente la red XOR de `Program.cs`.

- **Densa / totalmente conectada** significa que cada entrada se conecta con cada unidad. Una capa con `n` entradas y `j` unidades tiene por tanto `n × j` pesos más `j` sesgos.
- La **capa oculta** es «oculta» solo en el sentido de que sus salidas nunca se observan directamente — son valores intermedios que alimentan la capa siguiente.
- Los datos fluyen de izquierda a derecha en el **forward pass**; la información del error fluye de derecha a izquierda en el **backward pass**.

¿Qué aprenden realmente las unidades ocultas? Nadie les asigna un trabajo. El entrenamiento descubre que cierta característica intermedia es útil y una unidad acaba calculándola. Ese es el trato que ofrece la profundidad: **aprender representaciones intermedias útiles, y luego resolver un problema fácil sobre ellas.**

Los libros de texto suelen decir que las unidades ocultas de XOR aprenden puertas lógicas limpias como OR y AND. Esto es lo que las cuatro unidades de *esta* red entrenada calculan realmente (medido, no supuesto — el ejercicio 7 te enseña a imprimirlo tú mismo):

```
x1,x2      u1       u2       u3       u4
0,0      0.532   -0.570    0.753   -0.342
0,1     -0.697    0.762   -0.963    0.991
1,0      0.994   -0.997   -0.977   -0.874
1,1      0.903   -0.926   -1.000    0.939

pesos de la capa de salida:
        -2.243    2.737   -4.753   -4.449
```

Léelo y la historia demasiado ordenada se desmorona:

- **u1 y u2 son imágenes especulares casi perfectas** (u2 ≈ −u1 en cada fila). Dos unidades aprendieron la misma característica con signo opuesto —

pura redundancia. Sus pesos de salida ($-2{,}24$ y $+2{,}74$) también tienen signos opuestos, así que se refuerzan en lugar de cancelarse, actuando como una sola característica con un peso efectivo cercano a -5.

- **u3 es aproximadamente NOR** — fuertemente positiva solo cuando ambas entradas son 0.
- **u4 sigue aproximadamente a x2** — positiva siempre que x2 = 1.

Así que no es OR-y-AND. Es *una* descomposición válida entre muchas, encontrada por descenso de gradiente desde un punto de partida aleatorio, y otra semilla produce otra distinta.

Esta es la lección más útil. Las representaciones aprendidas suelen ser redundantes, parcialmente duplicadas y solo vagamente interpretables. Leer significado en unidades ocultas individuales es todo un área de investigación (*interpretabilidad*) precisamente porque es genuinamente difícil — la red no tiene obligación alguna de organizarse como lo haría un humano.

---

4. Aprender = descenso de gradiente

Ahora la pregunta central: *¿cómo* ajustas miles de parámetros en la dirección correcta?

La imagen de bajar la colina

Imagina la pérdida como un paisaje. Cada parámetro es una dirección en la que puedes moverte, y la altitud es cuán equivocada está la red. Quieres el valle más bajo. Estás en una niebla espesa y solo puedes sentir la pendiente bajo tus pies.

La estrategia: **detecta hacia dónde baja la pendiente, da un paso pequeño y repite.**

Eso es el descenso de gradiente. El **gradiente** es la pendiente — y el cálculo nos la da exactamente, sin necesidad de adivinar:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

Léelo así: *peso nuevo = peso viejo − learning rate × pendiente de la pérdida respecto a ese peso.*

Por qué el signo menos

La derivada `∂L/∂w` responde: «si aumento `w` ligeramente, ¿la pérdida sube o baja?»

- Pendiente **positiva** → aumentar `w` aumenta la pérdida → así que **reduce** `w`.
- Pendiente **negativa** → aumentar `w` reduce la pérdida → así que **aumenta** `w`.

Ambos casos se resuelven restando el gradiente. El signo menos es lo que lo convierte en *descenso*.

El learning rate η

Cuán grande es el paso.

- **Demasiado bajo** → el entrenamiento se arrastra; miles de épocas desperdiciadas.
- **Demasiado alto** → te pasas del valle, rebotas por la ladera opuesta y la pérdida explota a `NaN`.
- **Adecuado** → descenso constante.

No hay fórmula para elegirlo. De `0.1` a `0.5` es un rango de partida razonable para una red pequeña como esta. Si tu pérdida llega alguna vez a `NaN` o infinito, **baja primero el learning rate** — es la causa en la inmensa mayoría de los casos.

Una advertencia crucial

La derivada de cada parámetro supone que **todo lo demás permanece fijo**. Eso solo es cierto para un paso infinitesimal. Da un paso grande y todos los parámetros se mueven a la vez, y el terreno que mediste ya no es el terreno sobre el que estás. Esto es exactamente por qué los pasos deben ser pequeños y por qué el proceso es iterativo en lugar de resolverse de un tirón.

---

5. Medir el error: la función de pérdida

La pérdida convierte «cuán equivocada estaba esta predicción» en un único número a minimizar.

MSE (error cuadrático medio), el que usa esta implementación por defecto:

$$L = \frac{1}{m}\sum_j (a_j - y_j)^2$$

Toma la diferencia entre predicción y objetivo, elévala al cuadrado, promedia sobre las salidas. Elevar al cuadrado hace dos cosas útiles: vuelve positivos los errores (de modo que $+0{,}3$ y $-0{,}3$ son igual de malos en lugar de cancelarse) y castiga desproporcionadamente los errores grandes.

Su derivada — la semilla de todo el backward pass:

$$\frac{\partial L}{\partial a_j} = \frac{2(a_j - y_j)}{m}$$

Razonable: si la predicción supera al objetivo, la derivada es positiva, lo que significa «baja esta salida».

MSE es la elección natural para regresión (predecir números). Para clasificación, la **cross-entropy** es mejor, y este código la implementa — véase §27.

---

6. La regla de la cadena — la única pieza de cálculo que necesitas

Backpropagation es la regla de la cadena aplicada con cuidado. Si entiendes esta sección, el resto es mecánico.

La regla: si `a` depende de `z`, y `z` depende de `w`, entonces

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

La intuición: las tasas de cambio se multiplican a lo largo de una cadena. Si un coche es el doble de rápido que una bicicleta, y la bicicleta tres veces más rápida que caminar, el coche va a seis veces la velocidad de caminar. La misma composición se aplica a «cuánto mueve a `L` el empujar `w`».

En una red neuronal la cadena es exactamente:

```
w ⟶ z ⟶ a ⟶ (next layer) ⟶ ... ⟶ L
```

de modo que la influencia de cualquier peso sobre la pérdida final es el producto de todas las tasas de cambio locales a lo largo del camino desde ese peso hasta la pérdida.

Los tres factores en nuestro caso son todos sencillos:

Factor	Qué es	Valor
∂L/∂a	Cuánto cambia la pérdida con la salida de esta unidad	Viene de la pérdida, o de la capa superior
∂a/∂z	Cuánto cambia la salida con la suma ponderada	`g'(z)` — la derivada de la activación
∂z/∂w	Cuánto cambia la suma ponderada con este peso	`x` — ¡simplemente la entrada!

Vale la pena detenerse en el último. Puesto que `z = w₁x₁ + w₂x₂ + b`, la derivada respecto a `w₁` es sencillamente `x₁`. **El gradiente de un peso es proporcional a la entrada que multiplica.** Lo cual es intuitivo: un peso conectado a una entrada que valía cero no tuvo efecto en esta predicción, así que no recibe culpa ni actualización.

---

7. Ejemplo resuelto: una neurona, a mano

Hagamos un `forward pass` y un `backward pass` completos con números reales. *Estos valores están calculados y verificados — sigue el hilo con una calculadora.*

Planteamiento. Una neurona, dos entradas, activación sigmoide.

```
entradas              x = [1.0, 0.5]
pesos                 w = [0.3, −0.2]
sesgo                 b = 0.1
objetivo              y = 1.0
learning rate   η = 0.5
```

> **Una nota sobre el `/m`.** El §5 dio la derivada de la pérdida como `2(a − y)/m`, donde `m` es el número de salidas. Esta red tiene una sola salida, así que `m = 1` y la división desaparece. Por eso los ejemplos de abajo usan simplemente `2(a − y)` — no están omitiendo un término, es que `m` resulta ser 1. El código de `Network.cs` siempre divide correctamente.

Forward pass

Suma ponderada:

```
z = (0.3 × 1.0) + (−0.2 × 0.5) + 0.1
  = 0.3 − 0.1 + 0.1
  = 0.3
```

Activación (sigmoide: `1 / (1 + e⁻ᶻ)`):

```
a = 1 / (1 + e^−0.3) = 0.574443
```

Pérdida:

```
L = (a − y)² = (0.574443 − 1.0)² = (−0.425557)² = 0.181099
```

La red predijo 0,574 donde se quería 1,0. Ahora lo corregimos.

Backward pass

Paso 1 — ¿cómo cambia la pérdida con la salida?

```
dL/da = 2(a − y) = 2 × (0.574443 − 1.0) = −0.851115
```

Negativa, lo que significa: *aumentar* `a` *reduciría la pérdida.* Correcto — queremos que `a` crezca hacia 1,0.

Paso 2 — propagar el gradiente hacia atrás a través de la activación. La derivada de la sigmoide es `a(1−a)`:

```
g'(z) = a(1 − a) = 0.574443 × 0.425557 = 0.244458

delta = dL/da × g'(z) = −0.851115 × 0.244458 = −0.208062
```

`delta` (δ) es el valor intermedio más importante de todo backpropagation. Significa: **«cuánto cambia la pérdida por unidad de cambio en la `z` de esta neurona».** Una vez que tienes δ para una neurona, cada gradiente en el que participa está a una multiplicación de distancia.

Paso 3 — los gradientes de los parámetros. Multiplica δ por cada entrada:

```
dL/dw₀ = delta × x₀ = −0.208062 × 1.0 = −0.208062
dL/dw₁ = delta × x₁ = −0.208062 × 0.5 = −0.104031
dL/db  = delta × 1   = −0.208062
```

Fíjate en que el gradiente de `w₁` es **la mitad** del de `w₀` — porque su entrada era 0,5 en lugar de 1,0. Tuvo la mitad de influencia, así que recibe la mitad de la corrección. El gradiente del sesgo es simplemente δ, ya que un sesgo es un peso sobre una entrada constante de 1.

La actualización

Resta `η × gradiente` de cada parámetro:

```
w₀ = 0.3  − 0.5 × (−0.208062) =  0.404031
w₁ = −0.2 − 0.5 × (−0.104031) = −0.147984
b  = 0.1  − 0.5 × (−0.208062) =  0.204031
```

¿Funcionó?

Ejecuta el forward pass otra vez con los parámetros nuevos:

```
z = (0.404031 × 1.0) + (−0.147984 × 0.5) + 0.204031 = 0.534070
a = sigmoid(0.534070) = 0.630432
L = (0.630432 − 1.0)² = 0.136581
```

La pérdida bajó de 0,181099 a 0,136581, y la predicción se movió de 0,574 a 0,630 — más cerca del objetivo de 1,0.

Eso es un paso completo de entrenamiento. Una red es esto repetido para cada neurona y cada ejemplo, miles de veces.

---

8. Ejemplo resuelto: la cadena a través de dos capas

Lo anterior trataba una sola capa. La idea nueva y esencial en una red *profunda* es propagar el gradiente **hacia la capa anterior**. Aquí está el caso más pequeño posible, también verificado.

Planteamiento. Una entrada → una unidad oculta tanh → una unidad de salida sigmoide.

```
x = 0.8      w₁ = 0.5, b₁ = 0.0   (oculta, tanh)
             w₂ = 1.2, b₂ = −0.3  (salida, sigmoid)
objetivo y = 0.0
```

Hacia adelante

```
z₁ = 0.5 × 0.8 + 0.0 = 0.400000
h  = tanh(0.4)       = 0.379949      ← salida de la unidad oculta

z₂ = 1.2 × 0.379949 − 0.3 = 0.155939
a  = sigmoid(0.155939)    = 0.538906  ← predicción final

L  = (0.538906 − 0.0)² = 0.290420
```

Hacia atrás — primero la capa de salida

```
dL/da  = 2(a − y) = 2 × 0.538906 = 1.077812
sig'   = a(1−a)   = 0.538906 × 0.461094 = 0.248486
delta₂ = 1.077812 × 0.248486 = 0.267821

dL/dw₂ = delta₂ × h = 0.267821 × 0.379949 = 0.101758
dL/db₂ = delta₂                           = 0.267821
```

Hacia atrás — ahora cruzamos a la capa oculta

Este es el paso que lo convierte en `backpropagation`. La unidad oculta influyó en la pérdida solo *a través de* la unidad de salida, así que enviamos el gradiente de vuelta por el peso que las conecta, `w₂`:

```
dL/dh = delta₂ × w₂ = 0.267821 × 1.2 = 0.321386
```

Léelo así: *«la salida de la unidad oculta afecta a la pérdida en δ₂ escalado por la fuerza de la conexión usada durante el `forward pass`»*. El mismo peso `w₂` que lleva la señal hacia adelante lleva ahora el gradiente hacia atrás. Esa simetría es el corazón del algoritmo.

