Chapter 6

Solving XOR — Hidden Layers and Non-linearity

6.1 What You Already Know

You already know **function composition**: $f(g(x))$, applying one function to the output of another. And if you have ever touched digital logic, you know the basic **logic gates** — AND, OR, and NAND (NOT-AND) — and that any Boolean function whatsoever can be built by wiring enough of them together:

A	B	A AND B	A OR B	A NAND B
0	0	0	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Chapter 5 ended on a cliffhanger: no single perceptron, however its weights are chosen, can compute XOR. This chapter resolves that cliffhanger using nothing more exotic than these two familiar ideas — composition and logic gates — wired together in a very specific way.

6.2 The Solution to XOR

6.2.1 XOR as a Composition of Simpler Gates

The identity that unlocks everything is a piece of Boolean algebra:

XOR Decomposition

$$A \oplus B \;=\; (A \text{ OR } B) \text{ AND NOT}(A \text{ AND } B) \;=\; (A \text{ OR } B) \text{ AND } (A \text{ NAND } B)$$

Verify it directly: XOR is true exactly when *at least one* input is true (the OR condition) *and* they are *not both* true (the NAND condition). Each of OR and NAND, individually, is linearly separable — each was solved by a single perceptron back in Chapter 5's framework. XOR itself is not linearly separable, but it is a simple *logical combination* of two things that are.

6.2.2 Two-Layer Implementation

Translate the decomposition directly into a network of perceptrons, using the step-function convention of Chapter 5 (threshold at $z \geq 0$, output $\{0, 1\}$ for this construction):

Hand-Engineered XOR Network

Hidden neuron 1 (computes OR): $w = (1, 1)$, $b = -0.5$ **Hidden neuron 2 (computes NAND):** $w = (-1, -1)$, $b = 1.5$ **Output neuron (computes AND of the two hidden outputs):** $w = (1, 1)$, $b = -1.5$

Verifying the Hand-Engineered Network Solves XOR

Trace all four inputs through both layers:

x_1	x_2	h_1=OR	h_2=NAND	output	target
0	0	0	1	step(0+1−1.5)=0	0
0	1	1	1	step(1+1−1.5)=1	1
1	0	1	1	step(1+1−1.5)=1	1
1	1	1	0	step(1+0−1.5)=0	0

Every row matches. Three perceptrons, none individually capable of solving XOR, solve it exactly once composed in two layers. No training was needed for this particular network — the weights were derived directly from the Boolean identity above — but Section 6.5 shows the same architecture arriving at an equivalent solution purely by gradient descent, with no hand-engineering at all.

6.2.3 Visualization: Transforming the Input Space

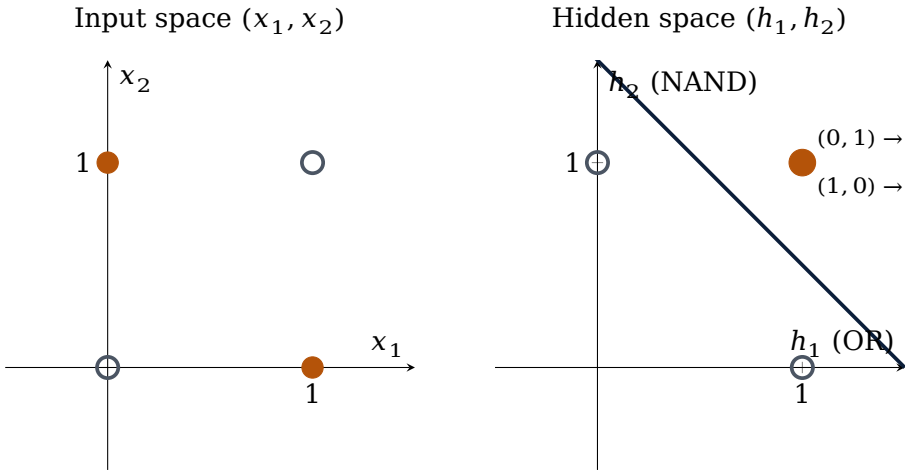

Figure 6.1: The hidden layer as a space transformation. **Left:** the original input space, where the two "alarm" points (filled) and two "no-alarm" points (open) interleave at the corners of a square — not linearly separable. **Right:** the same four points, passed through the hidden layer $(h_1, h_2) = (\text{OR}, \text{NAND})$. The two inputs $(0, 1)$ and $(1, 0)$, both XOR-true, land on the exact same point $(1, 1)$ in hidden space; the two XOR-false inputs land at $(0, 1)$ and $(1, 0)$ in hidden space. In this transformed space, a single straight line ($h_1 + h_2 = 1.5$, exactly the output neuron's decision boundary) separates them trivially. This is the general mechanism by which a hidden layer solves non-linear problems: it does not bend the decision boundary in the original space so much as *re-coordinatise* the data into a new space where a straight boundary suffices.

