ZOFF.TECH

STUDY GUIDE · ENGLISH

Neural networks from scratch in C#

A complete, working neural network with *backpropagation*, written in C# with SIMD, and an explanation of **why every piece exists**, starting from zero assumed knowledge.

2,052	98.02%	27
LINES OF GUIDE	MNIST IN 37 S	SECTIONS + GLOSSARY

Fabricio Zoff · Zoff.tech

github.com/fabbrik/neural-network-csharp

Measured on: Apple M3 Pro (11 cores) · macOS 26.3 · .NET 10.0.10 · Arm64 RyuJIT

A complete, working neural network with backpropagation, written in C# with SIMD — and an explanation of **why every piece exists**, starting from zero assumed knowledge.

How to use this guide. Part I explains the ideas with no code — read it first, even if you're impatient. Part II maps each idea onto the actual C# and explains the engineering. Part III is practice, debugging, and exercises. The worked examples in §7 and §8 have been computed and verified; follow along with a calculator and you will genuinely understand backpropagation, which is the one concept everything else depends on.

File	Contents
`Activations.cs`	`IActivation` and the activation functions
`Perceptron.cs`	The 1958 single-unit perceptron and its update rule
`Dense.cs`	The dense (fully connected) layer: forward, backward, gradient step
`SimdOps.cs`	Vectorized `Dot` and `AddScaled`
`ILayer.cs`	Non-generic layer interface, so a network can mix activation types
`Network.cs`	Layer stack, mini-batch SGD training loop, MSE loss, `Summary()`
`Sequential.cs`	Keras-style fluent builder with input-size inference
`Loss.cs`	`ILoss`, mean squared error, and softmax cross-entropy
`ModelIO.cs`	Saving and loading trained models
`GradientCheck.cs`	Finite-difference verification of the backward pass
`Datasets.cs`	Generated two-moons data, for train/test experiments
`Program.cs`	Demos: perceptron, XOR, save/load, gradient check, two moons
`Idx.cs`	Reader for MNIST's IDX file format (big-endian, one-hot labels)
`MnistData.cs`	Downloads and caches MNIST; skips cleanly when offline
`ImageFile.cs`	PNG and Netpbm decoding, with no dependencies
`DigitPreprocessor.cs`	Puts any image into MNIST's conventions — §22
`NN.Mnist/Program.cs`	The digit recognizer: train, evaluate, read an image
`models/`	A trained recognizer, checked in so digit reading works on a fresh clone
`bench/`	Benchmarks behind every performance claim below

Contents

27. Softmax and cross-entropy
28. Glossary

> **If you only read three sections:** §7 and §8 (backpropagation worked by hand) and §21 (how you know it's right).
>
> **Where the claims are checked.** Every performance claim in Part II links to a measurement in `bench/README.md`, including one that turned out to be false and is corrected in §11. Numbers in this guide come from an Apple M3 Pro on .NET 10; see the note at the top of §22 on why yours may differ in the last digits.

Part I — The Concepts

1. What problem is a neural network actually solving?

You have examples of inputs paired with correct outputs:

Input (x)	Correct output (y)
0, 0	0
0, 1	1
1, 0	1
1, 1	0

You want a function that reproduces this mapping — and, more importantly, generalizes to inputs it hasn't seen. You don't know the formula. So instead of writing one, you:

1. Build a function with **thousands of adjustable numbers** in it (the *parameters*).
2. Define a score for how wrong its outputs are (the *loss*).
3. Work out, for every parameter, which way to nudge it to reduce the loss.
4. Nudge them all a little in that direction, and repeat.

That's the whole idea. Two names get attached to the end of that list, and it's worth keeping them straight from the start because they're constantly confused:

- **Backpropagation** is step 3 — *computing* the gradients efficiently. It's the clever part, and the thing this codebase exists to demonstrate.
- **Gradient descent** is step 4 — *applying* them.

Backprop tells you which way is downhill; gradient descent takes the step. Everything else in this repo is bookkeeping and speed.

> The network isn't "reasoning." It's a large parameterized formula being fitted to data, the same way you'd fit a line to points — just with far more parameters and a nonlinear shape.

2. The neuron

A single neuron (here: a **unit**) does exactly three things.

Step 1 — weighted sum. Each input gets a weight saying how much it matters, positive or negative. Add them up, plus a bias:

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

Think of it as a **vote**. Each input pushes the result up or down in proportion to its weight. The bias `b` is the neuron's baseline — its opinion before seeing any input at all. Without a bias, `z` would always be 0 when all inputs are 0, which is a needless restriction.

Step 2 — activation. Feed `z` through a nonlinear function `g`:

$$a = g(z)$$

Step 3 — output `a`, which becomes an input to the next layer.

Why the activation is not optional

This is the most commonly skipped point, and it's important.

Suppose you skip the activation, so each layer is just a weighted sum. Stack two layers: layer 2 computes a weighted sum of layer 1's outputs, which are themselves weighted sums of the inputs. A weighted sum of weighted sums is... **still just a weighted sum**. Algebraically you can collapse the two layers into one:

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

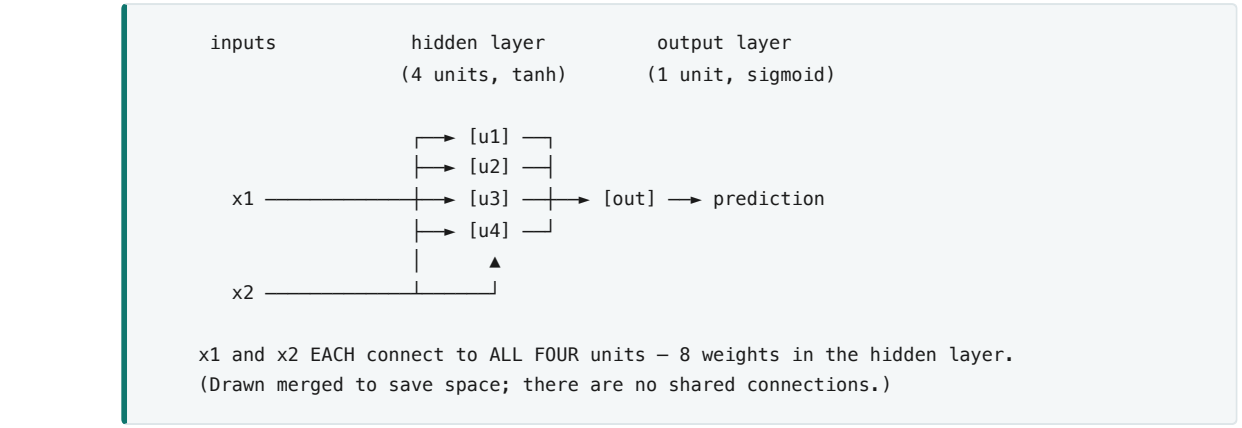
So a hundred stacked linear layers have exactly the power of a single linear layer — they can only draw straight lines. The nonlinear `g` between layers is what stops the collapse and is the entire reason depth buys you anything.

Nonlinearity is what makes a "deep" network deep. Remove it and depth is decoration.

---

3. Layers and the network

A **layer** is a group of units that all see the same inputs but have their own weights, so each learns to detect something different.



That is the exact XOR network in `Program.cs`.

- **Dense / fully connected** means every input connects to every unit. A layer with `n` inputs and `j` units therefore has `n × j` weights plus `j` biases.
- The **hidden layer** is "hidden" only in that its outputs are never observed directly — they're intermediate values feeding the next layer.
- Data flows left to right in the **forward pass**; error information flows right to left in the **backward pass**.

What do hidden units actually learn? Nobody assigns them jobs. Training discovers that some intermediate feature is useful and a unit drifts into computing it. That's the deal depth offers: **learn useful intermediate representations, then solve an easy problem on top of them.**

Textbooks usually say the XOR hidden units learn clean logic gates like OR and AND. Here is what the four units in *this* trained network actually compute (measured, not assumed — exercise 7 shows you how to print this yourself):

```
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

output layer weights:
        -2.243    2.737   -4.753   -4.449
```

Read it and the tidy story falls apart:

- **u1 and u2 are near-perfect mirror images** (u2 ≈ −u1 on every row). Two units learned the same feature with opposite sign — pure redundancy. Their output weights (−2.24 and +2.74) have opposite signs too, so they reinforce rather than cancel, acting as one feature with an effective weight near −5.

- **u3 is roughly NOR** — strongly positive only when both inputs are 0.
- **u4 roughly tracks x2** — positive whenever x2 = 1.

So it isn't OR-and-AND. It's *a* valid decomposition among many, found by gradient descent from one random starting point, and a different seed produces a different one.

This is the more useful lesson. Learned representations are typically redundant, partially duplicated, and only loosely interpretable. Reading meaning into individual hidden units is a whole research area (*interpretability*) precisely because it's genuinely hard — the network is under no obligation to organize itself the way a human would.

---

4. Learning = gradient descent

Now the core question: *how* do you adjust thousands of parameters in the right direction?

The hill-descending picture

Imagine the loss as a landscape. Every parameter is a direction you can move in, and the altitude is how wrong the network is. You want the lowest valley. You're in thick fog and can only feel the slope under your feet.

The strategy: **feel which way is downhill, take a small step, repeat.**

That's gradient descent. The **gradient** is the slope — and calculus gives it to us exactly, no guessing required:

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

Read this as: *new weight = old weight − learning rate × slope of loss with respect to that weight.*

Why the minus sign

The derivative `∂L/∂w` answers: "if I increase `w` slightly, does the loss go up or down?"