Después es la receta idéntica a la anterior, una capa más abajo:

```
tanh'  = 1 − h² = 1 − 0.144361 = 0.855639
delta₁ = dL/dh × tanh' = 0.321386 × 0.855639 = 0.274990

dL/dw₁ = delta₁ × x = 0.274990 × 0.8 = 0.219992
dL/db₁ = delta₁                      = 0.274990
```

El patrón

Cada capa, sin excepción, hace:

1. Recibir `dL/da` de arriba (o de la pérdida, si es la última capa).
2. `δ = dL/da × g'(z)` — empujar a través de la activación.
3. `dL/dW += δ × entrada`, `dL/db += δ` — registrar los gradientes de los parámetros.
4. `dL/dentrada = δ × W` — entregar el gradiente a la capa anterior, que vuelve al paso 1.

Con 2 capas o con 200, ese bucle es todo el algoritmo. `Dense.Backward` es una transcripción literal de esas cuatro líneas.

Por qué es eficiente

De forma ingenua podrías calcular el gradiente de cada peso por separado re-ejecutando la red — con un millón de pesos, eso es un millón de forward passes. Backpropagation calcula **todos** los gradientes en un único `backward pass` reutilizando los valores δ compartidos. Cuesta aproximadamente lo mismo que un forward pass. Esa eficiencia es la razón por la que las redes neuronales son entrenables siquiera, y por la que la popularización de la backpropagation en 1986 relanzó todo el campo.

---

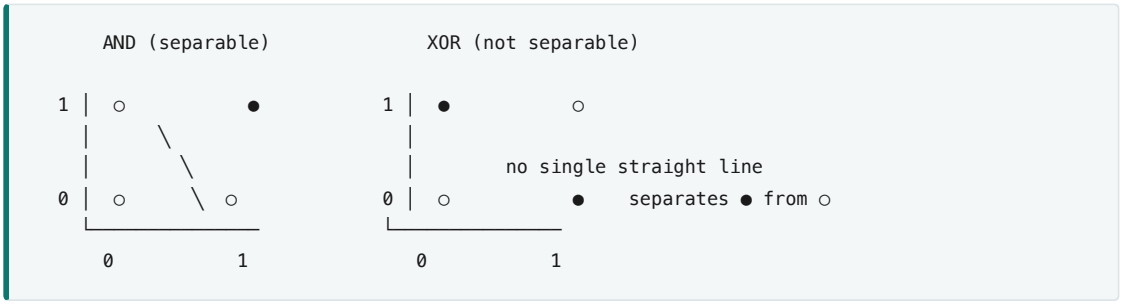
9. El perceptrón, XOR y por qué importa esta historia

El **perceptrón** (1958) es el antepasado de todo esto: una unidad, activación escalón y su propia regla de actualización.

$$w \mathrel{+}= \eta\,(y - \hat{y})\,x$$

Puesto que la función escalón solo produce 0 o 1, el error `(y - ŷ)` es siempre -1, 0 o $+1$: «demasiado alto», «correcto» o «demasiado bajo». `Perceptron.Train` se salta por completo las predicciones correctas y se detiene anticipadamente en cuanto pasa una época entera sin actualizaciones.

El teorema de convergencia: si los datos son *linealmente separables* — separables por una sola línea recta — está garantizado que converge en un número finito de pasos. Si no lo son, oscila para siempre.



El libro de Minsky y Papert de 1969, *Perceptrons*, analizó esta limitación con rigor, y ampliamente se le atribuye haber enfriado el entusiasmo por las redes neuronales durante los años 70.

Toma esa historia con cierta cautela — se repite en todas partes de forma simplificada. Minsky y Papert sí discutieron las redes multicapa en lugar de ignorarlas, y los historiadores suelen atribuir el colapso más amplio de la financiación

(el «invierno de la IA») más bien al informe Lighthill de 1973 y a las expectativas incumplidas en la IA en su conjunto. El resultado matemático es sólido; la narrativa ordenada de causa y efecto a su alrededor es discutida.

La solución son las dos cosas que añade este código: una capa oculta (para que la red pueda construir sus propias características) y una activación suave (para que backpropagation pueda calcular gradientes a través de ella).

Puedes ver toda la historia ejecutarse en `Program.cs`: el perceptrón clava AND en 4 épocas, y la red con una capa oculta resuelve el XOR que el perceptrón demostrablemente no puede.

Parte II — La implementación en C#

10. Orden de lectura

Si vas a leer el código por primera vez, hazlo en este orden:

1. `Activations.cs` — las activaciones. Pequeño, autocontenido, y cada una se corresponde con la tabla del §13.
2. `Dense.cs` → `Forward` — las matemáticas del §2, literalmente.
3. `Dense.cs` → `Backward` — el patrón de cuatro pasos del §8, literalmente.
4. `Network.cs` — el bucle de entrenamiento que lo mueve todo.
5. `SimdOps.cs` — optimización pura. **Sáltalo en la primera pasada**; no cambia nada conceptualmente.

11. Características de C# usadas aquí que quizá no conozcas

Estas son las construcciones que hacen que el código parezca poco familiar. Ninguna es esencial para las matemáticas.

`Span<float>` — una ventana a la memoria

```
public void Forward(ReadOnlySpan<float> aIn, Span<float> aOut)
```

Un `Span<float>` es una **vista** de un fragmento de un arreglo — un puntero más una longitud. No copia nada. `weights.AsSpan(10, 4)` se refiere a los elementos 10–13 del arreglo original; escribir a través del span escribe en el original.

Por qué está por todas partes aquí: el entrenamiento llama a `Forward` millones de veces. Devolver un `float[]` nuevo en cada llamada asignaría millones de arreglos y te enterraría en recolección de basura. Los spans permiten reutilizar para siempre un único buffer preasignado. `ReadOnlySpan` además documenta «esto solo lo leo».

La pega: un span no puede almacenarse en un campo de clase ni usarse dentro de métodos `async` (es un `ref struct`, solo de pila). Por eso las capas mantienen campos `float[]` y reparten spans.

Structs genéricos como política — `Dense<TActivation>`

```
public sealed class Dense<TActivation> : ILayer where TActivation : IActivation
```

La activación es un **parámetro de tipo**, no un campo. `Dense<Tanh>` y `Dense<Sigmoid>` son tipos distintos, y el JIT compila código máquina separado para cada uno, integrando la activación directamente dentro del bucle.

La alternativa obvia es un campo `Func<float, float>`. Cuesta una llamada indirecta por cada unidad de cada capa de cada ejemplo, y el JIT no puede integrarla dentro del bucle.

> **Un borrador anterior de esta guía te decía que eso la hacía más lenta. No lo hace, y la corrección es más instructiva que la afirmación original.** Comparada mediante un benchmark con una capa que usa delegados, la versión genérica es más rápida entre un 0 y un 6 % — ruido a cualquier tamaño realista (tabla completa).
>
> La razón es una proporción, no un coste de despacho. La activación se ejecuta **una vez por unidad**; el producto escalar que la alimenta ejecuta `Inputs` multiplicaciones-sumas por unidad. En una capa de 784 entradas, una llamada que no se puede integrar queda amortizada entre 784 multiplicaciones-sumas fusionadas — es invisible porque es *rara*, no porque sea rápida. La primera sospecha fue que tanh, una función trascendente, estaba ocultando la llamada, así que la prueba se repite con ReLU (una comparación y una selección). El resultado apenas se mueve.
>
> **La lección se generaliza más allá de este repositorio:** «llamada indirecta que el JIT no puede integrar» es una afirmación sobre la generación de código, y «más lenta» es una afirmación sobre el programa. Pasar de una a la otra exige saber con qué frecuencia ocurre la llamada respecto a todo lo demás — que es lo que te dice un profiler y no la intuición. Esto es también el §12 en miniatura: los efectos que importan son los del bucle más interno, y la activación no está en él.

Así que conserva el diseño genérico, pero por sus méritos reales: las activaciones `readonly struct` se componen a coste cero, nada se asigna en el montón, y el sistema de tipos impide confundir `Dense<Tanh>` con `Dense<ReLU>`. No

por velocidad.

La característica que lo habilita son los **miembros de interfaz estáticos abstractos** (C# 11):

```
public interface IActivation
{
    static abstract float Apply(float z);
}
```

Un método de interfaz sin ningún objeto asociado. Permite que `TActivation.Apply(z)` se resuelva e integre en tiempo de compilación/JIT.

Activaciones `readonly struct`

`Sigmoid`, `Tanh` y las demás son structs vacíos. No contienen datos y nunca se instancian — existen puramente como *nombres* que alimentan el sistema de genéricos. Coste nulo en tiempo de ejecución.

El JIT

C# compila a IL (bytecode), y el compilador JIT convierte el IL en código máquina en tiempo de ejecución. Integra métodos pequeños, elige instrucciones SIMD para la CPU concreta y — algo importante aquí — genera una **copia de código distinta por cada instanciación genérica con un tipo por valor**. Eso es lo que hace rápido a `Dense<Tanh>`, y también por qué el §14 mantiene los auxiliares SIMD *fuera* de la clase genérica.

---

12. Disposición de los datos — la decisión más trascendente

NumPy almacena `W` como `(n, j)`: **n características × j unidades**. Los pesos de la unidad `j` son la *columna* `W[:, j]`, cuyos elementos están separados `n` floats en memoria.

Esta versión en C# almacena la **transpuesta**, aplanada en un único arreglo:

```
Weights = [ unit0: w00 w01 w02 | unit1: w10 w11 w12 | unit2: ... ]
           └── Inputs floats ──┘
```

Los pesos de la unidad `j` son `Weights[j * Inputs .. (j+1) * Inputs]` — **contiguos**. Véase `Dense.UnitWeights`.

Por qué importa tanto

Una CPU nunca carga un solo float. Carga una **línea de cache de 64 bytes** (16 floats) cada vez, porque la memoria es enormemente más lenta que la aritmética — una lectura de memoria principal cuesta cientos de ciclos mientras que una multiplicación cuesta uno.

- **Acceso disperso (columna de NumPy):** cargas 16 floats, usas 1, descartas 15. Desperdicias ~94 % del ancho de banda de memoria, y SIMD no puede usarse sin una costosa instrucción `gather`.
- **Acceso contiguo:** cargas 16 floats, usas los 16, y una sola instrucción SIMD procesa entre 4 y 16 de ellos a la vez (§15).

> **Regla práctica:** organiza los datos de modo que el bucle más interno recorra direcciones consecutivas. Este único hábito vale más que todas las demás optimizaciones de este archivo juntas.

Cuánto vale realmente — y dónde no vale nada

Esa regla práctica es una afirmación fuerte, así que está medida. Mismos pesos, misma aritmética, misma activación; la única diferencia es el orden en memoria (detalles):

Forma de la capa	Orden por unidad (contiguo)	Orden por característica (disperso)	Coste de dispersar
2 × 4 — la capa XOR	19,1 ns	16,0 ns	**0,84× — ¡gana el disperso!**
64 × 64	525 ns	3249 ns	**6,2×**
784 × 128 — tamaño MNIST	9,75 μs	72,6 μs	**7,4×**

A tamaños realistas la afirmación se sostiene con holgura: 6,2–7,4×, el mayor efecto individual del código, y mayor que la ganancia de SIMD que en parte habilita.

Pero mira la primera fila. En la capa XOR la disposición «mala» es *más rápida*. Ocho pesos son 32 bytes — caben dentro de una sola línea de cache de 64 bytes, así que no hay ancho de banda desperdiciado que ahorrar ni recolección

que evitar. Lo único que queda es la preparación del bucle de la ruta SIMD, que el bucle disperso simple se salta.

Vale la pena detenerse en esto, porque la demo XOR es lo primero que ejecutas:

> **Todas las optimizaciones de esta guía no valen nada a la escala del ejemplo que las introduce.** El comportamiento de la cache es una propiedad de los datos que no caben en cache. La vectorización es una propiedad de los bucles lo bastante largos como para amortizar su preparación. Por debajo de esos umbrales estás midiendo `overhead`, y la implementación «obviamente peor» gana con frecuencia.
>
> El corolario es el útil: **el tamaño al que mides determina la respuesta que obtienes.** Un benchmark de la capa XOR te habría dicho que borrases la disposición contigua.