6.3 Non-linear Activation Functions

The hand-engineered network used a hard step function, exactly like Chapter 5's perceptron, which is instructive but, as Chapter 5 noted, untrainable by gradient descent (its derivative is zero almost everywhere). Practical networks use smooth, differentiable alternatives.

Three Standard Activation Functions

Sigmoid (met already in Chapter 4):

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad \text{range } (0, 1)$$

Tanh (hyperbolic tangent):

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = 2\sigma(2z) - 1, \qquad \text{range } (-1, 1)$$

ReLU (Rectified Linear Unit):

$$\text{ReLU}(z) = \max(0, z), \qquad \text{range } [0, \infty)$$

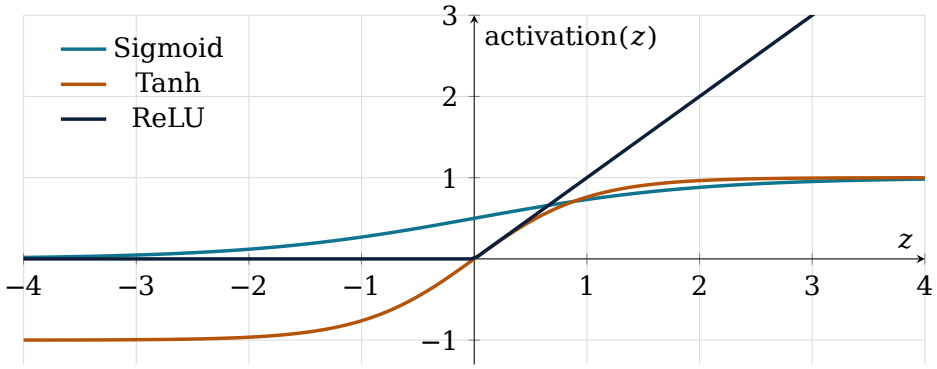

Figure 6.2: The three activation functions compared. Sigmoid and tanh are both S-shaped and saturate at both ends (their derivative vanishes far from zero, the source of the "vanishing gradient" problem examined in Chapter 7); tanh is simply a rescaled, zero-centred sigmoid. ReLU does not saturate for positive z at all, is trivial to differentiate (1 for $z > 0$, 0 for $z < 0$), and is the default choice in most modern deep networks precisely because of these two properties.

6.3.1 Why Non-linearity Is Essential

The reason a hidden layer needs a *non-linear* activation, and not merely a second linear layer, is a two-line algebraic fact. Suppose two layers both used the identity activation:

$$\mathbf{h} = W_1\mathbf{x} + \mathbf{b}_1, \qquad \hat{\mathbf{y}} = W_2\mathbf{h} + \mathbf{b}_2 = W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \underbrace{(W_2W_1)}_{W'}\mathbf{x} + \underbrace{(W_2\mathbf{b}_1 + \mathbf{b}_2)}_{b'}.$$

The composition of two linear functions is itself just one linear function, $W'\mathbf{x} + b'$ — exactly the single-layer model of Chapter 4, with no additional representational power whatsoever. This holds for any number of stacked linear layers: a hundred linear layers compute nothing a single linear layer could not compute exactly. Depth without non-linearity is not depth at all; it is expensive redundancy. Every non-trivial capability of a "deep" network, starting with its ability to solve XOR at all, depends entirely on the non-linear activation between layers.

6.4 The Universal Approximation Theorem

6.4.1 Statement

Universal Approximation Theorem — Cybenko 1989, Hornik 1991

A feedforward network with a single hidden layer, a finite number of neurons, and a suitable non-linear activation (sigmoid, tanh, and most other common choices all qualify) can approximate any continuous function on a compact subset of $\mathbb{R}^d$ to arbitrary accuracy, given enough hidden neurons.