- Slope **positive** → increasing `w` increases loss → so **decrease** `w`.
- Slope **negative** → increasing `w` decreases loss → so **increase** `w`.

Both cases are handled by subtracting the gradient. The minus sign is what makes it *descent*.

The learning rate η

How big a step to take.

- **Too small** → training crawls; thousands of wasted epochs.
- **Too large** → you overshoot the valley, bounce up the far side, and the loss explodes to `NaN`.
- **Just right** → steady decrease.

There's no formula for it. `0.1` to `0.5` is a reasonable starting range for a small network like this one. If your loss ever goes to `NaN` or infinity, **lower the learning rate first** — it's the cause the overwhelming majority of the time.

One crucial caveat

Each parameter's derivative assumes **everything else stays fixed**. That's only true for an infinitesimal step. Take a big step and all parameters move at once, and the terrain you measured is no longer the terrain you're standing on. This is exactly why steps must be small and why the process is iterative rather than one-shot.

---

5. Measuring wrongness: the loss function

The loss turns "how wrong was this prediction" into one number to minimize.

Mean squared error (MSE), what this implementation uses:

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

Take the difference between prediction and target, square it, average over outputs. Squaring does two useful things: it makes errors positive (so $+0.3$ and -0.3 are equally bad rather than cancelling), and it punishes large errors disproportionately.

Its derivative — the seed of the entire backward pass:

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

Sensible: if prediction exceeds target, the derivative is positive, meaning "lower this output."

MSE is the natural choice for regression (predicting numbers). For classification, **cross-entropy** is better, and this codebase implements it — see §27.

---

6. The chain rule — the one piece of calculus you need

Backpropagation is the chain rule applied carefully. If you understand this section, the rest is mechanical.

The rule: if `a` depends on `z`, and `z` depends on `w`, then

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

The intuition: rates of change multiply along a chain. If a car is twice as fast as a bike, and the bike is three times as fast as walking, the car is six times walking speed. The same composition applies to "how much does nudging `w` move `L`."

In a neural network the chain is exactly:

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

so the influence of any weight on the final loss is the product of every local rate of change along the path from that weight to the loss.

The three factors in our case are all easy:

Factor	What it is	Value
∂L/∂a	How much loss changes with this unit's output	Comes from the loss, or from the layer above
∂a/∂z	How much output changes with the weighted sum	`g'(z)` — the activation's derivative
∂z/∂w	How much the weighted sum changes with this weight	`x` — just the input!

That last one is worth pausing on. Since `z = w₁x₁ + w₂x₂ + b`, the derivative with respect to `w₁` is simply `x₁`. **A weight's gradient is proportional to the input it multiplies.** Which is intuitive: a weight attached to an input that was zero had no effect on this prediction, so it gets no blame and no update.

7. Worked example: one neuron, by hand

Let's do a complete forward and backward pass with real numbers. *These values are computed and verified — follow along with a calculator.*

Setup. One neuron, two inputs, sigmoid activation.

```
inputs   x = [1.0, 0.5]
weights  w = [0.3, -0.2]
bias     b = 0.1
target   y = 1.0
learning rate η = 0.5
```

> **A note on the `/m`.** §5 gave the loss derivative as `2(a − y)/m`, where `m` is the number of outputs. This network has one output, so `m = 1` and the division vanishes. That's why the examples below use plain `2(a − y)` — they aren't dropping a term, `m` just happens to be 1. The code in `Network.cs` always divides properly.

Forward pass

Weighted sum:

$$z = (0.3 \times 1.0) + (-0.2 \times 0.5) + 0.1$$
$$= 0.3 - 0.1 + 0.1$$
$$= 0.3$$

Activation (sigmoid: $1 / (1 + e^{-z})$):

$$a = 1 / (1 + e^{-0.3}) = 0.574443$$

Loss:

$$L = (a - y)^2 = (0.574443 - 1.0)^2 = (-0.425557)^2 = 0.181099$$

The network predicted 0.574 where 1.0 was wanted. Now we fix it.

Backward pass

Step 1 — how does loss change with the output?

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

Negative, meaning: *increasing* `a` *would decrease the loss.* Correct — we want `a` to grow toward 1.0.

Step 2 — push back through the activation. Sigmoid's derivative is `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) is the single most important intermediate value in backprop. It means: "**how much does the loss change per unit change in this neuron's** `z`**.**" Once you have δ for a neuron, every gradient it's involved in is one multiplication away.

Step 3 — the parameter gradients. Multiply δ by each input:

```
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
```

Notice `w₁`'s gradient is **half** of `w₀`'s — because its input was 0.5 instead of 1.0. It had half the influence, so it gets half the correction. The bias gradient is just δ, since a bias is a weight on a constant input of 1.

The update

Subtract `η × gradient` from each parameter:

```
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
```

Did it work?

Run the forward pass again with the new parameters:

```
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
```

Loss dropped from 0.181099 to 0.136581, and the prediction moved from 0.574 to 0.630 — closer to the target of 1.0.

That is a full training step. A network is this repeated across every neuron and every example, thousands of times.

---

8. Worked example: the chain across two layers

The above handled one layer. The essential new idea in a *deep* network is propagating the gradient **backwards into the previous layer**. Here's the smallest possible case, also verified.

Setup. One input → one tanh hidden unit → one sigmoid output unit.

```
x = 0.8      w₁ = 0.5, b₁ = 0.0   (hidden, tanh)
             w₂ = 1.2, b₂ = −0.3  (output, sigmoid)
target y = 0.0
```

Forward

```
z₁ = 0.5 × 0.8 + 0.0 = 0.400000
h  = tanh(0.4)       = 0.379949      ← hidden unit's output

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

L  = (0.538906 − 0.0)² = 0.290420
```

Backward — output layer first

```
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
```

Backward — now cross into the hidden layer

This is the step that makes it "backpropagation." The hidden unit influenced the loss only *through* the output unit, so we route the gradient back through the connecting weight `w₂`:

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

Read that as: "*the hidden unit's output affects the loss by δ_2 scaled by the strength of the connection carrying it forward.*" The same weight `w2` that carried the signal **forward** now carries the blame **backward**. That symmetry is the heart of the algorithm.

Then it's the identical recipe as before, one layer down:

```
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
```

The pattern

Every layer, without exception, does:

1. Receive `dL/da` from above (or from the loss, if it's the last layer).
2. `δ = dL/da × g'(z)` — push through the activation.
3. `dL/dW += δ × input`, `dL/db += δ` — record the parameter gradients.
4. `dL/dinput = δ × W` — hand backwards to the layer below, which returns to step 1.

With 2 layers or 200, that loop is the entire algorithm. `Dense.Backward` is a literal transcription of those four lines.

Why it's efficient

Naively you might compute each weight's gradient separately by re-running the network — with a million weights that's a million forward passes. Backprop computes **all** gradients in a single backward sweep by reusing the shared δ values. It costs roughly the same as one forward pass. That efficiency is why neural networks are trainable at all, and it's why the 1986 popularization of backprop restarted the entire field.

---

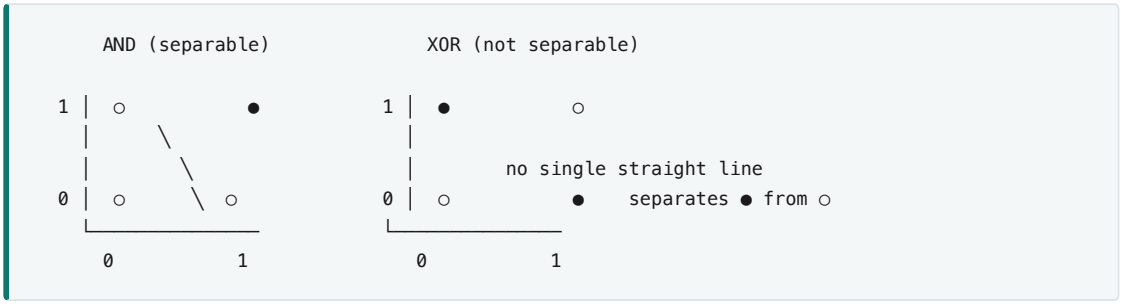
9. The perceptron, XOR, and why this history matters

The **perceptron** (1958) is the ancestor of all this: one unit, step activation, and its own update rule.

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

Since the step function outputs only 0 or 1, the error `(y - ŷ)` is always −1, 0, or +1: "too high," "correct," or "too low." `Perceptron.Train` skips correct predictions entirely and stops early once a full epoch passes with no updates.

The convergence theorem: if the data is *linearly separable* — separable by a single straight line — this is guaranteed to converge in finite steps. If it isn't, it oscillates forever.



Minsky and Papert's 1969 book *Perceptrons* analyzed this limitation rigorously, and it's widely credited with cooling enthusiasm for neural networks through the 1970s.

Treat that story with some caution — it's repeated everywhere in simplified form. Minsky and Papert did discuss multilayer networks rather than ignoring them, and historians generally attribute the broader "AI winter" funding collapse more to the 1973 Lighthill report and unmet expectations across AI as a whole. The mathematical result is solid; the tidy cause-and-effect narrative around it is contested.

The fix is the two things this codebase adds: a hidden layer (so the network can build its own features) and a smooth activation (so backprop can compute gradients through it).

You can watch the whole story run in `Program.cs`: the perceptron nails AND in 4 epochs, and the network with one hidden layer solves XOR that the perceptron provably cannot.