Un único arreglo plano, no un arreglo de arreglos. `float[]` gana a `float[][]` aquí: una asignación en lugar de `j`, una comprobación de límites en lugar de dos, sin perseguir punteros por unidad, y toda la matriz reside en un bloque contiguo que el prefetcher puede recorrer. (No medido por separado: la variante `float[][]` mezclaría el efecto de la disposición con el de las asignaciones, y el arreglo plano no es más difícil de escribir.)

---

13. Las activaciones en código

Activación	g(z)	g'(z) en términos de a = g(z)	Rango	Se usa para
Sigmoid	$1 / (1 + e^{-z})$	`a(1 - a)`	(0, 1)	Capa de salida, probabilidad binaria
Tanh	tanh(z)	`1 - a²`	(−1, 1)	Capas ocultas
ReLU	max(0, z)	`a > 0 ? 1 : 0`	[0, ∞)	Capas ocultas, redes profundas
Identity	z	`1`	ℝ	Salida de regresión
Step	z ≥ 0 ? 1 : 0	*lanza excepción*	{0, 1}	Solo el perceptrón

Por qué las derivadas se escriben en términos de `a`, no de `z`

Todas las activaciones aquí tienen una derivada recuperable a partir de **su propia salida**. La derivada de la sigmoide es `a(1−a)`; la de tanh es `1−a²`. El forward pass ya calculó `a`, así que el backward pass lo reutiliza en vez de recalcular `exp()` o `tanh()`.

Este es un ahorro real, no una microoptimización: las funciones trascendentes cuestan entre 20 y 100 veces una multiplicación. Es también la razón por la que `Forward` guarda sus salidas en cache — véase §14.

Por qué `Step` lanza una excepción en lugar de devolver 0

Su derivada es 0 en todos los puntos donde está definida (la función es plana), e indefinida en el salto. Devolver 0 multiplicaría por cero todos los gradientes de la red, y el entrenamiento parecería ejecutarse mientras no aprende **nada** — un error silencioso y enloquecedor.

Lanzar una excepción hace la lección explícita: **la planitud de la función escalón es precisamente por lo que el perceptrón necesita su propia regla de actualización, y por lo que la backpropagation exige activaciones suaves.** El descenso de gradiente necesita una pendiente que seguir; una escalera no tiene ninguna.

Cómo elegir una

- **Capas ocultas → tanh o ReLU.** Tanh está centrada en cero, es decir, sus salidas abarcan $(-1, 1)$. Las salidas de la sigmoide son todas positivas, lo que hace que todos los gradientes de peso que entran en una unidad compartan signo, de modo que la optimización zigzaguea en lugar de ir recta cuesta abajo.
- **Capa de salida → ajústala a la tarea.** Sigmoide para una probabilidad en (0,1); identidad para regresión sin cotas.
- **Sigmoide y tanh se saturan:** para $|z|$ grande la curva se aplana, `g'(z) ≈ 0`, y los gradientes casi se desvanecen. Apila muchas capas así y el gradiente que llega a las primeras capas es efectivamente cero — el **problema del gradiente que se desvanece**. La derivada de ReLU es exactamente 1 para entradas positivas, razón por la cual las redes profundas se pasaron a ella.

---

14. El forward pass en código

`Dense.Forward`:

```
for (int j = 0; j < Units; j++)
{
    float z = SimdOps.Dot(w.Slice(j * Inputs, Inputs), aIn) + Bias[j];
    aOut[j] = TActivation.Apply(z);
}
```

Línea por línea, esto es el §2: recorta los pesos de la unidad, haz el producto escalar con la entrada, suma el sesgo, activa. Compáralo con el NumPy original y la correspondencia es exacta — solo cambió la disposición.

Los dos forward passes, y el error que las separó

Backpropagation necesita tanto las entradas que produjeron estas activaciones (para `dL/dW = δ × entrada`) como las activaciones mismas (para `g'` a partir de `a`). Así que algo tiene que guardarlas en una cache, y el lugar obvio es el final de `Forward`:

```
aIn.CopyTo(_lastInput);      // lo que hacía antes el código
aOut.CopyTo(_lastOutput);
```

Eso es una trampa, y merece la pena entenderla porque la misma forma aparece en todas partes. `Forward` escribe la cache, pero `Backward` la lee mucho después. Cualquier llamada intermedia que ejecute un `forward pass` la sobrescribe. Y hay muchas llamadas razonables que lo hacen:

```
net.AccumulateGradients(x, y);
Console.WriteLine(net.Predict(somethingElse)[0]);   // ← destruye la cache en silencio
net.ApplyGradients(lr, 1);                          //   los gradientes ahora describen el
        ejemplo equivocado
```

No se lanza ninguna excepción. La pérdida sigue bajando. La red simplemente aprende algo sutilmente incorrecto — que es exactamente el modo de fallo para el que existe el §21, llegando por otra vía.

La solución es hacer que ese estado inválido no pueda expresarse, en lugar de documentarlo. Ahora hay dos métodos:

	¿guarda en cache?	usado por
`Forward`	no	`Predict`, `Loss`, `ForwardBatch`, `GradientCheck` — todo es inferencia
`ForwardTrain`	sí	solo `AccumulateGradients`

`Backward` consume la cache y la limpia, de modo que un segundo `Backward`, o uno sin un `ForwardTrain` previo, lanza una excepción en lugar de derivar calladamente un ejemplo obsoleto. `GradientCheck` depende directamente de esto: evalúa la pérdida dos veces por parámetro mientras los gradientes analíticos están en los acumuladores, lo cual solo es seguro porque `Loss` no puede tocar el estado de entrenamiento.

> **La lección general:** cuando el *efecto secundario* de un método lo consume otro método más tarde, el acoplamiento es invisible en ambos puntos de llamada. Prefiere dividir el método antes que documentar el orden. Un comentario que diga «no llames a `Predict` aquí» es un error esperando a alguien que no lo leyó.

El coste en memoria. La cache es también la razón por la que entrenar necesita mucha más memoria que inferir: **no puedes liberar las activaciones del forward pass hasta que el backward pass las haya consumido.** Eso es lo que significa en la práctica «batch size demasiado grande» — sin memoria, sosteniendo a la vez las activaciones de todos los ejemplos.

---

15. SIMD — hacer varias multiplicaciones a la vez

Sáltate esta sección en una primera lectura. Es velocidad pura; las matemáticas no cambian.

SIMD = *Single Instruction, Multiple Data* (una instrucción, múltiples datos). Las CPU modernas tienen registros anchos que contienen varios floats a la vez, y una instrucción opera sobre todos ellos simultáneamente.

Cuántos depende de tu CPU:

Hardware	`Vector<float>.Count`
Apple Silicon / ARM (NEON)	**4**
x86 con AVX2	8
x86 con AVX-512	16

`Vector<float>` de `System.Numerics` expone esto de forma portátil — escribes el código una vez y se adapta. Comprueba tu propia máquina con:

```
Console.WriteLine(System.Numerics.Vector<float>.Count);
```

Esto es justamente por lo que el código nunca fija un ancho: cada bucle de `SimdOps.cs` lee `Vector<float>.Count` en tiempo de ejecución, de modo que el mismo código fuente se ejecuta a ancho completo tanto en ARM como en x86.

> **Malentendido común, y el que inició la reescritura de este proyecto:** `Vector<float>` **no** es un vector matemático de longitud variable. Es un registro de hardware de ancho fijo. No puede almacenar los pesos de una capa — es un pipeline por el que haces fluir datos, `Count` floats cada vez.

El producto escalar (`SimdOps.Dot`)

```
var acc0 = Vector<float>.Zero;
var acc1 = Vector<float>.Zero;

for (; i <= n - 2 * width; i += 2 * width)
{
    acc0 += new Vector<float>(a.Slice(i, width))         * new Vector<float>(b.Slice(i,
        width));
    acc1 += new Vector<float>(a.Slice(i + width, width)) * new Vector<float>(b.Slice(i +
        width, width));
}
// ... luego un bucle de un vector, una suma horizontal y una cola escalar
```

Por qué dos acumuladores. Una instrucción de multiplicación-suma tiene ~4 ciclos de *latencia* pero puede *emitirse* cada ciclo. Con un solo acumulador, cada iteración debe esperar el resultado de la anterior — obtienes un resultado cada 4 ciclos, la cuarta parte del máximo. Dos cadenas independientes permiten a la CPU mantener cuatro operaciones en vuelo. Esto es **paralelismo a nivel de instrucción**, y cuesta un registro extra.

La cola escalar se ocupa de una longitud que no es múltiplo del ancho SIMD. Todo bucle SIMD escrito a mano necesita una; olvidarla descarta silenciosamente los últimos elementos.

¿Funciona algo de esto?

Ambas afirmaciones anteriores — que vectorizar compensa, y que el segundo acumulador compensa *otra vez* — están medidas contra un bucle escalar y contra una versión SIMD de un solo acumulador (detalles):

Longitud	Escalar	1 acumulador	2 acumuladores	Ganancia SIMD	Ganancia del 2.º acumulador
8	4,23 ns	1,27 ns	1,03 ns	4,1×	**1,23×**
64	38,3 ns	9,92 ns	7,91 ns	4,8×	**1,25×**
512	360 ns	82,4 ns	63,5 ns	5,7×	**1,30×**
4096	2937 ns	723 ns	500 ns	5,9×	**1,45×**

Tres cosas que notar.

La ganancia de SIMD supera el ancho del vector. `Vector<float>` solo tiene ancho 4 en la máquina ARM de la que salen estos números, y sin embargo la aceleración llega a 5,9×. Vectorizar no solo hace 4 multiplicaciones a la vez — también reduce a la cuarta parte la contabilidad del bucle y las comprobaciones de límites. Obtener *más* que el ancho es normal; el ancho es un suelo, no un techo.

El segundo acumulador se gana su sitio en cualquier longitud útil. Con 1,2–1,5× es la segunda mayor optimización del código tras la disposición de datos, lo cual es mucho para un registro extra.

> **Una nota al pie que vale más que la tabla.** Una revisión anterior de esta guía daba esa primera fila como **0,83× — el segundo acumulador costando un 17 %** — y dedicaba un párrafo a explicar por qué ocho floats son exactamente la longitud a la que debería perder. La explicación era plausible y el número era ruido: aquellas filas se habían medido con el `--job short` de BenchmarkDotNet, de tres iteraciones, cuyo margen de error era más ancho que el efecto que se pretendía describir. Bien medida, gana 1,23×, como todas las demás filas.
>
> Dos lecciones, y la segunda es la cara. **Un job corto sirve para triaje, no para conclusiones.** Y **un mecanismo convincente no es evidencia**: la historia sobre costes de preparación e iteraciones de cola resultaba tan persuasiva que nadie volvió a ejecutar la medición que se había inventado para explicar.

`AddScaled` — `dest += src × scale`

A diferencia de `Dot`, este no está escrito a mano: llama a `TensorPrimitives.MultiplyAdd` de `System.Numerics.Tensors`, que es **2,5× más rápido** que el bucle `Vector<float>` al que sustituyó. `Dot`, en cambio, conserva su propio bucle porque `TensorPrimitives.Dot` midió **1,5× más lento**: arrastra una sola cadena de acumulación a través de la reducción, que es justo la dependencia serie que el segundo acumulador existe para romper. Misma biblioteca, respuestas opuestas, decididas midiendo y no por reputación — ver `bench/README.es.md`. **«Usa la función optimizada de la biblioteca» es una hipótesis, no una conclusión.**

`SimdOps.AddScaled` es el caballo de batalla del `backward pass`. Fíjate en que, según el §8, los pasos 3 y 4 son ambos «suma un vector escalado en un acumulador» — la misma primitiva sirve para acumular el gradiente de los pesos, propagar el gradiente de entrada y para el propio paso de descenso.

Por qué estas viven fuera de `Dense<T>`

El JIT emite una copia de código distinta por cada instanciación genérica con tipo por valor (§11). Si se dejara dentro de `Dense<TActivation>`, `Dot` se duplicaría para `Dense<Tanh>`, `Dense<ReLU>`, `Dense<Sigmoid>`… inflando la cache de instrucciones sin ningún beneficio. Una clase auxiliar no genérica obtiene exactamente una copia.