6.4.2 Proof Sketch, via Step Functions

The intuition is a staircase approximation, the same idea used to build the Riemann integral. Two sigmoid units, with large opposing weights and offset biases, can be

combined to produce a narrow ”bump” function that is approximately zero everywhere except a chosen narrow interval, where it rises to approximately one. Summing many such bumps, each scaled to match the target function’s height over its own narrow interval, produces a piecewise-constant approximation of any continuous function — exactly a Riemann sum, built out of neurons instead of rectangles. As the number of hidden neurons grows, the intervals can be made arbitrarily narrow, and the approximation error shrinks toward zero, for any continuous function on a bounded domain.

Honest Boundary

The Universal Approximation Theorem guarantees **existence**: a network with enough hidden neurons *can* represent the target function to any desired accuracy. It says nothing at all about **how many** neurons are needed for a given function and a given accuracy (the number can be, and often is, astronomically large for anything but toy problems), and it says nothing about whether gradient descent (Chapter 4, Chapter 7) will actually *find* the right weights. This is the same existence-versus-construction gap already flagged for the Shannon coding theorem (Chapter 1) and the perceptron’s representational limits (Chapter 5): a proof that a solution exists is valuable and often surprising, but it is not, and was never meant to be, a training algorithm.

6.5 Worked Example: XOR in C#

The hand-engineered network of Section 6.2 solved XOR by direct construction, using Boolean algebra. Here, the same 2-2-1 architecture (two inputs, two hidden neurons, one output neuron) is trained **purely by gradient descent** on sigmoid activations, with no hand-derived weights at all, using nothing more than the chain-rule derivatives that a full, systematic treatment arrives at in Chapter 7.

XorNetwork.cs — training a 2-2-1 network on XOR

```
using System;

class XorNetwork
{
    static double Sigmoid(double z) => 1.0 / (1.0 + Math.Exp(-z));

    static void Main()
    {
        double[,] x = { {0,0}, {0,1}, {1,0}, {1,1} };
        double[] y = { 0, 1, 1, 0 };

        // Fixed, explicit initial weights -- reproducible, no RNG.
        double[,] w1 = { {0.5, -0.6}, {-0.7, 0.4} };
        double[] b1 = { 0.1, -0.2 };
        double[] w2 = { 0.8, -0.9 };
        double b2 = 0.1;

        const double eta = 2.0;
        const int epochs = 30000;

        for (int epoch = 0; epoch < epochs; epoch++)
        {
```

```
            var h = new double[4, 2];
            var o = new double[4];
            var deltaO = new double[4];
            var deltaH = new double[4, 2];

            for (int i = 0; i < 4; i++)
            {
                for (int j = 0; j < 2; j++)
                {
                    double pre = x[i, 0] * w1[0, j] + x[i, 1] * w1[1, j
                        ] + b1[j];
                    h[i, j] = Sigmoid(pre);
                }
                double oPre = h[i, 0] * w2[0] + h[i, 1] * w2[1] + b2;
                o[i] = Sigmoid(oPre);
                deltaO[i] = (o[i] - y[i]) * o[i] * (1 - o[i]);
            }

            double gradW2A = 0, gradW2B = 0, gradB2 = 0;
            var gradW1 = new double[2, 2];
            var gradB1 = new double[2];

            for (int i = 0; i < 4; i++)
            {
                gradW2A += h[i, 0] * deltaO[i];
                gradW2B += h[i, 1] * deltaO[i];
                gradB2 += deltaO[i];

                for (int j = 0; j < 2; j++)
                {
                    deltaH[i, j] = deltaO[i] * w2[j] * h[i, j] * (1 - h
                        [i, j]);
                    gradW1[0, j] += x[i, 0] * deltaH[i, j];
                    gradW1[1, j] += x[i, 1] * deltaH[i, j];
                    gradB1[j] += deltaH[i, j];
                }
            }

            w2[0] -= eta * gradW2A / 4; w2[1] -= eta * gradW2B / 4; b2
                -= eta * gradB2 / 4;
            for (int j = 0; j < 2; j++)
            {
                w1[0, j] -= eta * gradW1[0, j] / 4;
                w1[1, j] -= eta * gradW1[1, j] / 4;
                b1[j] -= eta * gradB1[j] / 4;
            }
        }

        // Final forward pass for reporting omitted here;
        // see the results table below.
    }
}
```