Part II — The C# Implementation

10. Reading order

If you're reading the code for the first time, go in this order:

1. `Activations.cs` — activations. Small, self-contained, and each one matches the table in §13.
2. `Dense.cs` → `Forward` — §2's math, literally.
3. `Dense.cs` → `Backward` — §8's four-step pattern, literally.
4. `Network.cs` — the training loop that drives it all.
5. `SimdOps.cs` — pure optimization. **Skip on the first pass**; it changes nothing conceptually.

11. C# features used here that you may not have met

These are the constructs that make the code look unfamiliar. None are essential to the math.

`Span<float>` — a window into memory

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

A `Span<float>` is a **view** of a slice of an array — a pointer plus a length. It copies nothing. `weights.AsSpan(10, 4)` refers to elements 10–13 of the original array; writing through the span writes the original.

Why it's everywhere here: training calls `Forward` millions of times. Returning a new `float[]` each call would allocate millions of arrays and bury you in garbage collection. Spans let one pre-allocated buffer be reused forever. `ReadOnlySpan` additionally documents "I only read this."

The catch: a span can't be stored in a class field or used inside `async` methods (it's a `ref struct`, stack-only). That's why the layers keep `float[]` fields and hand out spans.

Generic structs as policy — `Dense<TActivation>`

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

The activation is a **type parameter**, not a field. `Dense<Tanh>` and `Dense<Sigmoid>` are different types, and the JIT compiles separate machine code for each, inlining the activation directly into the loop.

The obvious alternative is a `Func<float, float>` field. It costs an indirect call for every unit of every layer of every example, and the compiler can't inline through it.

> **An earlier draft of this guide told you that made it slower. It doesn't, and the correction is more instructive than the original claim.** Benchmarked against a delegate-dispatched layer, the generic version is faster by 0–6% — noise at any realistic size (full table).
>
> The reason is a ratio, not a dispatch cost. The activation runs **once per unit**; the dot product feeding it runs `Inputs` multiply-adds per unit. On a 784-input layer, one un-inlinable call is amortized over 784 fused multiply-adds — it is invisible because it is *rare*, not because it is fast. The first suspicion was that tanh, a transcendental, was hiding the call, so the benchmark repeats it with ReLU (a compare and a select). The result barely moves.
>
> **The lesson generalizes past this repo:** "un-inlinable indirect call" is a statement about codegen, and "slower" is a statement about the program. Getting from one to the other requires knowing how often the call happens relative to everything else — which is what a profiler tells you and intuition does not. This is also §12 in miniature: the effects that matter are the ones in the innermost loop, and the activation isn't in it.

So keep the generic design, but for its real merits: `readonly struct` activations compose at zero cost, nothing allocates, and the type system prevents `Dense<Tanh>` and `Dense<ReLU>` from being confused. Not for speed.

The enabling feature is **static abstract interface members** (C# 11):

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

An interface method with no object attached. It lets `TActivation.Apply(z)` be resolved and inlined at compile/JIT time.

`readonly struct` activations

`Sigmoid`, `Tanh`, and the rest are empty structs. They hold no data and are never instantiated — they exist purely as *names* to feed the generic system. Zero runtime cost.

The JIT

C# compiles to IL (bytecode), and the JIT compiler turns IL into machine code at runtime. It inlines small methods, picks SIMD instructions for the actual CPU, and — importantly here — generates a **separate code copy per value-type generic instantiation**. That's what makes `Dense<Tanh>` fast, and also why §14 keeps the SIMD helpers *out* of the generic class.

---

12. Data layout — the most consequential decision

NumPy stores `W` as `(n, j)`: **n features × j units**. Unit `j`'s weights are the *column* `W[:, j]`, whose elements sit `n` floats apart in memory.

This C# version stores the **transpose**, flattened into one array:

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

Unit `j`'s weights are `Weights[j * Inputs .. (j+1) * Inputs]` — **contiguous**. See `Dense.UnitWeights`.

Why this matters so much

A CPU never loads one float. It loads a **64-byte cache line** (16 floats) at a time, because memory is vastly slower than arithmetic — a main-memory fetch costs hundreds of cycles while a multiply costs one.

- **Strided (NumPy column) access:** load 16 floats, use 1, discard 15. You waste ~94% of your memory bandwidth, and SIMD can't be used without an expensive gather instruction.
- **Contiguous access:** load 16 floats, use all 16, and a single SIMD instruction processes 4–16 of them at once (§15).

> **Rule of thumb:** arrange data so the innermost loop walks consecutive addresses. This one habit is worth more than every other optimization in this file combined.

What it's actually worth — and where it's worth nothing

That rule of thumb is a strong claim, so it's measured. Same weights, same arithmetic, same activation; the only difference is memory order (details):

Layer shape	Unit-major (contiguous)	Feature-major (strided)	Cost of striding
2 × 4 — the XOR layer	19.1 ns	16.0 ns	**0.84× — striding *wins***
64 × 64	525 ns	3249 ns	**6.2×**
784 × 128 — MNIST-sized	9.75 μs	72.6 μs	**7.4×**

At realistic sizes the claim holds comfortably: 6.2–7.4×, the largest single effect in the codebase, and larger than the SIMD win it partly enables.

But look at the first row. On the XOR layer the "bad" layout is *faster*. Eight weights are 32 bytes — they fit inside one 64-byte cache line, so there is no wasted bandwidth to save and no gather to avoid. All that remains is the SIMD path's loop setup, which the plain strided loop skips.

This is worth dwelling on, because the XOR demo is the first thing you run:

> **Every optimization in this guide is worth nothing at the scale of the example that introduces it.** Cache behaviour is a property of data that doesn't fit in cache. Vectorization is a property of loops long enough to amortize their setup. Below those thresholds you are measuring overhead, and the "obviously worse" implementation frequently wins.
>
> The corollary is the useful one: **the size at which you benchmark determines the answer you get.** A benchmark of the XOR layer would have told you to delete the contiguous layout.

One flat array, not jagged. `float[]` beats `float[][]` here: one allocation instead of `j`, one bounds check instead of two, no pointer chase per unit, and the whole matrix sits in one contiguous block the prefetcher can stream. (Not separately benchmarked — the jagged variant would confound layout with allocation, and the flat array is no harder to write.)

13. Activations in code

Activation	g(z)	g'(z) in terms of a = g(z)	Range	Use for
Sigmoid	$1 / (1 + e^{-z})$	`a(1 − a)`	(0, 1)	Output layer, binary probability
Tanh	tanh(z)	`1 − a²`	(−1, 1)	Hidden layers
ReLU	max(0, z)	`a > 0 ? 1 : 0`	[0, ∞)	Hidden layers, deep nets
Identity	z	`1`	ℝ	Regression output
Step	z ≥ 0 ? 1 : 0	*throws*	{0, 1}	Perceptron only

Why derivatives are written in terms of `a`, not `z`

Every activation here has a derivative recoverable from **its own output**. Sigmoid's derivative is `a(1−a)`; tanh's is `1−a²`. The forward pass already computed `a`, so the backward pass reuses it instead of recomputing `exp()` or `tanh()`.

This is a real saving, not a micro-optimization: transcendental functions cost 20–100× a multiply. It's also why `Forward` caches its outputs — see §14.

Why `Step` throws instead of returning 0

Its derivative is 0 everywhere it's defined (the function is flat), and undefined at the jump. Returning 0 would multiply every gradient in the network by zero, and training would appear to run while learning **nothing** — a silent, maddening bug.

Throwing makes the lesson explicit: **the step function's flatness is precisely why the perceptron needs its own update rule, and why backprop requires smooth activations.** Gradient descent needs a slope to follow; a staircase has none.

Choosing one

- **Hidden layers → tanh or ReLU.** Tanh is zero-centered, meaning outputs span (−1, 1). Sigmoid's outputs are all positive, which makes every weight gradient into a unit share a sign, so optimization zigzags instead of heading straight downhill.

- **Output layer → match the task.** Sigmoid for a probability in (0,1); identity for unbounded regression.
- **Sigmoid and tanh saturate:** for large `|z|` the curve flattens, `g'(z) ≈ 0`, and gradients nearly vanish. Stack many such layers and the gradient reaching the early layers is effectively zero — the **vanishing gradient problem**. ReLU's derivative is exactly 1 for positive inputs, which is why deep networks moved to it.

---

14. The forward pass in code

`Dense.Forward`:

```
SimdOps.MatVec(Weights, aIn, Bias, aOut);   // dest[j] = dot(weights_j, x) + bias[j]
TActivation.ApplyAll(aOut);                 // then activate the whole vector at once
```

and `MatVec` is §2 line for line — slice the unit's weights, dot with the input, add bias:

```
for (int j = 0; j < dest.Length; j++)
    dest[j] = Dot(weights.Slice(j * n, n), x) + bias[j];
```

Compare it to the original NumPy and the correspondence is exact — only the layout changed.

Why the activation is a second pass rather than the last line of that loop. `exp` and `tanh` cost tens of cycles each, and a layer calls one per unit; applied to the output vector as a whole they are four at a time. That is worth 2× on the activation step and ~1.3× on the layer (§25, benchmarks). The `NoInlining` on `MatVec` is not decoration either — without it this exact code runs **6× slower**, for reasons the benchmark README documents in detail.