---

16. El backward pass en código

`Dense.Backward` — el patrón de cuatro pasos del §8, transcrito:

```
for (int j = 0; j < Units; j++)
{
    float delta = gradOut[j] * TActivation.DerivativeFromOutput(a[j]);   // paso 2
    if (delta == 0f) continue;                                          // atajo para ReLU
        muerta

    int offset = j * Inputs;
    SimdOps.AddScaled(_weightGrads.AsSpan(offset, Inputs), x, delta);    // paso 3: dL/dW +=
        δ·x
    _biasGrads[j] += delta;                                             // paso 3: dL/db += δ

    if (propagate)
        SimdOps.AddScaled(gradIn, Weights.AsSpan(offset, Inputs), delta); // paso 4: dL/dx +=
        δ·W
}
```

Tres cosas dignas de notar:

Los pasos 3 y 4 son la misma operación con los pesos y las entradas intercambiados. `dL/dW += δ·x` y `dL/dx += δ·W` son imágenes especulares — esa simetría es la razón por la que una sola primitiva `AddScaled` cubre ambos. Y gracias a la disposición del §12, *ambos* recorren la memoria de forma contigua. La decisión que aceleró el forward pass rinde aquí dos veces más.

`gradIn` está vacío para la primera capa. Nada consume el gradiente respecto a los datos de entrada crudos, así que calcularlo sería trabajo desperdiciado. (En algunas técnicas — ejemplos adversarios, transferencia de estilo — ese gradiente de entrada es exactamente lo que quieres. Aquí no.)

`if (delta == 0f) continue;` se salta las unidades que no aportan nada. Con ReLU esto es frecuente: cualquier unidad cuya salida se recortó a 0 tiene derivada nula, así que no acumula nada. Lo cual también señala un modo de fallo real — véase §23.

Acumular ahora, aplicar después

`Backward` solo **suma en** `_weightGrads`. Nunca toca `Weights`. `ApplyGradients` realiza el paso de descenso real y limpia los acumuladores:

$$W -= \eta \cdot \frac{1}{\text{batchSize}} \frac{\partial L}{\partial W}$$

¿Por qué separarlos? Porque permite sumar gradientes de varios ejemplos antes de actualizar — los mini-batches, §20. Dividir por el batch size promedia en lugar de sumar, de modo que tu `learning rate` sigue funcionando cuando cam-

bias el batch size.

Esta separación no es una peculiaridad de este código. PyTorch divide exactamente por la misma costura: `loss.backward()` acumula, `optimizer.step()` aplica. Si pasas a un framework real, esto te resultará familiar.

---

17. Inicialización de los pesos — realmente no es opcional

`Dense.Initialize` usa **Xavier/Glorot uniforme**:

$$W \sim \mathcal{U}(-\sqrt{\frac{6}{n_{in} + n_{out}}},\ +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

Por qué fallan los pesos idénticos — el problema de la simetría

Empieza todos los pesos de una capa con el mismo valor y cada unidad calcula la salida idéntica. Salidas idénticas obtienen gradientes idénticos. Gradientes idénticos producen actualizaciones idénticas. **Las unidades permanecen idénticas para siempre.**

Una capa de 100 unidades inicializada así tiene el poder expresivo de *una* unidad, de forma permanente — ninguna cantidad de entrenamiento escapa de ello, porque nada rompe el empate. La inicialización aleatoria **rompe la simetría**: cada unidad empieza distinta, recibe un gradiente distinto y se especializa.

Por qué los pesos *todos a cero* fallan aún más

El cero es un caso especial, y vale la pena separarlo porque es el que puedes ejecutar (ejercicio 2). No se limita a colapsar la capa a una unidad efectiva — detiene el aprendizaje **por completo**:

- Cada unidad oculta produce `g(0)`, y el gradiente devuelto a la capa oculta es `δ·W = δ·0 = 0`.
- El gradiente de los pesos ocultos es `δ·x` con `δ = 0`, así que los pesos ocultos nunca se mueven.
- Con `tanh`, las salidas ocultas son `tanh(0) = 0`, así que los gradientes de peso de la capa de salida (`δ·h`) también son cero.

Nada salvo el sesgo de salida puede moverse, y en XOR ni siquiera eso — los gradientes de los cuatro ejemplos se cancelan exactamente. Ejecuta el ejercicio 2 y verás la pérdida congelada en **exactamente 0,250000** con cada predicción

en 0,5000: la red adivinando permanentemente la media, sin haber aprendido literalmente nada.

Los sesgos pueden empezar en cero sin problema — los pesos aleatorios ya rompieron la simetría.

Por qué esa escala en concreto

Demasiado grande y las activaciones se saturan en ± 1, donde los gradientes se desvanecen; demasiado pequeña y la señal decae hacia cero al atravesar las capas. El `6/(fan_in + fan_out)` de Xavier mantiene la varianza de las activaciones aproximadamente constante entre capas, de modo que las señales ni explotan ni se desvanecen con la profundidad. Para ReLU, la **inicialización de He** (varianza `2/fan_in`) es la opción mejor ajustada, ya que ReLU descarta la mitad de su rango de entrada.

---

18. El perceptrón en código

`Perceptron` es la excepción del conjunto: es lo único aquí que no usa backpropagation. Existe para hacer concreta y ejecutable la historia del §9.

Es un `Dense<Step>` de exactamente una unidad, envuelto en su propia regla de entrenamiento:

```
public float Predict(ReadOnlySpan<float> x)
{
    _layer.Forward(x, _out);
    return _out[0];
}
```

El bucle de entrenamiento (`Perceptron.Train`) es la regla de 1958 al pie de la letra:

```
float error = y[i] - Predict(xi);
if (error == 0f) continue;          // predicción correcta: no hay actualización

float delta = learningRate * error;
for (int k = 0; k < Inputs; k++)
    w[k] += delta * xi[k];
Bias += delta;
```

Tres cosas que notar, todas las cuales contrastan de forma instructiva con backpropagation:

No aparece ninguna derivada por ninguna parte. Compáralo con `Dense.Backward`, donde cada gradiente se multiplica por `g'(z)`. La regla del perceptrón no necesita ninguna — lo cual viene muy bien, porque `Step.DerivativeFromOutput` lanza una excepción (§13). Este es el mismo hecho visto desde dos direcciones: una función escalón no tiene pendiente aprovechable, así que el descenso de gradiente no puede funcionar sobre ella, y el perceptrón lo esquiva no siendo descenso de gradiente.

`error` solo vale −1, 0 o +1, ya que tanto la predicción como el objetivo son 0 o 1. Es una dirección — «demasiado alto», «correcto», «demasiado bajo» — no una magnitud. La δ de la backpropagation lleva también magnitud, que es lo que le permite decir *cuánto* corregir.

Las predicciones correctas no provocan actualización, y una época sin actualizaciones termina el entrenamiento anticipadamente. Esa salida temprana es lo que hace de «convergió en 4 épocas» una afirmación con sentido, en lugar de simplemente haberse alcanzado el límite de épocas — y es por lo que el caso XOR agota en cambio todo el presupuesto.

Las pruebas fijan ambos comportamientos: `Perceptron_converges_on_linearly_separable_data` y `Perceptron_cannot_learn_xor`. La segunda afirma un *fracaso*, lo cual es inusual y deliberado — fija una propiedad del algoritmo que el resto de la biblioteca existe para superar.

---

19. La red y el constructor Sequential

`Network` contiene un `ILayer[]`. Este es un modelo **secuencial** exactamente en el sentido de Keras: una cadena estrictamente lineal, donde la salida de cada capa alimenta a la siguiente, sin ramificaciones.

Cómo construir una

`Sequential` es un constructor fluido que refleja la API de Keras que encontrarás en cursos y tutoriales:

```
var net = new Sequential(inputs: 2)
    .Dense<Tanh>(4)
    .Dense<Sigmoid>(1)
    .Build(seed: 42);
```

```
# equivalente en Keras
model = Sequential([
    Dense(4, activation='tanh'),
    Dense(1, activation='sigmoid'),
])
```

El verdadero trabajo del constructor es la **inferencia del tamaño de entrada**. Declaras el ancho de entrada una vez; cada capa toma su número de entradas del número de unidades de la capa anterior. El constructor directo sigue funcionando y hace lo mismo, pero te obliga a declarar — y mantener coherentes — ambos extremos de cada capa:

```
var net = new Network(seed: 42,
    new Dense<Tanh>(inputs: 2, units: 4),
    new Dense<Sigmoid>(inputs: 4, units: 1));   // el 4 debe coincidir a mano
```

Dos errores de forma son imposibles de expresar mediante el constructor fluido y simplemente los detecta el constructor de la clase: `.Add(layer)` rechaza una capa cuyo `Inputs` no coincide con el ancho actual, y `Build()` rechaza una pila vacía.

`Summary()`

Como el `model.summary()` de Keras:

```
Layer                    Output    Params
─────────────────────────────────────────
Dense<Tanh>                   4        12
Dense<Sigmoid>                1         5
─────────────────────────────────────────
Input width: 2
Trainable parameters: 17
```

Comprueba tú mismo la aritmética — es la fórmula del §3. La capa tanh tiene 2 entradas × 4 unidades = 8 pesos, más 4 sesgos = 12. La capa de salida tiene 4 × 1 = 4 pesos más 1 sesgo = 5.

Guardar y cargar un modelo entrenado

El entrenamiento produce una única cosa de valor: los pesos. `ModelIO` los escribe, junto con suficiente arquitectura para reconstruir la red:

```
ModelIO.Save(net, "xor.nnm");            // after training
var loaded = ModelIO.Load("xor.nnm");    // later, or in another program
float p = loaded.Predict(input)[0];      // listo para usar; no hace falta entrenar
```

El modelo XOR ocupa **127 bytes**, y la recarga es idéntica bit a bit: mismas entradas, mismas salidas, exactamente.

El formato del archivo, little-endian:

```
magic     8 bytes   "NNMODEL\0"
version   int32     2
loss      string    "mse" or "softmax-cross-entropy"   (version 2+)
  layers    int32     cantidad de capas
per layer:
  descriptor  string   "Dense<Tanh>"
  inputs      int32
  units       int32
  weights     float32 × inputs × units
  biases      float32 × units
```

Merece la pena entender cinco decisiones que hay ahí dentro, porque son las que la gente hace mal:

Guarda la arquitectura, no solo los pesos. Un volcado pelado de floats no tiene ni idea de qué forma tenía, así que cargarlo en código que no coincide produce basura en silencio. Almacenar los tipos y formas de las capas hace que una discrepancia se detecte de inmediato.

Bytes mágicos y un número de versión. Sin ellos, cualquier archivo equivocado se interpreta como floats y «funciona» hasta que las predicciones resultan ser un disparate. Con ellos, obtienes *«Not a model file»* o *«Model format version 99 is not supported»*. Los archivos truncados también se detectan, nombrando la capa a la que se le acabaron los datos. **Fallar ruidosamente ante una entrada mala** es la lección general; los archivos de modelo son simplemente un sitio donde es fácil saltársela.

Una tabla descriptor→constructor (`ModelIO.Factories`) convierte la cadena `"Dense<Tanh>"` de vuelta en un tipo real. C# no puede construir un tipo genérico a partir de una cadena sin esta tabla o sin reflexión, y una tabla explícita es a la vez más rápida y más segura — la reflexión permitiría que un archivo malicioso nombrara *cualquier* tipo de tu proceso. `ModelIO.Register` la extiende para capas personalizadas.

La carga no debe inicializar. El constructor público de `Network` aleatoriza los pesos, así que construir una red a partir de capas cargadas destruiría los parámetros que acabas de leer. Por eso `ModelIO` llama a `Network.FromTrainedLayers`, que se salta la inicialización. Es un error real

con el que me topé al escribir esto, y es desagradable precisamente porque no revienta — simplemente obtendrías una red sin entrenar que se carga «con éxito».

Guarda la pérdida, no solo las capas — el cambio que llevó el formato a la versión 2, y el caso para el que se puso ahí el campo de versión. Los pesos de un clasificador softmax carecen de sentido sin saber que hay que aplicarles softmax: cárgalo como una red normal y devuelve logits sin cotas donde quien llama espera probabilidades. No se lanza nada, `Predict` sigue devolviendo diez números, y solo sus *valores* están mal (§27). Los archivos de versión 1 son anteriores a que hubiera elección de pérdida y siguen cargándose — como MSE, que es lo que eran. **Una versión de formato se gana su sitio la primera vez que necesitas añadir un campo**, y «los archivos antiguos siguen funcionando» es todo el rendimiento de haberla escrito.

