IN402

Machine Learning

Chapter

8

Neural Networks: Architecture and Forward Propagation

Author: Abbas El-Hajj Youssef

University: Lebanese University

Department: Department of Computer Science

These notes extend course materials taught by Prof. Ahmad Faour with additional content from textbooks and supplementary resources.

Disclaimer: *This is not an official course document.*

Contents

1 The Computational Challenge That Sparked a Revolution

Consider the challenge of reading handwritten zip codes on envelopes. Each digit—a mere squiggle of ink—contains patterns that humans recognize instantly but that defeated the best algorithms of the 1980s. A handwritten "7" might be crossed or uncrossed, slanted left or right, written with flourish or haste. Two people's sevens share few pixels in common, yet we recognize both without effort.

Figure 1: *The digit recognition challenge: different pixel patterns must map to the same classification.*

Consider the mathematics this demands. A modest 28×28 pixel grayscale image contains 784 intensity values, each ranging from 0 to 255. We seek a function $f : \mathbb{R}^{784} \to \{0, 1, \ldots, 9\}$ that correctly classifies images regardless of handwriting style, slant angle, or pen thickness. How do we construct such a function?

The Central Question

This chapter addresses a fundamental problem: **How can we construct mathematical functions capable of recognizing patterns that vary enormously in surface appearance but share deep structural similarity?**
The answer leads us to **neural networks**—computational architectures that learn to build complex functions from simple, reusable pieces.

2 The Neuron: Atomic Unit of Neural Computation

The **neuron** (also called a **unit** or **node**) is the fundamental building block of every neural network. Understanding neurons deeply is essential—everything else in this chapter follows from this foundation. We treat neural networks as purely mathematical objects: *compositions of parameterized functions optimized to minimize a loss.*

2.1 The Two-Stage Computation

A neuron receives multiple inputs and produces a single output through two distinct stages.

The Artificial Neuron

Given inputs $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$, weights $\mathbf{w} = [w_1, w_2, \ldots, w_n]^\top$, and bias b:

Stage 1: Linear Combination (Pre-activation)

$$z = \sum_{j=1}^{n} w_j x_j + b = \mathbf{w}^\top \mathbf{x} + b \tag{1}$$

Stage 2: Nonlinear Transformation (Activation)

$$a = g(z) \tag{2}$$

where $g : \mathbb{R} \to \mathbb{R}$ is called the **activation function**.

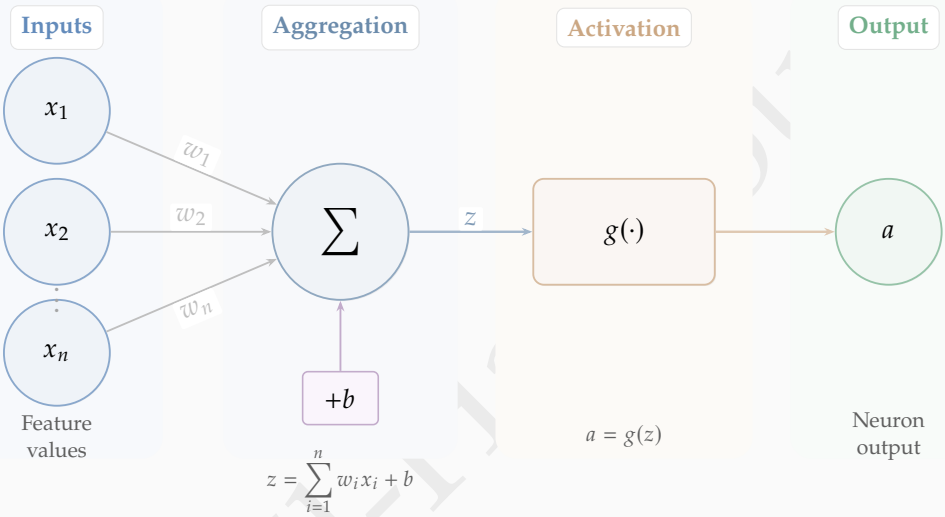

Figure 2: *Anatomy of an artificial neuron.* ***Inputs*** $x_1, \ldots, x_n$ *are multiplied by* ***weights*** $w_1, \ldots, w_n$ *and summed with* ***bias*** b *to produce the pre-activation* z*. The* ***activation function*** $g(\cdot)$ *applies a nonlinearity to yield the output* a*. Common choices for* g *include sigmoid, tanh, and ReLU.*

2.2 The Critical Connection: A Neuron *Is* Logistic Regression

If you understood logistic regression from earlier chapters, you already understand 90% of a neuron. Consider a single neuron with **sigmoid activation**:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}$$

This is exactly logistic regression. Both models compute a weighted sum $z = \mathbf{w}^\top \mathbf{x} + b$, apply the sigmoid function $\hat{y} = \sigma(z)$, and define a *linear* decision boundary where $\mathbf{w}^\top \mathbf{x} + b = 0$.
The neural network generalization extends this in three ways:

1. Employing various activation functions beyond sigmoid (ReLU, tanh, etc.)
2. Combining *multiple* neurons into layers
3. Stacking layers to enable *nonlinear* decision boundaries

This connection provides your conceptual anchor: you already understand the atomic unit. The power of neural networks emerges from *combining* these units systematically.