The two forward passes, and the bug that split them

Backprop needs both the inputs that produced these activations (for `dL/dW = δ × input`) and the activations themselves (for `g'` from `a`). So something has to cache them, and the obvious place is the bottom of `Forward`:

```
aIn.CopyTo(_lastInput);      // what the code used to do
aOut.CopyTo(_lastOutput);
```

That is a trap, and it is worth understanding because the same shape appears everywhere. The cache is written by `Forward` but read by `Backward`, an arbitrary distance later. Anything that runs a forward pass in between overwrites it. And plenty of reasonable things do:

```
net.AccumulateGradients(x, y);
Console.WriteLine(net.Predict(somethingElse)[0]);   // ← silently destroys the cache
net.ApplyGradients(lr, 1);                          //   gradients now describe the wrong example
```

Nothing throws. The loss still falls. The network just learns something subtly wrong — which is exactly the failure mode §21 exists to catch, arriving by a different route.

The fix is to make it unrepresentable rather than to document it. There are now two methods:

	caches?	used by
`Forward`	no	`Predict`, `Loss`, `ForwardBatch`, `GradientCheck` — all inference
`ForwardTrain`	yes	`AccumulateGradients` only

`Backward` consumes the cache and clears it, so a second `Backward`, or one with no preceding `ForwardTrain`, throws instead of quietly differentiating a stale example. `GradientCheck` depends on this directly: it evaluates the loss twice per parameter while analytic gradients sit in the accumulators, which is only safe because `Loss` cannot touch training state.

> **The general lesson:** when a method's *side effect* is consumed by a different method later, the coupling is invisible at both call sites. Prefer splitting the method over documenting the ordering. A comment saying "don't call `Predict` here" is a bug waiting for someone who didn't read it.

The memory cost. The cache is also why training needs far more memory than inference: **you cannot free forward activations until the backward pass has consumed them.** That is what "batch size too large" means in practice — out of memory, holding every example's activations at once.

---

15. SIMD — doing several multiplications at once

Skip this section on a first read. It's pure speed; the math is unchanged.

SIMD = *Single Instruction, Multiple Data*. Modern CPUs have wide registers holding several floats at once, and one instruction operates on all of them simultaneously.

How many depends on your CPU:

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

`Vector<float>` from `System.Numerics` exposes this portably — you write the code once and it adapts. Check your own machine with:

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

This is exactly why the code never hardcodes a width: every loop in `SimdOps.cs` reads `Vector<float>.Count` at runtime, so the same source runs at full width on both ARM and x86.

> **Common misconception, and the one that started this project's rewrite:** `Vector<float>` is **not** a variable-length mathematical vector. It's a fixed-width hardware register. It cannot store a layer's weights — it's a pipe you stream data through, `Count` floats at a time.

The dot product (`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));
}
// ... then a single-width loop, a horizontal sum, and a scalar tail
```

Why two accumulators. A multiply-add instruction has ~4 cycles of *latency* but can be *issued* every cycle. With a single accumulator, each iteration must wait for the previous one's result — you get one result per 4 cycles, a quarter of

peak. Two independent chains let the CPU keep four operations in flight. This is **instruction-level parallelism**, and it costs one extra register.

The scalar tail handles a length that isn't a multiple of the SIMD width. Every hand-written SIMD loop needs one; forgetting it silently drops the last few elements.

Does any of it work?

Both claims above — that vectorizing pays, and that the second accumulator pays *again* — are benchmarked against a scalar loop and against a single-accumulator SIMD version (details):

Length	Scalar	1 accumulator	2 accumulators	SIMD win	2nd accumulator win
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×**

Three things to notice.

The SIMD win exceeds the vector width. `Vector<float>` is only 4 wide on the ARM machine these numbers come from, yet the speedup reaches 5.9×. Vectorizing doesn't just do 4 multiplies at once — it also quarters the loop bookkeeping and the bounds checks. Getting *more* than the width is normal; the width is a floor, not a ceiling.

The second accumulator earns its keep at any useful length. At 1.2–1.5× it is the second-largest optimization in the codebase after data layout, which is a lot for one extra register.

> **A footnote worth more than the table.** An earlier revision of this guide reported that first row as **0.83× — the second accumulator actively costing 17%** — and spent a paragraph explaining why eight floats is exactly the length at which it should lose. The explanation was plausible and the number was noise: those rows had been measured with BenchmarkDotNet's three-iteration `--job short`, whose error bar was wider than the effect being described. Re-measured properly it wins 1.23×, like every other row.
>
> Two lessons, and the second is the expensive one. **A short job is for triage, not conclusions.** And **a confident mechanism is not evidence** — the story about setup costs and tail iterations was convincing enough that nobody re-ran the measurement it was invented to explain.

`AddScaled` — `dest += src × scale`

`SimdOps.AddScaled` is the workhorse of the *backward* pass. Notice from §8 that steps 3 and 4 are both "add a scaled vector into an accumulator" — the same primitive serves weight-gradient accumulation, input-gradient propagation, and the descent step itself.

Unlike `Dot`, this one is *not* hand-rolled: it calls `TensorPrimitives.MultiplyAdd` from `System.Numerics.Tensors`, which is **2.5× faster** than the `Vector<float>` loop it replaced. The split between the two is worth understanding, because it is not a matter of taste:

	`Dot`	`AddScaled`
Shape	reduction — one value out	streaming — one value per element
`TensorPrimitives` version	**1.5× slower** at 4096	**2.5× faster** at 4096
Why	carries a single accumulator chain through the reduction — the exact serial dependency the second accumulator exists to break	no dependency chain to carry, unrolls freely, and emits a real fused multiply-add

So the library uses the runtime's kernel for one of its two primitives and its own loop for the other. Neither answer was predictable from reading the API; both came from the benchmarks. "**Use the optimized library function" is a hypothesis, not a conclusion.**

Why these live outside `Dense<T>`

The JIT emits a separate code copy per value-type generic instantiation (§11). Left inside `Dense<TActivation>`, `Dot` would be duplicated for `Dense<Tanh>`, `Dense<ReLU>`, `Dense<Sigmoid>`… bloating the instruction cache for no benefit. A non-generic helper class gets exactly one copy.

---

16. The backward pass in code

`Dense.Backward` — §8's four-step pattern, transcribed:

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

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

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

Three things worth noticing:

Steps 3 and 4 are the same operation with weights and inputs swapped. `dL/dW += δ·x` and `dL/dx += δ·W` are mirror images — that symmetry is why one `AddScaled` primitive covers both. And thanks to the layout from §12, *both* walk memory contiguously. The decision that sped up the forward pass pays off twice more here.

`gradIn` is empty for the first layer. Nothing consumes the gradient with respect to the raw input data, so computing it would be wasted work. (In some techniques — adversarial examples, style transfer — that input gradient is exactly what you want. Not here.)

`if (delta == 0f) continue;` skips units contributing nothing. With ReLU this is common: any unit whose output was clamped to 0 has a zero derivative, so it accumulates nothing. Which also flags a real failure mode — see §23.

Accumulate now, apply later

`Backward` only **adds into** `_weightGrads`. It never touches `Weights`. `ApplyGradients` performs the actual descent step and clears the accumulators:

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

Why separate them? Because it lets you sum gradients over several examples before updating — mini-batching, §20. Dividing by batch size averages rather than sums, so your learning rate keeps working when you change batch size.

This split is not an idiosyncrasy of this code. PyTorch divides at exactly the same seam: `loss.backward()` accumulates, `optimizer.step()` applies. If you move to a real framework, this will look familiar.

---

17. Weight initialization — genuinely not optional

`Dense.Initialize` uses **Xavier/Glorot uniform**:

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

Why identical weights fail — the symmetry problem

Start every weight in a layer at the same value and every unit computes the identical output. Identical outputs earn identical gradients. Identical gradients produce identical updates. **The units stay identical forever.**

A 100-unit layer initialized this way has the expressive power of *one* unit, permanently — no amount of training escapes it, because nothing breaks the tie. Random initialization **breaks the symmetry**: each unit starts different, receives a different gradient, and specializes.

Why all-*zero* weights fail even harder

Zero is a special case, and it's worth separating because it's the one you can run (exercise 2). It doesn't merely collapse the layer to one effective unit — it stops learning **entirely**:

- Every hidden unit outputs `g(0)`, and the gradient handed back to the hidden layer is `δ·W = δ·0 = 0`.

- The hidden weight gradient is `δ·x` with `δ = 0`, so hidden weights never move.
- With `tanh`, the hidden outputs are `tanh(0) = 0`, so the output layer's weight gradients (`δ·h`) are zero too.

Nothing but the output bias can move, and on XOR even that stays put — the four examples' gradients cancel exactly. Run exercise 2 and you'll see loss frozen at **exactly 0.250000** with every prediction 0.5000: the network permanently guessing the mean, having learned literally nothing.

Biases can safely start at zero — the random weights already broke the symmetry.

Why that particular scale

Too large and activations saturate at ±1 where gradients vanish; too small and the signal decays toward zero as it passes through layers. Xavier's `6/(fan_in + fan_out)` keeps activation variance roughly constant across layers, so signals neither explode nor vanish with depth. For ReLU, **He initialization** (variance `2/fan_in`) is the better-matched choice, since ReLU discards half its input range.

---

18. The perceptron in code

`Perceptron` is the odd one out: it's the only thing here that doesn't use backpropagation. It exists to make §9's history concrete and runnable.

It's a `Dense<Step>` of exactly one unit, wrapped in its own training rule:

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

The training loop (`Perceptron.Train`) is the 1958 rule verbatim:

```
float error = y[i] - Predict(xi);
if (error == 0f) continue;          // correct prediction: no update at all

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

Three things to notice, all of which contrast instructively with backprop:

No derivative appears anywhere. Compare with `Dense.Backward`, where every gradient is multiplied by `g'(z)`. The perceptron rule doesn't need one — which is exactly as well, because `Step.DerivativeFromOutput` throws (§13). This is the same fact from two directions: a step function has no usable slope, so gradient descent can't work on it, and the perceptron sidesteps that by not being gradient descent.