> **Lo que *no* se guarda:** los acumuladores de gradiente y la cache de activaciones del `forward pass`. Eso es espacio de trabajo del entrenamiento, reconstruido vacío al cargar. Si más adelante añades momentum o Adam, su estado también habría que guardarlo para reanudar el entrenamiento a mitad — véase §25 punto 7.

Dónde deja de bastar lo secuencial

Un arreglo recorrido de principio a fin solo puede expresar una cadena. Las conexiones de salto (ResNet), múltiples entradas o salidas, y la concatenación necesitan todas un **grafo** de capas con una ordenación topológica en lugar de estos dos bucles — para eso está la API funcional de Keras. Todo lo de un curso introductorio es secuencial, razón por la cual Keras lo hace el modo predeterminado.

La pila de capas en sí

Por qué existe una interfaz no genérica junto a los genéricos. `Dense<Tanh>` y `Dense<Sigmoid>` son *tipos sin relación entre sí* — no puedes ponerlos en el mismo arreglo. Los genéricos dan velocidad; la interfaz `ILayer` da heterogeneidad. Necesitas ambas, así que el código tiene ambas.

Forward pass: `x → layer0 → layer1 → … → output`

Backward pass: recorre el arreglo en sentido inverso, y el `gradIn` de cada capa se convierte en el `gradOut` de la siguiente:

```
for (int i = last; i >= 0; i--)
    _layers[i].Backward(_grads[i], i > 0 ? _grads[i - 1] : Span<float>.Empty);
```

Esa única línea es el paso «entrega el gradiente a la capa anterior» del §8, generalizado a cualquier profundidad.

El constructor valida que las formas de capas adyacentes coincidan — el `Inputs` de la capa `i` debe igualar el `Units` de la capa `i-1` — y falla de inmediato con un mensaje claro en lugar de producir un disparate más tarde. También preasigna todos los buffers, de modo que el entrenamiento no asigna nada por ejemplo.

Hilos, y quién es dueño de los buffers

Esos buffers preasignados son lo que hace que el entrenamiento no asigne memoria, y son también la razón de dos reglas que la API no puede imponerte.

Una red por hilo. `Network` y `Dense` mantienen buffers de activación mutables, acumuladores de gradiente, la cache del forward pass y el orden del barajado. Nada de eso está sincronizado, y nada es seguro de compartir. Dos hilos llamando a `Predict` sobre una red entrelazarán escrituras en el mismo arreglo de activaciones y ambos obtendrán basura — sin ninguna excepción, porque técnicamente nada está mal a nivel de tipos.

La única excepción es `Dense.Forward` sobre una capa independiente, que tras la separación del §14 no escribe ningún estado de instancia. `Network.Predict` sigue escribiendo buffers de activación compartidos, así que una red por hilo sigue siendo la regla. (El §25 punto 4 señala que la biblioteca tampoco *usa* hilos nunca — no hay ningún `Parallel.For` en ninguna parte.)

`Predict` te presta un buffer; no te lo regala.

```
ReadOnlySpan<float> a = net.Predict(x1);
ReadOnlySpan<float> b = net.Predict(x2);   // a y b son la MISMA memoria; ambos contienen
        ahora el resultado de x2
```

El span devuelto es una vista de `_activations[^1]`, que la siguiente forward pass sobrescribe. Cópialo si necesitas conservarlo:

```
float[] kept = net.Predict(x1).ToArray();
```

Este es el intercambio estándar de las API sin asignaciones, y `Span<T>` es justamente el tipo que lo hace explícito — véase §11. `ModelIO.Register` tiene el mismo carácter a escala de proceso: el acceso a la tabla está sincronizado, pero registrar tipos de capa personalizados durante el arranque mantiene el comportamiento de carga independiente del momento en que ocurra.

---

20. El bucle de entrenamiento

Por **época** (`epoch`, una pasada completa sobre los datos):

1. **Baraja** el orden de los ejemplos.
2. Para cada **mini-batch**: acumula gradientes sobre sus ejemplos, y luego aplícalos una vez.

Por qué barajar

Con un orden fijo la red puede aprender el *orden* en lugar de los datos, y muestras consecutivas correlacionadas (todas de la clase A, luego todas de la clase B) producen estimaciones sesgadas del gradiente que hacen que el entrenamiento dé bandazos. Barajar hace que cada `batch` sea una muestra más justa del dataset completo.

Batch size — los tres regímenes

Batch size	Nombre	Comportamiento
1	Estocástico (SGD)	Actualizaciones rápidas y ruidosas. El ruido puede ayudar a escapar de mínimos locales poco profundos.
8–256	Mini-batch	El compromiso estándar, y mucho más eficiente para el hardware.
Todos	Full batch	El gradiente más suave, el avance más lento, más memoria.

Aquí el valor predeterminado es `full batch`, ya que XOR tiene cuatro ejemplos.

Promediar sobre un `batch` reduce el ruido del gradiente — los ejemplos individuales discrepan sobre cuál es la mejor dirección, y promediar encuentra su consenso.

La sección de las dos lunas de la demo es donde esto se hace visible (§22): 1000 ejemplos con batch size 32 dan ~31 actualizaciones por época, así que 150 épocas son 4650 actualizaciones. Las 4000 épocas de XOR son 4000 actualizaciones. **Las épocas no son la unidad que importa** — lo son las actualizaciones, y la proporción entre ambas es el batch size.

Ten en cuenta que aquí los mini-batches son actualmente un recurso *estadístico*, no de rendimiento. Los frameworks reales agrupan en batches porque eso permite que un bloque de pesos cargado sirva a muchos ejemplos a la vez; esta biblioteca sigue recorriendo los pesos una vez por ejemplo en cualquier caso, razón por la cual `ForwardBatch` no mide más rápido que un bucle (§25 punto 1).

Épocas

Una época = una pasada sobre todos los ejemplos. Las redes necesitan muchas porque cada actualización es un paso pequeño (§4).

XOR requiere aquí ~4000 épocas, lo cual suena enorme. Pero la demo se ejecuta a **full batch**, así que una época = una actualización de parámetros: 4000 pasos reales de descenso de gradiente, cada uno informado por los cuatro ejemplos (16 000 evaluaciones de ejemplo). Cuatro mil pasos pequeños para ajustar 17 parámetros no es nada notable — y es por lo que importa la distinción entre una *época* y una *actualización*. Con batch size 1 esas mismas 4000 épocas significarían 16 000 actualizaciones.

---

21. Gradient checking — cómo *sabes* que el código es correcto

Esta es la sección que la mayoría de los tutoriales omite, y es la más valiosa en la práctica.

El problema: un backward pass sutilmente incorrecto normalmente sigue entrenando *hasta cierto punto*. La pérdida baja, nada revienta, y los resultados son simplemente mediocres — indistinguibles de «hay que ajustar los hiperparámetros». Estos errores pueden costar días.

La defensa: compararla contra una estimación numérica que no use tu código de backpropagation (`GradientCheck.cs`). Mueve un peso arriba y abajo, mide cómo se mueve realmente la pérdida, y compara:

$$\frac{\partial L}{\partial w} \approx \frac{L(w+\epsilon) - L(w-\epsilon)}{2\epsilon}$$

Esta es la definición de derivada con un ε finito en lugar de un límite. Es demasiado lenta para entrenar — dos forward passes completos *por parámetro* — pero perfecta para verificar. La diferencia **central** (en ambas direcciones) tiene error O(ε²) frente a O(ε) de la versión de un solo lado, lo que compensa de sobra la segunda evaluación.

El resultado medido, y qué verifica

ε	error relativo máximo	dominado por
1e-1	9,1e-3	truncamiento — ε demasiado grueso para ser una buena derivada
1e-2	**2,4e-4**	equilibrado ← el mejor
1e-3	1,7e-3	el redondeo empieza a colarse
1e-4	1,5e-2	redondeo de float32 — `L(w+ε)` y `L(w−ε)` casi idénticos

La forma de U es la evidencia que buscas. Dos fuentes de error luchan entre sí: un ε grande es una mala aproximación de una derivada, mientras que un ε pequeño resta dos floats casi iguales y pierde precisión catastróficamente. Un gradiente implementado correctamente muestra este compromiso con un punto óptimo en el medio. **Un gradiente incorrecto muestra error O(1) con cualquier ε** — sin punto óptimo, porque no está aproximando nada.

`float` toca fondo alrededor de 1e-4; las comprobaciones de producción usan `double` y esperan ~1e-10.

Regla: siempre que añadas un tipo de capa o una activación, verifica su gradiente antes de confiar en ella.

Cómo usarla

```
float error = GradientCheck.MaxRelativeError(net, oneInput, itsTarget);
```

Funciona con cualquier red porque las capas exponen sus parámetros mediante un índice plano y agnóstico al tipo (`GetParameter` / `SetParameter` / `GetParameterGradient` en `ILayer`) en lugar de que la verificación conozca específicamente a `Dense`.

En la batería de pruebas

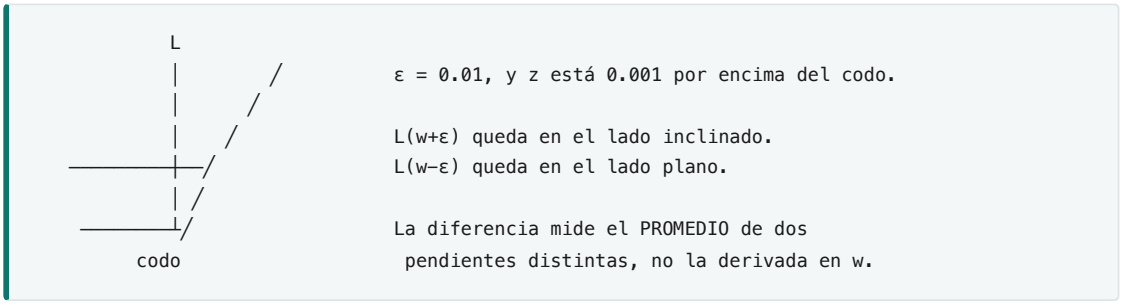
Las comprobaciones se ejecutan como pruebas reales (`dotnet test`), incluidas dos que es fácil pasar por alto:

- **Una derivada incorrecta a propósito** — una variante de `Tanh` que usa `1 + a²` en lugar de `1 − a²` — debe producir error O(1). Sin esto, las pruebas que pasan podrían estar pasando *vacuamente*: una comprobación que no puede fallar no demuestra nada.
- **La profundidad eleva el suelo del error.** Tres capas miden ~4,6e-3 frente a ~2,4e-4 de dos, porque cada capa extra acumula redondeo de float32 en las evaluaciones de pérdida de las que depende la diferencia finita. El umbral se relaja para la prueba más profunda — pero sigue separando un gradiente correcto de uno incorrecto por más de un orden de magnitud.

La excepción de ReLU — cuando una comprobación que falla *no* es un error

La regla de arriba dice «verifica el gradiente antes de confiar en ella». Hay un caso en el que la comprobación falla sobre código correcto, y conviene conocerlo antes de que te ocurra, porque la conclusión natural es que tu backward pass está roto.

Las diferencias finitas suponen que la pérdida es suave entre `w−ε` y `w+ε`. ReLU no lo es. Tiene un codo en `z = 0`. Si mover un peso en ε empuja la `z` de alguna unidad a cruzar el cero, las dos evaluaciones de pérdida quedan a lados opuestos de la esquina:



El gradiente analítico es **correcto**. La *estimación numérica* es la que está mal. Medido sobre exactamente ese montaje — una unidad ReLU con `z` a 0,001 del codo:

ε	error relativo máximo	qué significa
1e-2 (predeterminado)	**2,9e-1**	cruza la esquina — parece un fallo total
1e-4	8,8e-2	ε ya menor que la distancia al codo; se recupera

Compáralo con una derivada genuinamente rota, que se mantiene por encima de 1e-1 con *cualquier* ε y para *cualquier* activación. Eso te da dos diagnósticos para distinguirlas:

1. **Reduce ε.** Un artefacto del codo mejora bruscamente; un error real no se mueve.
2. **Cambia la activación.** Reconstruye la misma forma de red con tanh. La aritmética de la capa bajo prueba es idéntica, y tanh no tiene esquina que cruzar — medido 4,1e-4 en la forma de arriba. Si tanh pasa y ReLU no, has encontrado un codo, no un error.

Ambos comportamientos están fijados por pruebas (`ReLUKinkTests`), así que esto sigue siendo cierto. Es una limitación conocida de la técnica y no de este código, razón por la cual las comprobaciones de producción prefieren activaciones suaves incluso cuando la red desplegada usa ReLU.