Training Results

Loss (mean squared error over all four examples) against training epoch, and the final predictions:

Epoch	MSE Loss
0	0.2530
5,000	0.00071
10,000	0.00030
20,000	0.00013
30,000 (final)	0.0000863

x_1	x_2	Predicted	Target
0	0	0.009	0
0	1	0.989	1
1	0	0.991	1
1	1	0.008	0

Every prediction lands within 1.2% of its target, with no XOR identity, no Boolean algebra, and no hand-set weight anywhere in the process — only the generic gradient descent machinery of Chapter 4, applied to a slightly deeper composition of functions. The final trained weights (W_1 approximately $\begin{pmatrix} 6.40 & 6.41 \\ -6.52 & -6.20 \end{pmatrix}$, $\mathbf{b}_1 \approx (-3.45, 3.11)$, $\mathbf{w}_2 \approx (10.70, -10.43)$, $b_2 \approx 4.98$) bear no resemblance to the clean ± 1, ± 0.5 weights of Section 6.2's hand-engineered solution, yet compute an almost identical function. This is a first, concrete instance of a pattern that recurs throughout the rest of this book: gradient descent routinely finds *a* solution, but rarely finds the same, humanly-readable solution a domain expert would have written by hand — a central motivation, later, for the Deterministic Islands of Chapter 12.

6.6 From XOR to Deep Networks

6.6.1 Hierarchical Feature Learning

XOR needed exactly one hidden layer to become solvable. Real industrial perception and diagnosis problems — interpreting a vibration spectrum, a camera image of a component, or a multivariate sensor trace — are vastly more structured than XOR, and benefit from **multiple** hidden layers stacked in sequence, each building on the representation the previous layer constructed. This is **hierarchical feature learning**: rather than hand-engineering features (as a domain expert might design a vibration-envelope statistic), a deep network learns, layer by layer, its own internal features, each one a re-coordinatisation of the last, in exactly the sense Figure 6.1 made concrete for a single hidden layer.

6.6.2 Each Layer, a More Abstract Representation

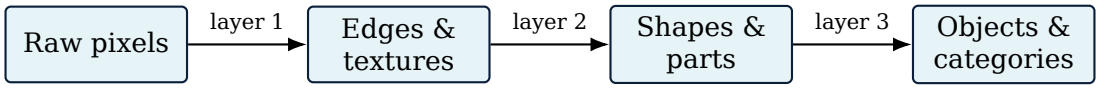

Figure 6.3: Hierarchical feature learning, schematically, for an image classifier. Early layers learn low-level, local features (edges, textures) that are useful for almost any visual task; deeper layers combine these into increasingly abstract, task-specific concepts (shapes, parts, whole objects). No layer is told what to represent; each learns whatever composition of the previous layer's outputs best reduces the training loss. Chapter 8 formalises the specific architecture (convolutional networks) that makes this hierarchy efficient for spatial data.

6.6.3 Examples: From Pixels to Objects

This hierarchy is not a metaphor; when the internal representations of a trained image classifier are inspected directly, early layers are consistently found to have learned detectors resembling edges, colour blobs, and simple textures — features general enough to be useful for almost any visual task — while the deepest layers, just before the final classification, respond to highly specific, often human-recognisable concepts (a particular texture of corrosion, the silhouette of a specific component) that are useful only for the particular task the network was trained on. The same hierarchy appears, in different guises, in a vibration-based fault classifier (raw waveform → frequency-band energy → fault signatures → fault category) and in the ILM of Chapter 10 (tokens → local syntax → semantic content → task-specific decisions).

Honest Boundary

The Universal Approximation Theorem (Section 6.4) guarantees that a sufficiently wide single hidden layer can represent any continuous function. It does **not** guarantee **generalisation**: a network that perfectly represents, or even perfectly memorises, its training data may still fail badly on new data, for exactly the bias-variance reasons established in Chapter 3. Representational capacity and generalisation are different properties, established by different theorems, and possessing one is no guarantee of the other.

Deep networks are hard to train, in ways breadth alone does not suffer from: Chapter 7 examines vanishing and exploding gradients, the non-convexity of the loss landscape, and the practical optimisation difficulties that a purely existential theorem like Universal Approximation is silent about. "A network exists that solves your problem" and "gradient descent will find it in a practical amount of time and data" are two entirely different claims, and this book will never conflate them.