`error` is only ever −1, 0, or +1, since both prediction and target are 0 or 1. It's a direction — "too high", "correct", "too low" — not a magnitude. Backprop's δ carries magnitude as well, which is what lets it say *how much* to correct.

Correct predictions cause no update, and an epoch with no updates ends training early. That early exit is what makes "converged in 4 epochs" a meaningful statement rather than just the epoch limit being hit — and it's why the XOR case runs the full budget instead.

The tests pin both behaviours: `Perceptron_converges_on_linearly_separable_data` and `Perceptron_cannot_learn_xor`. The second asserts *failure*, which is unusual and deliberate — it locks in a property of the algorithm that the rest of the library exists to overcome.

---

19. The network, and the Sequential builder

`Network` holds `ILayer[]`. This is a **sequential** model in exactly the Keras sense: a strictly linear chain, each layer's output feeding the next, no branching.

Building one

`Sequential` is a fluent builder that mirrors the Keras API you'll meet in courses and tutorials:

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

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

The builder's real job is **input-size inference**. You declare the input width once; each layer takes its input count from the previous layer's unit count. The direct constructor still works and does the same thing, but makes you state — and keep consistent — both ends of every layer:

```
var net = new Network(seed: 42,
    new Dense<Tanh>(inputs: 2, units: 4),
    new Dense<Sigmoid>(inputs: 4, units: 1));   // the 4 must match by hand
```

Two shape errors are impossible to express through the builder and merely caught by the constructor: `.Add(layer)` rejects a layer whose `Inputs` doesn't match the current width, and `Build()` rejects an empty stack.

`Summary()`

Like Keras' `model.summary()`:

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

Check the arithmetic yourself — it's the formula from §3. The tanh layer has 2 inputs × 4 units = 8 weights, plus 4 biases = 12. The output layer has 4 × 1 = 4 weights plus 1 bias = 5.

Saving and loading a trained model

Training produces one thing of value: the weights. `ModelIO` writes them, together with enough architecture to rebuild the network:

```
ModelIO.Save(net, "xor.nnm");            // after training
var loaded = ModelIO.Load("xor.nnm");    // later, or in another program
float p = loaded.Predict(input)[0];      // ready to use immediately — no training needed
```

The XOR model is **127 bytes**, and the reload is bit-for-bit identical: same inputs, same outputs, exactly.

The file format, little-endian:

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

Five decisions in there are worth understanding, because they're the ones people get wrong:

Save the architecture, not just the weights. A bare float dump has no idea what shape it was, so loading it into mismatched code silently produces garbage. Storing layer types and shapes means a mismatch is caught immediately.

Magic bytes and a version number. Without them, any wrong file gets interpreted as floats and "works" until the predictions turn out to be nonsense. With them, you get "*Not a model file*" or "*Model format version 99 is not supported*". Truncated files are caught too, naming the layer that ran out of data. **Fail loudly on bad input** is the general lesson; model files are just a place where it's easy to skip.

A descriptor→constructor table (`ModelIO.Factories`) turns the string `"Dense<Tanh>"` back into a real type. C# can't construct a generic type from a string without either this table or reflection, and an explicit table is both faster and safer — reflection would let a malicious file name *any* type in your process. `ModelIO.Register` extends it for custom layers.

Loading must not initialize. The public `Network` constructor randomizes weights, so constructing a network from loaded layers would destroy the parameters you just read. That's why `ModelIO` calls `Network.FromTrainedLayers`

, which skips initialization. It's a real bug I hit writing this, and it's nasty precisely because it doesn't crash — you'd just get an untrained network that loads "successfully."

Save the loss, not just the layers — the change that took the format to version 2, and the case the version field was put there for. A softmax classifier's weights are meaningless without knowing softmax applies to them: load one as a plain network and it returns unbounded logits where the caller expects probabilities. Nothing throws, `Predict` still returns ten numbers, and only their *values* are wrong (§27). Version 1 files predate any choice of loss and still load — as mean squared error, which is what they were. **A format version earns its keep the first time you need to add a field**, and "old files keep working" is the whole return on having written it down.

> **What is *not* saved:** gradient accumulators and the forward-pass activation cache. Those are training scratch space, rebuilt empty on load. If you later add momentum or Adam, their state would also need saving to resume training mid-run — see §25 item 7.

Where sequential stops being enough

An array walked front-to-back can only express a chain. Skip connections (ResNet), multiple inputs or outputs, and concatenation all need a **graph** of layers with a topological sort instead of these two loops — that's what Keras' functional API is for. Everything in an introductory course is sequential, which is why Keras makes it the default.

The layer stack itself

Why a non-generic interface exists alongside the generics. `Dense<Tanh>` and `Dense<Sigmoid>` are *unrelated types* — you cannot put them in the same array. Generics give speed; the `ILayer` interface gives heterogeneity. You need both, so the code has both.

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

Backward: walk the array in reverse, each layer's `gradIn` becoming the next one's `gradOut`:

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

That single line is §8's "hand the gradient backwards" step, generalized to any depth.

The constructor validates that adjacent layer shapes match — layer `i`'s `Inputs` must equal layer `i-1`'s `Units` — and fails immediately with a clear message rather than producing nonsense later. It also pre-allocates every buffer, so training allocates nothing per example.

Threading, and who owns the buffers

Those pre-allocated buffers are what make training allocation-free, and they are also the reason for two rules that the API cannot enforce for you.

One network per thread. `Network` and `Dense` hold mutable activation buffers, gradient accumulators, the forward-pass cache, and the shuffle order. None of it is synchronized, and none of it is safe to share. Two threads calling `Predict` on one network will interleave writes into the same activation array and both get nonsense — with no exception, because nothing is technically wrong at the type level.

The one exception is `Dense.Forward` on a standalone layer, which after the §14 split writes no instance state at all. `Network.Predict` still writes shared activation buffers, so one network per thread remains the rule. (§25 item 4 notes that the library never *uses* threads either — there's no `Parallel.For` anywhere.)

`Predict` lends you a buffer; it doesn't give you one.

```
ReadOnlySpan<float> a = net.Predict(x1);
ReadOnlySpan<float> b = net.Predict(x2);   // a and b are the SAME memory — both now hold x2's
        result
```

The returned span views `_activations[^1]`, which the next forward pass overwrites. Copy it if you need to keep it:

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

This is the standard trade for zero-allocation APIs, and `Span<T>` is exactly the type that makes it explicit — see §11. `ModelIO.Register` has the same character at process scope: table access is synchronized, but registering custom layer types during start-up keeps load behavior independent of timing.

---

20. The training loop

Per **epoch** (one full pass over the data):

1. **Shuffle** the example order.
2. For each **mini-batch**: accumulate gradients over its examples, then apply once.

Why shuffle

With a fixed order the network can learn the *order* rather than the data, and consecutive correlated samples (all of class A, then all of class B) yield biased gradient estimates that make training lurch. Shuffling makes each batch a fairer sample of the whole dataset.

Batch size — the three regimes

Batch size	Name	Behavior
1	Stochastic (SGD)	Fast, noisy updates. Noise can help escape shallow local minima.
8–256	Mini-batch	The standard compromise, and far more hardware-efficient.
All	Full batch	Smoothest gradient, slowest progress, most memory.

The default here is full batch, since XOR has four examples.

Averaging over a batch reduces gradient noise — individual examples disagree about the best direction, and averaging finds their consensus.

The two-moons section of the demo is where this becomes visible (§22): 1000 examples at batch size 32 gives ~31 updates per epoch, so 150 epochs is 4650 updates. XOR's 4000 epochs are 4000 updates. **Epochs are not the unit that matters** — updates are, and the ratio between them is the batch size.

Note that mini-batching is currently a *statistical* device here, not a performance one. Real frameworks batch because it lets one loaded weight block serve many examples at once; this library still walks the weights once per example either way, which is why `ForwardBatch` measures no faster than a loop (§25 item 1).

Epochs

One epoch = one pass over all examples. Networks need many because each update is a small step (§4).

XOR takes ~4000 epochs here, which sounds enormous. But the demo runs **full batch**, so one epoch = one parameter update: 4,000 actual gradient-descent steps, each informed by all four examples (16,000 example evaluations). Four thousand small steps to tune 17 parameters is unremarkable — and it's why the distinction between an *epoch* and an *update* matters. At batch size 1 the same 4000 epochs would mean 16,000 updates.

---

21. Gradient checking — how you *know* the code is right

This is the section most tutorials omit, and it's the most practically valuable one.

The problem: a subtly wrong backward pass usually still trains *somewhat*. Loss goes down, nothing crashes, and results are merely mediocre — indistinguishable from "needs tuning." These bugs can cost days.

The defense: compare against a numerical estimate that makes no use of your backprop code (`GradientCheck.cs`). Nudge one weight up and down, measure how the loss actually moves, and compare:

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

This is the definition of a derivative with a finite ε instead of a limit. It's far too slow for training — two full forward passes *per parameter* — but perfect for verification. The **central** difference (both directions) has $O(\varepsilon^2)$ error versus $O(\varepsilon)$ for the one-sided version, which easily repays the second evaluation.