Parte III — Práctica

22. Resultados

> **Por qué tus dígitos pueden diferir.** Todos los números de esta guía se produjeron en un **Apple M3 Pro, .NET 10, ARM64** (`Vector<float>.Count == 4`). La suma en coma flotante no es asociativa — `(a+b)+c` y `a+(b+c)` pueden diferir en los últimos bits — y el producto escalar SIMD suma en un orden que depende del ancho del vector, que es 8 en AVX2 y 16 en AVX-512. Así que el mismo código, la misma semilla y los mismos datos pueden producir dígitos finales ligeramente distintos en x86, y esas diferencias se acumulan a lo largo de 4000 épocas.
>
> Espera que los últimos dígitos cambien. Espera que las conclusiones no: ningún resultado aquí depende de un dígito que no sea estable. La integración continua se ejecuta tanto en ARM como en x86 exactamente por esta razón. Vale la pena interiorizarlo en general — **la reproducibilidad bit a bit entre máquinas no es algo que el código en coma flotante te dé gratis**, y perseguirla es un desperdicio de tiempo habitual.

A escala XOR

```
Perceptron on AND: converged in 4 epochs      ← linealmente separable

Network on XOR (2 -> 4 tanh -> 1 sigmoid):
  epoch  1000  loss 0.002304
  epoch  4000  loss 0.000350
  0 XOR 0 -> 0.0100    1 XOR 0 -> 0.9779
  0 XOR 1 -> 0.9826    1 XOR 1 -> 0.0226      ← la capa oculta haciendo su trabajo

Gradient check: max relative error = 3.521E-004
```

Las salidas de XOR no son exactamente 0 y 1 porque la sigmoide solo *se aproxima* a sus límites — alcanzar 1,0 exactamente requeriría pesos infinitos. 0,98 significa «un 1 con confianza».

Más allá de XOR — las dos lunas

XOR demuestra que la capa oculta derrota a Minsky y Papert, y eso es *todo* lo que puede demostrar. Con cuatro ejemplos sin ruido y sin datos reservados no hay manera de demostrar las tres cosas que dominan el entrenamiento en la práctica:

	por qué XOR no puede mostrarlo
Mini-batches	Cuatro ejemplos son un `full batch`. Cada «época» es una sola actualización.
Generalización	Los cuatro puntos *son* el problema. No hay nada reservado a lo que generalizar.
Sobreajuste	Nada que memorizar.

Así que las dos últimas secciones de la demo usan `Datasets.Moons`: dos medias lunas entrelazadas con ruido gaussiano, 1500 puntos, divididos en 1000 de entrenamiento y 500 de prueba. Generados en lugar de descargados — sin archivos de datos, sin acceso a la red, idénticos en cualquier máquina para una semilla dada.

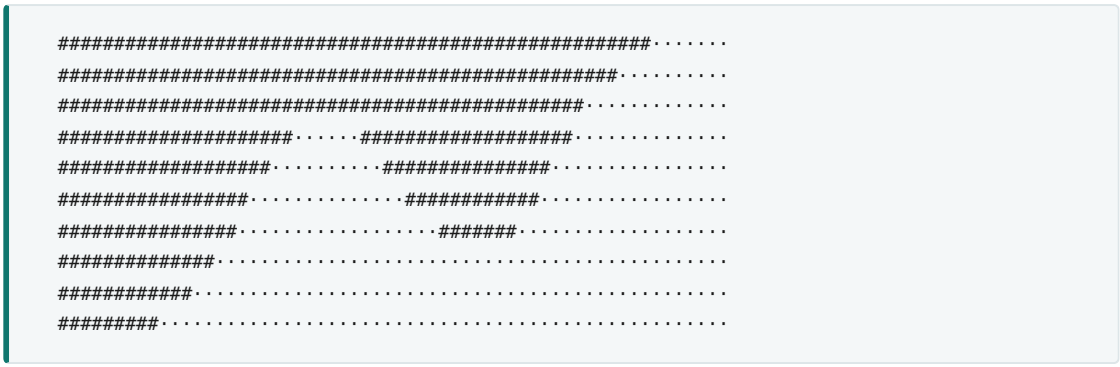
Una red 2 → 16 → 16 → 1, batch size 32, learning rate 0,3:

```
epoch   train loss   train acc   test acc
    1       0.1511      83.9%     85.2%
   30       0.0383      95.9%     96.4%
   90       0.0190      97.6%     96.0%
  150       0.0177      98.1%     96.4%
```

Que el `test accuracy` siga al `train accuracy` es el aspecto que tiene la generalización. Fíjate también en que ninguna llega al 100 % y en que eso es correcto: con este nivel de ruido las medias lunas se solapan de verdad, así que algunos puntos son inclasificables y un modelo que puntuara 100 % sería un modelo que los memorizó.

Fíjate en el número de épocas frente a las 4000 de XOR. Aquí hay ~31 actualizaciones por época en lugar de una, que es la distinción del §20 hecha concreta: **4650 actualizaciones, no 150.**

La frontera aprendida, muestreada por todo el plano:



Esa curva es todo el argumento a favor de las capas ocultas, dibujado a escala. Un perceptrón solo puede poner una línea recta sobre esta imagen — y la batería de pruebas afirma que lo hace apreciablemente peor.

Sobreajuste, a propósito

Mismo problema, misma demo: 4417 parámetros entrenados con **20** puntos — 221 veces más parámetros que datos.

```
epoch    train acc   test acc
    1       85.0%     82.0%
 1000       90.0%     86.4%
 3000      100.0%     93.4%
```

Perfecto con los datos que ha visto; **93,4 % con los que no, frente al 96,4 % de la red más pequeña con más datos.** Esa brecha es el sobreajuste: capacidad gastada en memorizar el ruido de 20 puntos en lugar de aprender la forma.

La parte importante es lo que hizo falta para verlo. La pérdida de entrenamiento cayó todo el tiempo; no se lanzó nada; el modelo parecía *mejor* según todos los números disponibles durante el entrenamiento. **Solo la partición reservada lo hizo visible**, y nada en esta biblioteca la calcula por ti — §25 punto 6.

Dígitos manuscritos — un problema real

`dotnet run -c Release --project src/NN.Mnist` entrena con MNIST: 60 000 dígitos manuscritos, escala de grises 28×28, el dataset al que toda introducción acaba llegando. El salto de escala respecto a las dos lunas es lo importante — 784 entradas en lugar de 2, y 101 770 parámetros en lugar de 337.

Arquitectura: **784 → 128 tanh → 10 logits → softmax**, una salida por dígito, entrenada con objetivos one-hot (un vector con un único 1 en la clase correcta) y cross-entropy. La predicción es la probabilidad más alta.

```
epoch   train loss   test acc      elapsed
    1      0.32598    93.53%        2.1s
   10      0.04203    97.86%       18.6s
   20      0.01349    98.02%       36.9s

Final: train 99.90%, test 98.02%, 37.3s total
```

98,02 % sobre dígitos que nunca ha visto, en 37 segundos, con el código de esta guía. Dos segundos por época sobre 60 000 ejemplos — el trabajo de SIMD y disposición de los §12 y §15 opera por fin a un tamaño en el que importa. Por eso los benchmarks también miden una capa de 784×128.

Tres cosas que enseña esta demo y que nada más pequeño puede:

El `accuracy` oculta la estructura; una matriz de confusión la muestra. Los errores no están repartidos de manera uniforme:

```
          0     1     2     3     4     5     6     7     8     9    accuracy
1         ·  1128     3     1     ·     1     1     1     ·     ·      99.4%
5         3     1     ·    12     1   863     5     ·     5     2      96.7%
9         2     2     1     7     4     3     2     4     2   982      97.3%
```

Los unos son casi perfectos; los cincos son la peor clase, y doce de ellos se clasifican como 3 — dos dígitos que comparten realmente un trazo superior y una panza inferior. Los errores de la red están *estructurados*, y la demo imprime los dígitos mal clasificados en ASCII para que veas que la mayoría son de los que una persona también dudaría.

La elección de la pérdida vale más que cualquier cantidad de entrenamiento adicional. La misma red con MSE sobre diez sigmoides solo alcanza el 97,41 %, y necesita un `learning rate` de **1,0** para lograrlo — muy fuera del 0,1–0,5 que recomienda el §4. Ejecuta `--loss mse` y obsérvalo. La causa es un gradiente que llega demasiado reducido por dos factores:

$$\frac{\partial L}{\partial a} = \frac{2(a-y)}{10} \quad \text{luego} \quad \times\, \sigma'(z) = a(1-a) \le 0.25$$

Dividir entre diez salidas, y luego multiplicar por un factor que como mucho vale 0,25 y se desploma hacia cero conforme las salidas se saturan — es decir, exactamente cuando la red está equivocada con confianza y más necesita aprender. El enorme tamaño de paso es la compensación por una desventaja que impuso la función de pérdida.

Softmax con cross-entropy la elimina. Medido sobre arquitectura, semilla y épocas idénticas:

Pérdida	Test accuracy	Learning rate
MSE sobre diez sigmoides	97,41 %	1,0
MSE, con el learning rate de la cross-entropy	92,93 %	0,1
Softmax + cross-entropy	**98,02 %**	**0,1**

La fila del medio es la honesta: con el mismo learning rate la diferencia es de cinco puntos, y MSE solo cierra la brecha dando pasos diez veces mayores. El §27 explica el mecanismo.

Los límites documentados de la biblioteca se vuelven medibles en lugar de teóricos. El 98,0 % es lo normal para esta arquitectura. Lo que queda es el §25 punto 2 — SGD simple, sin momentum ni Adam — que es el ejercicio 10, con un número concreto que batir.

Y aquí es donde guardar un modelo por fin significa algo

La demo escribe la red entrenada en disco y la reutiliza. La salida está en inglés porque es la salida real del programa:

```
Saved to ~/.../nn-mnist/mnist-128.nnm
  397 KB for 101,770 parameters (4.0 bytes each — float32 plus a header)
  Reloaded and verified: all 1,000 sampled predictions are bit-for-bit identical.

(next run)
Loaded a trained model — no training needed.
  397 KB, 101,770 parameters, loaded in 4 ms
```

37 segundos se convierten en 4 milisegundos, con `accuracy` idéntico. El §19 explicó el formato de archivo con un modelo de 17 parámetros, donde la persistencia es una curiosidad. Aquí es la diferencia entre una demo que ejecutas una vez y una demo que puedes usar de verdad — y es la forma normal del machine learning desplegado, donde el entrenamiento ocurre rara vez en hardware caro y la inferencia ocurre constantemente en otro sitio. Los 397 KB son todo el entregable; las 60 000 imágenes de entrenamiento no hacen falta para clasificar nada.

Vale la pena destacar tres detalles.

La arquitectura se recupera desde el archivo. A nada en la ruta de carga se le dice que está leyendo una red 784-128-10 — `ModelIO` guardó el tipo y la forma de cada capa junto a sus pesos, de modo que `Load` reconstruye la pila y `Summary()` la imprime. Por eso el §19 argumentaba en contra de un volcado pelado de pesos.

La recarga se verifica, no se supone. Tras guardar, la demo recarga y compara 1000 predicciones bit a bit. Esto importa más con 101 770 parámetros que con 17: con diecisiete, un peso mal serializado casi con seguridad rompe una predicción de forma visible, mientras que con cien mil un único valor fuera de sitio desplaza el `accuracy` una fracción de punto porcentual y se lee como ruido. La igualdad exacta es la única comparación que lo detecta, y `ModelScaleTests` fija esa misma propiedad.

El nombre del archivo lleva lo que el formato no. El archivo del modelo registra la arquitectura pero no tiene ni idea de *cuántos datos vio*. Un modelo entrenado con 5000 imágenes se recarga tan tranquilamente junto a uno entrenado con 60 000, así que la demo pone tanto el ancho de la capa oculta como cualquier límite del training set en el nombre del archivo. Ninguna de las dos discrepancias sería un error — ambas serían resultados silenciosamente incorrectos, que es la clase más difícil de notar.

> Esto es también el §25 punto 7 en la práctica: solo se guardan los parámetros, no el estado del optimizador ni el historial de entrenamiento. Aquí está bien, porque lo único que se quiere es inferencia. No podrías reanudar desde este archivo un entrenamiento interrumpido.