The measured result, and what it verifies

ε	max relative error	dominated by
1e-1	9.1e-3	truncation — ε too coarse to be a good derivative
1e-2	**2.4e-4**	balanced ← best
1e-3	1.7e-3	roundoff creeping in
1e-4	1.5e-2	float32 roundoff — `L(w+ε)` and `L(w−ε)` nearly identical

The U-shape is the evidence you want. Two error sources fight each other: large ε is a poor approximation of a derivative, while small ε subtracts two nearly-equal floats and loses precision catastrophically. A correct implemented

gradient shows this tradeoff with a sweet spot in the middle. **A wrong gradient shows O(1) error at every ε** — no sweet spot, because it isn't approximating anything.

`float` bottoms out around 1e-4; production checks use `double` and expect ~1e-10.

Rule: whenever you add a layer type or activation, gradient-check it before trusting it.

Using it

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

It works on any network because layers expose their parameters through a flat, type-agnostic index (`GetParameter` / `SetParameter` / `GetParameterGradient` on `ILayer`) rather than the check knowing about `Dense` specifically.

In the test suite

The checks run as real tests (`dotnet test`), including two that are easy to overlook:

- **A deliberately broken derivative** — a `Tanh` variant using `1 + a²` instead of `1 − a²` — must produce O(1) error. Without this, the passing tests might be passing *vacuously*: a check that cannot fail proves nothing.
- **Depth raises the error floor.** Three layers measure ~4.6e-3 against ~2.4e-4 for two, because each extra layer compounds float32 roundoff in the loss evaluations the finite difference depends on. The threshold is loosened for the deeper test — but it still separates a correct gradient from a broken one by more than an order of magnitude.

The ReLU exception — when a failing check is *not* a bug

The rule above says "gradient-check it before trusting it." There is one case where the check fails on correct code, and you need to know it before it happens to you, because the natural conclusion is that your backward pass is broken.

Finite differences assume the loss is smooth between `w−ε` and `w+ε`. ReLU isn't. It has a kink at `z = 0`. If nudging a weight by ε pushes some unit's `z` across zero, the two loss evaluations sit on opposite sides of the corner:

```
          L
          |        /          ε = 0.01, and z sits 0.001 above the kink.
          |      /
          |    /              L(w+ε) is on the sloped side.
  ────────┼──/                L(w−ε) is on the flat side.
          | /
  ────────└/                  Their difference measures the AVERAGE of two
       kink                    different slopes — not the derivative at w.
```

The analytic gradient is **right**. The *numerical estimate* is wrong. Measured on exactly that setup — one ReLU unit with `z` 0.001 from the kink:

ε	max relative error	what it means
1e-2 (default)	**2.9e-1**	straddles the corner — looks like a total failure
1e-4	8.8e-2	ε now smaller than the distance to the kink; recovering

Compare a genuinely broken derivative, which sits above 1e-1 at *every* ε and for *every* activation. That gives you two diagnostics for telling them apart:

1. **Shrink ε.** A kink artifact improves sharply; a real bug doesn't move.
2. **Swap the activation.** Rebuild the same network shape with tanh. The layer arithmetic under test is identical, and tanh has no corner to straddle — measured 4.1e-4 on the shape above. If tanh passes and ReLU doesn't, you found a kink, not a bug.

Both behaviours are pinned by tests (`ReLUKinkTests`), so this stays true. It's a known limitation of the technique rather than of this code, which is why production checks favour smooth activations even when the deployed network uses ReLU.

Part III — Practice

22. Results

> **Why your digits may differ.** Every number in this guide was produced on an **Apple M3 Pro, .NET 10, ARM64** (`Vector<float>.Count == 4`). Float addition is not associative — `(a+b)+c` and `a+(b+c)` can differ in the last bits — and the SIMD dot product sums in an order that depends on the vector width, which is 8 on AVX2 and 16 on AVX-512. So the same source, same seed, and same data can produce slightly different trailing digits on x86, and those differences compound over 4000 epochs.
>
> Expect the last digits to move. Expect the conclusions not to: no result here depends on a digit that isn't stable. CI runs on both ARM and x86 for exactly this reason. This is worth internalizing generally — **bit-for-bit reproducibility across machines is not something floating-point code gives you for free**, and chasing it is a common waste of time.

XOR-scale

```
Perceptron on AND: converged in 4 epochs      ← linearly 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      ← the hidden layer earning its keep

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

The XOR outputs aren't exactly 0 and 1 because sigmoid only *approaches* its limits — reaching 1.0 exactly would need infinite weights. 0.98 means "confidently 1."

Beyond XOR — the two moons

XOR proves the hidden layer defeats Minsky and Papert, and that is *all* it can prove. With four noise-free examples and no held-out data there is no way to demonstrate the three things that dominate practical training:

	why XOR can't show it
Mini-batching	Four examples is one full batch. Every "epoch" is a single update.
Generalization	The four points *are* the problem. There is nothing held out to generalize to.
Overfitting	Nothing to memorize.

So the demo's last two sections use `Datasets.Moons`: two interleaving crescents with Gaussian noise, 1500 points, split 1000 train / 500 test. Generated rather than downloaded — no data files, no network access, identical on every machine for a given seed.

A 2 → 16 → 16 → 1 network, 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%
```

Test accuracy tracking train accuracy is what generalization looks like. Note also that neither reaches 100% and that this is correct: at this noise level the crescents genuinely overlap, so some points are unclassifiable and a model scoring 100% would be a model that memorized them.

Note the epoch counts against XOR's 4000. There are ~31 updates per epoch here rather than one, which is the distinction from §20 made concrete: **4650 updates, not 150.**

The learned boundary, sampled across the plane:

```
##################################################·······
#################################################·········
################################################···········
####################·····##################·············
##################·········##############···············
################·············############·················
################···············#######··················
#############·····································
###########·····································
#########·····································
```

That curve is the entire argument for hidden layers, drawn to scale. A perceptron can only put a straight line on this picture — and the test suite asserts it does measurably worse.

Overfitting, deliberately

Same problem, same demo: 4417 parameters trained on **20** points — 221× more knobs than data.

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

Perfect on data it has seen; **93.4% on data it hasn't, against 96.4% for the smaller network given more data.** That gap is overfitting: capacity spent memorizing the noise in 20 points instead of learning the shape.

The important part is what it took to see it. Training loss fell the whole way; nothing threw; the model looked *better* by every number available at training time. **Only the held-out split made it visible**, and nothing in this library computes one for you — §25 item 6.

Handwritten digits — the real thing

`dotnet run -c Release --project src/NN.Mnist` trains on MNIST: 60,000 handwritten digits, 28×28 greyscale, the dataset every introduction eventually arrives at. The jump in scale from the moons is the point — 784 inputs instead of 2, and 101,770 parameters instead of 337.

Architecture: **784 → 128 tanh → 10 logits → softmax**, one output per digit, trained on one-hot targets with cross-entropy. The prediction is whichever probability is highest.

```
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% on digits it has never seen, in 37 seconds, from the code in this guide. Two seconds per epoch over 60,000 examples — the SIMD and layout work of §12 and §15 is finally operating at a size where it matters, which is also why the benchmarks measure a 784×128 layer.

Three things this demo teaches that nothing smaller can:

Accuracy hides structure; a confusion matrix shows it. The errors are not spread evenly:

```
          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%
```

Ones are nearly perfect; fives are the worst class, and twelve of them are called 3 — two digits that genuinely share a top stroke and a lower bowl. The network's mistakes are *structured*, and the demo prints the misclassified digits as ASCII so you can see that most are ones a person would also hesitate over.

The choice of loss is worth more than any amount of extra training. The same network scored by MSE over ten sigmoids reaches only 97.41%, and needs a learning rate of **1.0** to do it — far outside the 0.1–0.5 §4 recommends. Run `--loss mse` and watch. The cause is a gradient arriving already flattened, twice over:

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

Dividing by ten outputs, then multiplying by a factor that peaks at 0.25 and collapses toward zero as outputs saturate — that is, exactly when the network is confidently wrong and most needs to learn. The huge step size is compensation for a handicap the loss function imposed.

Softmax with cross-entropy removes it. Measured on identical architecture, seed and epochs:

Loss	Test accuracy	Learning rate
MSE over ten sigmoids	97.41%	1.0
MSE, at cross-entropy's learning rate	92.93%	0.1
Softmax + cross-entropy	**98.02%**	**0.1**

The middle row is the honest one: at the same learning rate the difference is five points, and MSE only closes the gap by taking steps ten times larger. §27 explains the mechanism.

The library's documented limits become measurable rather than theoretical. 98.0% is par for this architecture. What remains is §25 item 2 — plain SGD, no momentum or Adam — which is exercise 10, with a concrete number to beat.

And this is where saving a model finally means something

The demo writes the trained network to disk and reuses it:

```
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 seconds becomes 4 milliseconds, at identical accuracy. §19 explained the file format on a model with 17 parameters, where persistence is a curiosity. Here it is the difference between a demo you run once and a demo you can actually use — and it is the normal shape of deployed machine learning, where training happens rarely on expensive hardware and inference happens constantly somewhere else. The 397 KB is the entire deliverable; the 60,000 training images are not needed to classify anything.

Three details are worth pulling out.

The architecture comes back out of the file. Nothing in the loading path is told it is reading a 784-128-10 network — `ModelIO` stored each layer's type and shape next to its weights, so `Load` reconstructs the stack and `Summary()` prints it. That is why §19 argued against a bare weight dump.