Leer un dígito desde un archivo de imagen

Un modelo entrenado en disco significa que puedes apuntarlo a una imagen. El repositorio incluye uno — un reconocedor entrenado, `models/mnist-784-128-10.nnm` — de modo que esto funciona en un clon recién hecho, sin entrenar nada y sin descargar el dataset:

```
dotnet run -c Release --project src/NN.Mnist -- --image my-digit.png
```

```
Model:  .../mnist-784-128-10.nnm
Image:  my-digit.png

  248x248 image, normalized to MNIST's 28x28 convention:

                       ..******
                     ==@@@@@@@@@@++
                   ..@@@@%%--..@@@@..
                   ++@@%%      ::@@++
                   ++@@--        @@++

  This is a 0.  (confidence 0.999)

    0  0.999  ████████████████████████████████
    9  0.000
```

PNG y Netpbm se decodifican en `ImageFile.cs` sin más dependencias que el framework. La mayor parte de ese archivo no es descompresión — de eso se encarga `ZLibStream` — sino la **reversión de los filtros por fila** que usa PNG: antes de comprimir, cada fila almacena la *diferencia* entre cada byte y una predicción calculada a partir de sus vecinos (el de la izquierda, el de arriba, o la mezcla de ambos de Paeth). Eso convierte las imágenes suaves en rachas de bytes casi nulos que deflate comprime muy bien. Por tanto la decodificación es estrictamente secuencial: el filtro «Up» se refiere a la fila de arriba ya reconstruida.

Pero el decodificador es la parte fácil. La parte importante es `DigitPreprocessor`.

La red no aprendió «dígitos» — aprendió las convenciones de MNIST

Esta es la razón más común, con diferencia, por la que un reconocedor hecho desde cero puntúa 97 % en el test set y luego falla con la primera foto que le das, y merece la pena interiorizarla mucho más allá de este proyecto.

Las imágenes de MNIST no son meramente «fotos de dígitos». Son imágenes bajo tres reglas concretas:

Convención	Tu imagen probablemente	Consecuencia si se ignora
Tinta blanca sobre negro	Bolígrafo oscuro sobre papel blanco	La red ve un marco brillante con un agujero oscuro — nada parecido a un dígito
El dígito llena un recuadro de 20×20	Pequeño, con margen alrededor	Reducir todo el marco deja un borrón de unos pocos píxeles
Centrado por centro de masa en 28×28	Donde quiera que estuviese	Cada trazo cae donde la red aprendió a ver fondo

Viola cualquiera de ellas y el `accuracy` se desploma de una forma que parece exactamente un modelo roto. Así que el preprocesador:

1. **Detecta la polaridad desde el borde**, no desde toda la imagen — el borde del marco es fondo casi por definición, mientras que un dígito grueso puede arrastrar la media global hacia lo oscuro más de lo que creerías.
2. **Recorta al rectángulo delimitador de la tinta**, y luego escala para caber en un recuadro de 20×20 *preservando la relación de aspecto*. Estirarlo para llenar el cuadrado le daría a un `1` la tinta de un `8` en los sitios equivocados.
3. **Remuestrea con un filtro de caja**, promediando cada píxel de origen que cae dentro de cada píxel de destino. El vecino más próximo es la elección obvia y es incorrecta: reducir 248 píxeles a 20 muestreando uno de cada doce descarta la

mayor parte del trazo y produce un dígito punteado y roto. Promediar lo conserva como los bordes grises suaves que el propio MNIST tiene.

4. **Centra por centro de masa**, no por rectángulo delimitador. MNIST lo hacía así, y la diferencia es real — un 7 con un trazo descendente largo lleva su masa arriba y el centro de su rectángulo abajo.

> **La lección se generaliza.** Esas cien líneas valen tanto como los 101 770 parámetros entrenados, porque los parámetros carecen de sentido aplicados a una entrada con la forma equivocada. «La mayor parte del machine learning es preparación de datos» suele decirse de los datos de *entrenamiento*; es igual de cierto para los datos que le entregas a un modelo terminado. El modelo es una función, y una función aplicada fuera de su dominio devuelve disparates con confianza en lugar de un error.

Verificándolo de extremo a extremo

El pipeline se comprobó exportando dígitos de prueba de MNIST como **PNG de 248×248, oscuro sobre claro y con márgenes amplios** — violando deliberadamente las tres convenciones — y leyéndolos de vuelta:

Diez de diez coincidieron con lo que el modelo predice sobre los datos crudos de MNIST, incluido uno en el que se *equivoca*: un 5 que llama 6 — con 0,898 de confianza a través de la pipeline de imagen, y 0,898 sobre el mismo dígito leído directamente del dataset. Ese último es el resultado útil. El pipeline reproduce los errores del modelo con la misma fidelidad que sus aciertos, y hasta tres decimales, que es como sabes que el preprocesado es transparente y no está ayudando por accidente. Un paso de preprocesado que «arreglara» ese 5 sería prueba de un error, no de calidad.

Si una predicción vuelve con poca confianza o con un segundo candidato muy cercano, la demo lo dice y te señala el renderizado 28×28 que imprimió. **Mira esa imagen primero.** Un dígito que aparece invertido, diminuto o descentrado es un problema de preprocesado, y ninguna cantidad de entrenamiento adicional lo arreglará.

> El dataset no está en el repositorio. La demo lo descarga una vez (~11 MB) y lo guarda en cache fuera del árbol de trabajo; las ejecuciones posteriores, incluidas las que no tienen conexión, leen esa cache. Sin red ni cache, la demo lo dice y termina limpiamente — un repositorio didáctico no debería fallar porque un servidor espejo esté caído. El formato IDX que analiza merece un vistazo (`Idx.cs`): un número mágico, unas dimensiones y bytes crudos. El detalle peligroso es el orden de bytes **big-endian**; si lo lees como little-endian, todo se rompe en silencio.

---

23. Manual de depuración

Los modos de fallo con los que realmente te toparás, y qué significan:

Síntoma	Causa probable	Solución
La pérdida → `NaN` o ∞	Learning rate demasiado alto; los pesos explotan	Divide el learning rate entre 10
La pérdida se estanca en un valor intermedio	Inicialización a cero o constante — simetría sin romper (§17)	Inicialización aleatoria
La pérdida baja y se estanca alta	Capacidad insuficiente, o unidades saturadas	Más unidades ocultas; prueba tanh en vez de sigmoide
La pérdida apenas se mueve	Learning rate demasiado bajo, o gradientes que se desvanecen	Sube el learning rate; usa ReLU/tanh
Entrena pero predice mal	Error en el gradiente, o muy pocas épocas	**Verifica el gradiente primero** (§21)
Una red ReLU deja de mejorar	Unidades muertas — salida 0 para toda entrada, gradiente permanentemente 0	Baja el learning rate; prueba leaky ReLU
Perfecto en entrenamiento, malo en datos nuevos	Sobreajuste — memorización, no aprendizaje	Más datos, menos parámetros, regularización

Verifica siempre el gradiente antes de ajustar hiperparámetros. Ajustar sobre un gradiente con errores es una pérdida de tiempo sin límite.

---

24. Ejercicios

Ordenados aproximadamente por valor. Los de «rómpelo» son los que más enseñan.

1. **Observa fallar a XOR.** Apunta `Perceptron` a los datos de XOR en lugar de a AND. No convergerá — ese es el muro de Minsky–Papert del §9, y sentirlo supera a leerlo.
2. **Rompe la inicialización.** Añade `Array.Clear(Weights); return;` al principio de `Dense.Initialize` para que los pesos se queden a cero. XOR se congela con una pérdida de **exactamente 0,250000**, prediciendo 0,5000 para toda entrada, para siempre. Ese es el bloqueo de pesos a cero del §17 — la demostración más convincente que hay aquí de por qué importa la inicialización. Después prueba a inicializar todos los pesos al mismo valor *no nulo* (0,5, por ejemplo) y observa el fallo distinto y más leve: la capa aprende, pero como si tuviera una sola unidad.
3. **Rompe el gradiente.** Cambia `Tanh.DerivativeFromOutput` a `1 + a * a`. Observa cómo la verificación del gradiente salta a O(1) mientras el entrenamiento *sigue funcionando en parte*. Para esto existe el §21.
4. **Barre el learning rate.** Prueba 0,01, 0,1, 0,5, 2,0 y 10,0 en XOR. Verás el avance lento, el descenso sano y la divergencia a `NaN` — todo el espectro del §4.
5. **Encoge la capa oculta** a 1 unidad. XOR vuelve a ser irresoluble; 2 es el mínimo teórico. Averigua a partir de dónde empieza a funcionar de forma fiable.
6. **Prueba ReLU** en la capa oculta. Puede que necesite un learning rate menor. Imprime la salida de cada unidad oculta para las cuatro entradas para detectar unidades muertas.
7. **Reproduce y luego perturba la tabla de características ocultas del §3.** Convierte `net.Layers[0]` a `Dense<Tanh>`, llama a `Forward` con cada una de las cuatro entradas e imprime los resultados — en una máquina ARM deberías obtener los números del §3, y en x86 los últimos dígitos pueden diferir (§22). Ahora cambia la semilla de la red (`new Network(seed: 7, …)`) e imprime otra vez. Obtendrás una descomposición completamente distinta e igual de válida. Este es el ejercicio más esclarecedor de todos: muestra que no hay un único conjunto «correcto» de características aprendidas.
8. **Barre el batch size en las dos lunas.** Prueba 1, 8, 32, 256 y `full batch` con un número fijo de épocas, y representa el `test accuracy` frente a *actualizaciones* en lugar de épocas. Los tres regímenes del §20, sobre un dataset lo bastante grande como para distinguirlos.
9. **Averigua dónde empieza el sobreajuste.** La demo usa 20 puntos de entrenamiento. Barre 20, 50, 200 y 1000 con capacidad fija y observa cómo se cierra la brecha entre entrenamiento y prueba. Después mantén los datos en 20 y encoge la red en su lugar. Dos curas distintas para la misma enfermedad.
10. **Añade momentum:** mantén un buffer de velocidad por capa, `v = βv + grad` ($\beta \approx 0{,}9$), y avanza a lo largo de `v`. Mídelo en MNIST, donde la referencia es

98,02 % en 37 s (§22) — un número lo bastante concreto como para batirlo o no. Esta es la mayor mejora aún sin implementar.

11. **Demuestra que la pérdida importa, y luego explícalo.** Ejecuta MNIST de ambas formas — `--loss mse --retrain` frente al valor predeterminado — y confirma la tabla del §22 en tu propia máquina. Después responde a la pregunta que plantea: MSE alcanza el 97,41 % *solo* con un `learning rate` de 1,0, y se queda en 92,93 % con el 0,1 de la cross-entropy. ¿Cuál de los dos factores reductores del §27 explica más parte de esa brecha? (Prueba MSE sobre una salida `Dense<Identity>` para eliminar `σ'(z)` manteniendo MSE, y mira dónde queda.)
12. **Deriva tú mismo el gradiente fusionado.** El §27 afirma que el jacobiano de softmax y el `1/p` de la cross-entropy se cancelan y dejan `p − y`. Haz el álgebra para el caso de dos clases y míralo ocurrir; después rompe a propósito `SoftmaxCrossEntropy.Gradient` — usa `p − y` multiplicado por 2, por ejemplo — y ejecuta `SoftmaxGradientTests`. Un gradiente incorrecto pero plausible sigue entrenando; la comprobación sigue detectándolo. Este es el argumento del §21 aplicado a la única pieza de álgebra del código.
13. **Mira en qué se equivoca MNIST.** La demo imprime sus propios dígitos mal clasificados. ¿Son genuinamente ambiguos, o la red falla en algo que a una persona le resultaría fácil? Después lee la matriz de confusión: ¿qué pares confunde, y comparten trazos esos dígitos? Este es el hábito que separa el «98 %» de saber qué hace realmente un modelo.
14. **Rompe la cache del forward pass a propósito.** Haz que `Forward` vuelva a guardar en cache (§14) y llama a `net.Predict(...)` entre `AccumulateGradients` y `ApplyGradients`. No se lanza nada, la pérdida sigue bajando, y la red entrena con el ejemplo equivocado. Después ejecuta `CacheLifetimeTests` y míralas detectarlo. La mejor demostración disponible de por qué «sigue entrenando» no demuestra nada.
15. **Implementa `ForwardBatch` como un GEMM real por bloques** (§25 punto 1) y compáralo con el resultado nulo existente en `bench/`. Esta es la mayor ganancia medida que sigue sobre la mesa.
16. **Vuelve a ejecutar los benchmarks en tu propia máquina.** Si es x86, `Vector<float>` tiene ancho 8 o 16 en lugar de 4. ¿Qué conclusiones de `bench/README.es.md` cambian, y cuáles se mantienen? Las que se mantienen son en las que merece la pena confiar.