The reload is verified, not assumed. After saving, the demo reloads and compares 1,000 predictions bit-for-bit. This matters more at 101,770 parameters than at 17: with seventeen, a mis-serialized weight almost certainly breaks a prediction visibly, while with a hundred thousand a single misplaced value shifts accuracy by a fraction of a percent and reads as noise. Exact equality is the only comparison that catches it, and `ModelScaleTests` pins the same property.

The filename carries what the format doesn't. The model file records the architecture but has no idea *how much data it saw*. A model trained on 5,000 images reloads perfectly happily alongside one trained on 60,000, so the demo puts both the hidden width and any training-set limit in the filename. Neither mismatch would be an error — both would be silently wrong results, which is the harder kind to notice.

> This is also §25 item 7 in practice: only parameters are saved, not optimizer state or training history. Fine here, because inference is all that's wanted. You could not resume an interrupted training run from this file.

Reading a digit out of an image file

A trained model on disk means you can point it at a picture. The repository ships one — a trained recognizer, `models/mnist-784-128-10.nnm` — so this works on a fresh clone with no training run and no dataset download:

```
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 and Netpbm are decoded in `ImageFile.cs` with no dependency beyond the framework. Most of that file is not decompression — `ZLibStream` does that — but **unfiltering**, PNG's own contribution: before compressing, each scanline stores the *difference* between every byte and a prediction from its neighbours (left, above, or Paeth's blend of both), which turns smooth images into runs of near-zero bytes that deflate flattens. Decoding is therefore strictly sequential — the "Up" filter refers to the already-reconstructed row above.

But the decoder is the easy half. The important half is `DigitPreprocessor`.

The network did not learn "digits" — it learned MNIST's conventions

This is the single most common reason a from-scratch recognizer scores 97% on the test set and then fails on the first photo you feed it, and it is worth internalizing well beyond this project.

MNIST images are not merely "pictures of digits." They are pictures under three specific rules:

Convention	Your image probably	Consequence if ignored
White ink on black	Dark pen on white paper	The network sees a bright frame with a dark hole — nothing like a digit
Digit fills a 20×20 box	Small, with margin around it	Downscaling the whole frame leaves a smudge a few pixels wide
Centred by centre of mass in 28×28	Wherever it happened to be	Every stroke lands where the network learned to see background

Violate any one and accuracy collapses in a way that looks exactly like a broken model. So the preprocessor:

1. **Detects polarity from the border**, not the whole image — the frame's edge is background almost by definition, whereas a thick digit can drag the overall mean darker than you'd guess.
2. **Crops to the ink's bounding box**, then scales to fit a 20×20 box *preserving aspect ratio*. Stretching to fill the square would give a `1` an `8`'s worth of ink in the wrong places.
3. **Resamples by box filter**, averaging every source pixel that falls into each destination pixel. Nearest-neighbour is the obvious choice and is wrong: downscaling 248 pixels to 20 by sampling one in twelve drops most of the stroke and yields a dotted, broken digit. Averaging preserves it as the soft grey edges MNIST itself has.
4. **Centres by centre of mass**, not bounding box. MNIST did this, and the difference is real — a 7 with a long descending stroke carries its mass high and its box centre low.

> **The lesson generalizes.** Those hundred lines are worth as much as the 101,770 trained parameters, because the parameters are meaningless applied to input in the wrong shape. "Most of machine learning is data preparation" is usually said about *training* data; it is just as true of the data you hand a finished model. The model is a function, and a function applied outside its domain returns confident nonsense rather than an error.

Verifying it end to end

The pipeline was checked by exporting MNIST test digits as **248×248 PNGs, dark-on-light with wide margins** — deliberately violating all three conventions — and reading them back:

Ten out of ten agreed with what the model predicts on the raw MNIST data, including one it gets *wrong*: a 5 it calls a 6 — at 0.898 confidence through the image pipeline, and 0.898 on the same digit read straight from the dataset. That last one is the useful result. The pipeline reproduces the model's mistakes as faithfully as its successes, and to three decimal places, which is how you know the preprocessing is transparent rather than accidentally helping. A preprocessing step that "fixed" that 5 would be evidence of a bug, not of quality.

If a prediction comes back with low confidence or a close runner-up, the demo says so and points at the 28×28 rendering it printed. **Look at that picture first.** A digit that appears inverted, tiny, or off-centre is a preprocessing problem, and no amount of additional training will fix it.

> The dataset is not in the repository. The demo downloads it once (~11 MB) and caches it outside the working tree; later runs, including offline ones, read the cache. With neither network nor cache it says so and exits cleanly — a teaching repo should not fail because a mirror is down. The IDX format it parses is worth a look (`Idx.cs`): a magic number, some dimensions, and raw bytes — **big-endian**, which is the one detail that silently ruins everything if you miss it.

---

23. Debugging playbook

The failure modes you'll actually hit, and what they mean:

Symptom	Likely cause	Fix
Loss → `NaN` or ∞	Learning rate too high; weights exploding	Divide learning rate by 10
Loss flat at a middling value	Zero/constant init — symmetry unbroken (§17)	Random initialization
Loss decreases then plateaus high	Not enough capacity, or saturated units	More hidden units; try tanh over sigmoid
Loss barely moves	Learning rate too low, or vanishing gradients	Raise learning rate; use ReLU/tanh
Trains but predicts poorly	Gradient bug, or genuinely too few epochs	**Gradient check first** (§21)
ReLU net stops improving	Dead units — output 0 for all inputs, so gradient is permanently 0	Lower learning rate; try leaky ReLU
Perfect on training data, bad on new data	Overfitting — memorization, not learning	More data, fewer parameters, regularization

Always gradient-check before tuning hyperparameters. Tuning a buggy gradient is an unbounded waste of time.

---

24. Exercises

Worked roughly in order of value. The "break it" ones teach the most.

1. **Watch XOR fail.** Point `Perceptron` at the XOR data instead of AND. It won't converge — that's the Minsky–Papert wall from §9, and feeling it beats reading about it.
2. **Break initialization.** Add `Array.Clear(Weights); return;` at the top of `Dense.Initialize` so weights stay zero. XOR freezes at loss **exactly 0.250000**, predicting 0.5000 for every input, forever. That's §17's zero-weight deadlock — the most convincing demonstration here of why initialization matters. Then try initializing every weight to the same *nonzero* value (say 0.5) and watch the different, milder failure: the layer learns, but as though it had a single unit.
3. **Break the gradient.** Change `Tanh.DerivativeFromOutput` to `1 + a * a`. Watch the gradient check jump to O(1) while training *still partly works*. This is why §21 exists.
4. **Sweep the learning rate.** Try 0.01, 0.1, 0.5, 2.0, 10.0 on XOR. You'll see slow crawling, healthy descent, and divergence to `NaN` — the whole spectrum from §4.
5. **Shrink the hidden layer** to 1 unit. XOR becomes unsolvable again; 2 is the theoretical minimum. Find where it starts working reliably.
6. **Try ReLU** in the hidden layer. It may need a lower learning rate. Print each hidden unit's output across all four inputs to spot dead units.
7. **Reproduce and then perturb the hidden-feature table in §3.** Cast `net.Layers[0]` to `Dense<Tanh>`, call `Forward` on each of the four inputs, and print the results — on an ARM machine you should get the numbers in §3, and on x86 the last digits may differ (§22). Now change the network's seed (`new Network(seed: 7, …)`) and print again. You'll get a completely different, equally valid decomposition. This is the most illuminating exercise here: it shows there's no single "correct" set of learned features.
8. **Sweep the batch size on the moons.** Try 1, 8, 32, 256, and full batch at a fixed epoch count, and plot test accuracy against *updates* rather than

epochs. §20's three regimes, on a dataset large enough for them to be distinguishable.

9. **Find where overfitting starts.** The demo uses 20 training points. Sweep 20, 50, 200, 1000 at fixed capacity and watch the train/test gap close. Then hold the data at 20 and shrink the network instead. Two different cures for the same disease.
10. **Add momentum:** keep a velocity buffer per layer, `v = βv + grad` (β ≈ 0.9), and step along `v`. Measure it on MNIST, where the baseline is **98.02% in 37 s** (§22) — a number concrete enough to beat or fail to beat. This is the biggest improvement still unimplemented.
11. **Prove the loss matters, then explain it.** Run MNIST both ways — `--loss mse --retrain` against the default — and confirm the table in §22 on your own machine. Then answer the question it raises: MSE reaches 97.41% *only* at a learning rate of 1.0, and manages 92.93% at cross-entropy's 0.1. Which of the two shrinking factors in §27 accounts for more of that gap? (Try MSE over a `Dense<Identity>` output to remove `σ'(z)` while keeping MSE, and see where it lands.)
12. **Derive the fused gradient yourself.** §27 states that softmax's Jacobian and cross-entropy's `1/p` cancel to leave `p − y`. Do the algebra for the two-class case and watch it happen, then deliberately break `SoftmaxCrossEntropy.Gradient` — use `p − y` scaled by 2, say — and run `SoftmaxGradientTests`. A wrong-but-plausible gradient still trains; the check still catches it. This is §21's argument applied to the one piece of algebra in the codebase.
13. **Look at what MNIST gets wrong.** The demo prints its own misclassified digits. Are they genuinely ambiguous, or is the network failing on something a person would find easy? Then read the confusion matrix: which pairs does it mix up, and do those digits share strokes? This is the habit that separates "98%" from knowing what a model actually does.
14. **Break the forward cache on purpose.** Make `Forward` cache again (§14) and call `net.Predict(...)` between `AccumulateGradients` and `ApplyGradients`. Nothing throws, loss still falls, and the network trains on the wrong example. Then run `CacheLifetimeTests` and watch them catch it. The best available demonstration of why "it still trains" proves nothing.
15. **Implement `ForwardBatch` as a real tiled GEMM** (§25 item 1) and benchmark it against the existing null result in `bench/`. This is the largest measured win still on the table.

16. **Re-run the benchmarks on your own machine.** If it's x86, `Vector<float>` is 8 or 16 wide rather than 4. Which conclusions in `bench/README.md` change, and which hold? The ones that hold are the ones worth trusting.

---

25. What this implementation does *not* do

Honest limits, ordered by how much they cost:

1. **No GEMM batching.** `ForwardBatch` just loops the single-example path, re-streaming the whole weight matrix per example. Single-example inference is **memory-bandwidth bound** — about one multiply-add per float loaded. Batching into a *tiled matrix-matrix multiply* reuses each loaded weight block across many examples, raising arithmetic intensity by roughly the batch size. Tuned BLAS libraries commonly report order-of-magnitude gains from this.

   Half of this is now measured. `ForwardBatch` benchmarks at **0.98× a manual loop** at batch sizes 1, 32 and 256 — a null result, confirming it buys nothing today (table). The 2% is loop overhead the caller no longer pays, not arithmetic. The *other* half — that a real tiled GEMM would dominate every other optimization here — remains unmeasured, because it remains unwritten. That is exercise 14, and it is still the largest single win available.
2. **Plain SGD.** No momentum, Adam, learning-rate schedule, or weight decay. Adam typically converges several times faster — measurable now against MNIST's 98.02% in 37 s (§22). With cross-entropy implemented, this is the largest remaining gap.
3. **Two losses only** — mean squared error and softmax cross-entropy (§27). Enough for regression and single-label classification; multi-label classification wants per-output binary cross-entropy, which does not exist here.
4. **Single-threaded.** No `Parallel.For` over units or batch rows.
5. **No explicit FMA.** `acc += a * b` may or may not fuse into one instruction; `Vector256.FusedMultiplyAdd` guarantees it, at the cost of writing a separate ARM path.
6. **No regularization or early stopping, and no automatic validation split.** `Train` will happily overfit and report a falling loss the whole way — §22 shows it doing exactly that. The demo splits train/test *by hand*, which is the minimum viable version; nothing in the library computes a validation score, watches it, or stops when it turns.

7. **Serialization saves parameters only** — not optimizer state (there is none yet) or training history. Fine for inference; you cannot resume training mid-run from a file.
8. **Single-example forward and backward.** Both walk one example at a time, so the API cannot express a batched backward pass even if item 1 were implemented. `ForwardBatch` is inference- only for this reason.

For production, the fastest C# is C# that calls something else: `TensorPrimitives`, ONNX Runtime, or a GPU. Nobody beats a tuned BLAS with hand-written loops. This code exists for understanding, and at this scale it will never be your bottleneck.

---

26. Where to go next

Roughly in order:

1. **Momentum, then Adam** — now the largest single improvement still available, since cross-entropy is implemented (§27). Baseline to beat: 98.02% in 37 s.
2. **A real dataset — already here.** MNIST is the classic first one, and `src/NN.Mnist` trains a 784 → 128 → 10 network on it to 98.0% in 37 seconds. The next step up is Fashion-MNIST (same shape and loader, harder problem) or CIFAR-10 (colour, and genuinely wants convolution).
3. **Regularization** — dropout, weight decay — once you can overfit something.
4. **Convolutional layers**, if images interest you.
5. **A real framework.** Having built this, PyTorch's `loss.backward()` / `optimizer.step()` will read as familiar machinery rather than magic — which is exactly the payoff for writing it yourself once.

---

27. Softmax and cross-entropy

MSE is the right loss for *regression* — predicting numbers. For *classification* — choosing one of several mutually exclusive categories — it is the wrong tool, and §22's MNIST run measures the cost: 97.41% and a learning rate of 1.0, against 98.02% at 0.1.

This section explains why, because the mechanism is one of the most useful pieces of calculus in practical machine learning, and it is short.

What's wrong with ten sigmoids

The MSE version gives each digit its own sigmoid output. Each one independently answers "is this a 7?" — and nothing stops all ten answering "yes, 0.9". That is incoherent when the image is exactly one digit, and it means the network spends capacity learning a constraint the architecture should have enforced for free.

Softmax: making the outputs compete

Softmax takes the last layer's raw scores — **logits**, unbounded and uninterpretable on their own — and turns them into a probability distribution:

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

Every output is positive, and they sum to exactly 1. Raising one *necessarily* lowers the others, which is the constraint ten independent sigmoids lack.

Two properties matter for the implementation:

It is shift-invariant. Adding a constant to every logit changes nothing — the constant factors out of every numerator and out of the denominator, and cancels. This is not a curiosity; it is the only reason the code works, because:

The naive formula overflows. `exp(z)` is infinity for z above about 88 in float32, and logits that large are ordinary in a trained network — giving `inf / inf = NaN`. Subtracting the largest logit first is exact (by shift-invariance) and makes the biggest exponent `exp(0) = 1`, so nothing can overflow. `SoftmaxCrossEntropy.Transform` does this, and the test suite checks it at logits of 1000.

Softmax cannot be an `IActivation`. Every activation in §13 maps one number to one number. Softmax needs the whole layer's outputs at once, because of the denominator. That is why it lives on the loss rather than the layer — see the note on `ILoss.Transform`.

Cross-entropy: scoring the distribution

Given a probability distribution, cross-entropy asks one question: *what probability did you assign to the right answer?*

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

Confidently right costs nothing. Confidently wrong costs **unboundedly** much — `-log(0.001)` is 6.9 and climbing, where MSE caps the penalty at 1 per output no matter how badly wrong you are. That difference in shape is what makes the gradient behave.

The fusion, which is the whole point

Differentiate the two separately and both are unpleasant:

- **Softmax alone** gives a full Jacobian — every output depends on every logit, so you get an n×n matrix per example rather than a vector.
- **Cross-entropy alone** gives a `1/p` term that explodes as p approaches 0 — exactly where a badly wrong network lives.

Compose them and almost everything cancels. What survives is:

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

Prediction minus target. No Jacobian, no division, nothing to overflow, and — critically — no `σ'(z)` factor to vanish. Compare the MSE-over-sigmoid chain from §22, whose gradient is scaled by `a(1−a) ≤ 0.25`, collapsing toward zero precisely when the network is most wrong. Cross-entropy's `1/p` blowing up and softmax's Jacobian shrinking cancel each other **exactly**.

That is the entire reason classifiers are built this way, and it is why the MNIST demo trains at a learning rate of 0.1 instead of 1.0.

Two things the code does about it

It requires a linear output layer. `p − y` is the derivative with respect to the *logits*. If the last layer squashed them through a sigmoid first, the formula would simply be false — and false in the worst way, since the network would still train, just badly. So `SoftmaxCrossEntropy.Validate` rejects any output layer that is not `Dense<Identity>`, once, at construction:

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

`SoftmaxOutput` exists precisely so the pairing cannot be got wrong by accident.

For the cases the shortcut doesn't cover there is `WithLoss`, which sets the loss explicitly and leaves the output layer to you — that is how the test suite installs deliberately broken losses to prove the gradient check can fail. A built network exposes its choice as `Network.LossFunction`, which is also what `ModelIO` writes to disk:

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

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

It is gradient-checked. A cancellation this convenient is exactly the kind of algebra that is easy to get *almost* right, and §21's argument applies with full force: an almost-right gradient still trains. `SoftmaxGradientTests` runs the same finite-difference check, including the accuracy U-curve, against the fused formula. That test is the difference between believing the derivation and knowing it.

The loss travels with the model

A softmax classifier's weights are meaningless without the knowledge that softmax applies to them: load one as a plain network and it returns unbounded logits where the caller expects probabilities. Nothing throws — the numbers are just wrong. So the loss is written into the model file, which is what took the format to **version 2**. Version 1 files still load, as MSE, which is what they were (§19).

Glossary

Term	Meaning
Unit / neuron	One output of a layer: weighted sum + bias, then an activation
Weight	How strongly one input influences one unit
Bias	A unit's baseline output before inputs are considered
z (pre-activation / logit)	The weighted sum, before the activation is applied
a (activation)	`g(z)` — the unit's output
δ (delta)	`dL/dz` — the gradient after passing back through the activation
Gradient	Vector of derivatives: which way is uphill, and how steeply
Loss	One number measuring how wrong the predictions are
Forward pass	Input → prediction
Backward pass	Loss → gradients for every parameter
Epoch	One full pass over the training set
Mini-batch	A group of examples whose gradients are averaged into one update
Learning rate (η)	Step size for gradient descent
Fan-in / fan-out	Number of inputs to / outputs from a layer; sets the init scale
Symmetry breaking	Random init ensuring units in a layer can learn different features
Linearly separable	Separable by one straight line/plane (AND yes, XOR no)
Saturation	An activation pinned at its flat extreme, where the gradient ≈ 0
Vanishing gradient	Gradients shrinking toward zero through depth, stalling early layers
Dead unit	A ReLU unit outputting 0 for every input — gradient permanently 0
Overfitting	Memorizing training data instead of learning generalizable patterns
Hyperparameter	A value you choose rather than learn (learning rate, layer sizes, epochs)
SIMD	One CPU instruction operating on 4–16 numbers simultaneously
GEMM	General Matrix Multiply — the batched operation real frameworks are built on