---

25. Lo que esta implementación *no* hace

Límites honestos, ordenados por cuánto cuestan:

1. **Sin agrupación GEMM.** `ForwardBatch` simplemente repite en bucle la ruta de un solo ejemplo, volviendo a recorrer toda la matriz de pesos por cada ejemplo. La inferencia de un solo ejemplo está **limitada por el ancho de banda de memoria** — más o menos una multiplicación-suma por float cargado. Agrupar en una *multiplicación matriz-matriz por bloques* reutiliza cada bloque de pesos cargado a lo largo de muchos ejemplos, elevando la intensidad aritmética aproximadamente por el batch size. Las bibliotecas BLAS afinadas suelen reportar mejoras de un orden de magnitud por esto.

   La mitad de esto ya está medida. `ForwardBatch` mide **0,98× frente a un bucle manual** con batch sizes de 1, 32 y 256 — un resultado nulo, que confirma que hoy no aporta nada (tabla). Ese 2 % es overhead de bucle que quien llama ya no paga, no aritmética. La *otra* mitad — que un GEMM real por bloques dominaría cualquier otra optimización de aquí — sigue sin medir, porque sigue sin escribirse. Ese es el ejercicio 15, y sigue siendo la mayor ganancia individual disponible.
2. **SGD simple.** Sin momentum, Adam, planificación del learning rate ni `weight decay`. Adam suele converger varias veces más rápido — ahora medible frente al 98,02 % en 37 s de MNIST (§22). Con la cross-entropy implementada, esta es la mayor brecha que queda.
3. **Solo dos pérdidas** — MSE y softmax con cross-entropy (§27). Suficiente para regresión y clasificación de etiqueta única; la clasificación multietiqueta quiere binary cross-entropy por salida, que aquí no existe.
4. **Single-threaded.** Ningún `Parallel.For` sobre unidades ni sobre filas del batch.
5. **Sin FMA explícito.** `acc += a * b` puede fusionarse o no en una sola instrucción; `Vector256.FusedMultiplyAdd` lo garantiza, al coste de escribir una ruta ARM aparte.
6. **Sin regularización ni parada temprana, y sin partición de validación automática.** `Train` sobreajustará tan contento e informará de una pérdida decreciente todo el camino — el §22 lo muestra haciendo exactamente eso. La demo separa entrenamiento y prueba *a mano*, que es la versión mínima viable; nada en la biblioteca calcula un `validation score`, lo vigila, ni se detiene cuando se da la vuelta.
7. **La serialización guarda solo parámetros** — no el estado del optimizador (todavía no hay ninguno) ni el historial de entrenamiento. Bien para inferencia; no puedes reanudar un entrenamiento a medias desde un archivo.

8. **Forward y backward passes de un solo ejemplo.** Ambos recorren un ejemplo cada vez, así que la API no puede expresar un backward pass por batches aunque el punto 1 estuviera implementado. `ForwardBatch` solo sirve para inferencia por esta razón.

Para producción, el C# más rápido es el C# que llama a otra cosa: `TensorPrimitives`, ONNX Runtime o una GPU. Nadie gana a un BLAS afinado con bucles escritos a mano. Este código existe para entender, y a esta escala nunca será tu cuello de botella.

---

26. Hacia dónde seguir

Aproximadamente en orden:

1. **Momentum, y luego Adam** — ahora la mayor mejora individual todavía disponible, dado que la cross-entropy ya está implementada (§27). Referencia a batir: 98,02 % en 37 s.
2. **Un dataset real — ya está aquí.** MNIST es el primero clásico, y `src/NN.Mnist` entrena con él una red 784 → 128 → 10 hasta el 98,0 % en 37 segundos. El siguiente escalón es Fashion-MNIST (misma forma y mismo cargador, problema más difícil) o CIFAR-10 (en color, y que pide convolución de verdad).
3. **Regularización** — dropout, weight decay — una vez que puedas sobreajustar algo.
4. **Capas convolucionales**, si te interesan las imágenes.
5. **Un framework real.** Habiendo construido esto, el `loss.backward()` / `optimizer.step()` de PyTorch se leerá como maquinaria familiar en lugar de como magia — que es exactamente la recompensa de escribirlo tú mismo una vez.

---

27. Softmax y cross-entropy

MSE es la pérdida adecuada para la *regresión* — predecir números. Para la *clasificación* — elegir una entre varias categorías mutuamente excluyentes — es la herramienta equivocada, y la ejecución de MNIST del §22 mide el coste: 97,41 % con un learning rate de 1,0, frente a 98,02 % con 0,1.

Esta sección explica por qué. El mecanismo es una de las piezas de cálculo más útiles del machine learning práctico, y además es corto.

Qué falla con diez sigmoides

La versión con MSE da a cada dígito su propia salida sigmoide. Cada una responde de forma independiente «¿es esto un 7?» — y nada impide que las diez respondan «sí, 0,9». Eso es incoherente cuando la imagen es exactamente un dígito, y significa que la red gasta capacidad aprendiendo una restricción que la arquitectura debería haber impuesto gratis.

Softmax: hacer que las salidas compitan

Softmax toma los scores crudos de la última capa — los **logits**, sin cotas e ininteligibles por sí solos — y las convierte en una distribución de probabilidad:

$$p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Todas las salidas son positivas, y suman exactamente 1. Subir una *necesariamente* baja las otras, que es la restricción de la que carecen diez sigmoides independientes.

Dos propiedades importan para la implementación:

Es invariante a desplazamientos. Sumar una constante a cada logit no cambia nada — la constante sale factor común de cada numerador y del denominador, y se cancela. Esto no es una curiosidad; es la única razón por la que el código funciona, porque:

La fórmula ingenua se desborda. `exp(z)` es infinito para z por encima de unos 88 en float32, y logits de ese tamaño son corrientes en una red entrenada — dando `inf / inf = NaN`. Restar primero el logit más grande es exacto (por la invariancia a desplazamientos) y hace que el mayor exponente sea `exp(0) = 1`, de modo que nada puede desbordarse. `SoftmaxCrossEntropy.Transform` hace esto, y la batería de pruebas lo comprueba con logits de 1000.

Softmax no puede ser un `IActivation`. Todas las activaciones del §13 llevan un número a un número. Softmax necesita las salidas de toda la capa a la vez, por el denominador. Por eso vive en la pérdida y no en la capa — véase la nota sobre `ILoss.Transform`.

Cross-entropy: evaluar la distribución

Dada una distribución de probabilidad, la cross-entropy hace una sola pregunta: *¿qué probabilidad asignaste a la respuesta correcta?*

$$L = -\sum_j y_j \log(p_j) \overset{\text{one-hot}}{=} -\log(p_{\text{correcta}})$$

Acertar con confianza no cuesta nada. Equivocarse con confianza cuesta **sin límite** — `-log(0,001)` es 6,9 y subiendo, mientras que MSE limita la penalización a 1 por salida por muy rematadamente mal que estés. Esa diferencia de forma es lo que hace que el gradiente se comporte.

La fusión, que es todo el asunto

Deriva las dos por separado y ambas son desagradables:

- **Softmax sola** da un jacobiano completo — cada salida depende de cada logit, así que obtienes una matriz n×n por ejemplo en lugar de un vector.
- **La cross-entropy sola** da un término `1/p` que explota cuando p se acerca a 0 — exactamente donde vive una red rematadamente equivocada.

Compónlas y casi todo se cancela. Lo que sobrevive es:

$$\frac{\partial L}{\partial z_j} = p_j - y_j$$

Predicción menos objetivo. Sin jacobiano, sin división, nada que pueda desbordarse y — algo crítico — sin ningún factor `σ'(z)` que se desvanezca. Compáralo con la cadena de MSE sobre sigmoide del §22, cuyo gradiente está escalado por `a(1−a) ≤ 0,25`, que se desploma hacia cero precisamente cuando la red está más equivocada. El `1/p` de la cross-entropy disparándose y el jacobiano de softmax encogiéndose se cancelan **exactamente**.

Esa es toda la razón por la que los clasificadores se construyen así, y es por lo que la demo de MNIST entrena con un learning rate de 0,1 en lugar de 1,0.

Dos cosas que el código hace al respecto

Exige una capa de salida lineal. `p − y` es la derivada respecto a los *logits*. Si la última capa los comprimiera antes a través de una sigmoide, la fórmula sería sencillamente falsa — y falsa de la peor manera, ya que la red seguiría entrenando, solo que mal. Así que `SoftmaxCrossEntropy.Validate` rechaza cualquier capa de salida que no sea `Dense<Identity>`, una vez, en la construcción:

```
var net = new Sequential(inputs: 784)
    .Dense<Tanh>(128)
    .SoftmaxOutput(10)      // Dense<Identity> + softmax cross-entropy, emparejadas
        correctamente
    .Build();
```

`SoftmaxOutput` existe precisamente para que el emparejamiento no pueda equivocarse por accidente.

Para los casos que el atajo no cubre está `WithLoss`, que fija la pérdida de forma explícita y te deja a ti la capa de salida — así es como la batería de pruebas instala pérdidas rotas a propósito para demostrar que la verificación del gradiente puede fallar. Una red ya construida expone su elección como `Network.LossFunction`, que es también lo que `ModelIO` escribe en disco:

```
var net = new Sequential(inputs: 3)
    .Dense<Tanh>(6)
    .Dense<Identity>(4)
    .WithLoss(SoftmaxCrossEntropy.Instance)   // equivalente a SoftmaxOutput(4)
    .Build(seed: 7);

net.LossFunction.Name;   // "softmax-cross-entropy"
```

Está verificada por gradiente. Una cancelación tan cómoda es exactamente el tipo de álgebra que es fácil hacer *casi* bien, y el argumento del §21 se aplica con toda su fuerza: un gradiente casi correcto sigue entrenando. `SoftmaxGradientTests` ejecuta la misma comprobación por diferencias finitas, incluida la curva en U del error relativo, contra la fórmula fusionada. Esa prueba es la diferencia entre creerse la derivación y saberla.

La pérdida viaja con el modelo

Los pesos de un clasificador softmax carecen de sentido sin saber que hay que aplicarles softmax: cárgalo como una red normal y devuelve logits sin cotas donde quien llama espera probabilidades. No se lanza nada — los números simplemente están mal. Así que la pérdida se escribe en el archivo del modelo, que es lo que llevó el formato a la **versión 2**. Los archivos de versión 1 siguen cargándose, como MSE, que es lo que eran (§19).

Glosario

Término	Significado
Unidad / neurona	Una salida de una capa: suma ponderada + sesgo, y luego una activación
Peso	Con cuánta fuerza influye una entrada sobre una unidad
Sesgo (bias)	La salida base de una unidad antes de considerar las entradas
z (preactivación / logit)	La suma ponderada, antes de aplicar la activación
a (activación)	`g(z)` — la salida de la unidad
δ (delta)	`dL/dz` — el gradiente tras pasar de vuelta por la activación
Gradiente	Vector de derivadas: dirección de mayor aumento de la pérdida y magnitud de esa pendiente
Pérdida (loss)	Un único número que mide cuán equivocadas están las predicciones
Forward pass	Entrada → predicción
Backward pass	Pérdida → gradientes para cada parámetro
Época (epoch)	Una pasada completa sobre el dataset de entrenamiento
Mini-batch	Un grupo de ejemplos cuyos gradientes se promedian en una actualización
Learning rate (η)	Tamaño del paso del descenso de gradiente
Fan-in / fan-out	Número de entradas a / salidas de una capa; fija la escala de inicialización
Features	Variables de entrada o representaciones intermedias que la red usa para predecir
Ruptura de simetría	Inicialización aleatoria que permite a las unidades aprender características distintas
Linealmente separable	Separable por una sola recta/plano (AND sí, XOR no)
Saturación	Una activación clavada en su extremo plano, donde el gradiente ≈ 0
Gradiente que se desvanece	Gradientes que encogen hacia cero con la profundidad, estancando las primeras capas
Unidad muerta	Una unidad ReLU que produce 0 para toda entrada — gradiente permanentemente 0
Sobreajuste (overfitting)	Memorizar los datos de entrenamiento en lugar de aprender patrones generalizables
Hiperparámetro	Un valor que eliges en lugar de aprender (learning rate, tamaños de capa, épocas)
Softmax	Convierte logits en una distribución de probabilidad que suma 1 (§27)
Cross-entropy	Pérdida que evalúa la probabilidad asignada a la clase correcta (§27)
Logit	Score crudo de la capa de salida, antes de softmax

Término	Significado
One-hot	Objetivo codificado como todo ceros salvo un 1 en la clase correcta
SIMD	Una instrucción de CPU que opera sobre 4–16 números simultáneamente
GEMM	General Matrix Multiply — la operación batch sobre la que se construyen los frameworks reales