2.3 Geometric Meaning of Weights and Bias

The weights and bias carry precise geometric meaning—they control *how the activation function is positioned and scaled* within input space.
Consider a single-input neuron computing $a = \sigma(wx + b)$. The sigmoid transitions from near 0 to near 1 around the point where $wx + b = 0$, which occurs at $x = -b/w$.

- **The weight** w controls *steepness* and *direction*:
  - Large $|w|$: Sharp, decisive transition
  - Small $|w|$: Gradual, uncertain transition
  - Negative w: Curve flips—output *decreases* as input increases
- **The bias** b controls *position*:
  - The transition midpoint occurs at $x = -b/w$
  - Changing b shifts the entire curve left or right

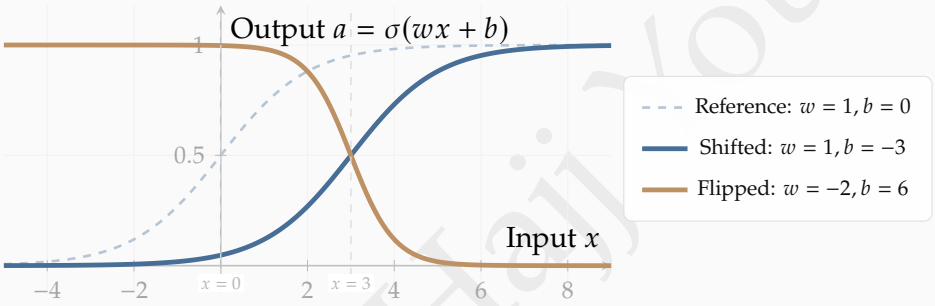

Figure 3: ***Geometric control via weights and bias.*** *The weight w determines steepness and direction; the bias b determines horizontal position. The transition center is located at $x = -b/w$.*

2.4 Common Activation Functions

The choice of activation function determines the "shape" of each computational building block. Three functions dominate modern practice.

2.4.1 Sigmoid: The Probability Gate

The **sigmoid** (logistic) function maps any real number to the interval $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

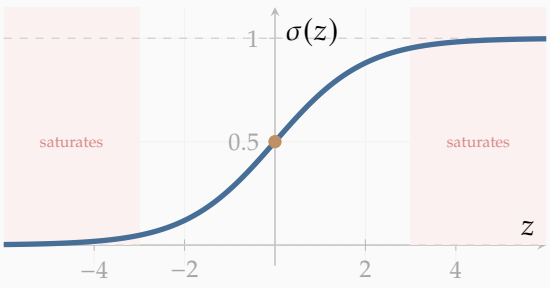

Figure 4: *The sigmoid function. Output range $(0, 1)$ makes it ideal for representing probabilities. Shaded regions indicate saturation zones where gradients become very small.*

Properties: Range $(0, 1)$—outputs interpretable as probabilities; smooth and differentiable everywhere; derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.
Primary use: Output layer for binary classification.
Limitation: Saturates for large $|z|$, causing vanishing gradients during training (discussed in the next chapter).

2.4.2 Tanh: The Zero-Centered Alternative

The **hyperbolic tangent** is a rescaled sigmoid:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{4}$$

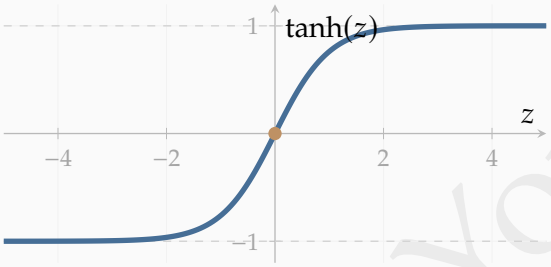

Figure 5: *The hyperbolic tangent. Range $(-1, 1)$ with zero-centered output—the function passes through the origin.*

Properties: Range $(-1, 1)$; *zero-centered* outputs (centered around 0); odd symmetry: $\tanh(-z) = -\tanh(z)$.
Limitation: Like sigmoid, tanh saturates at extreme values.

2.4.3 ReLU: The Modern Default

The **Rectified Linear Unit** revolutionized deep learning around 2012:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{5}$$

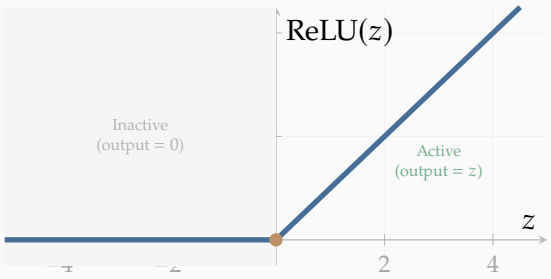

Figure 6: *The ReLU function. Piecewise linear and computationally trivial. Unlike sigmoid and tanh, it does not saturate for positive inputs.*

Properties: Range $[0, \infty)$; computationally trivial (just a comparison); no saturation for positive inputs.
Limitation: When $z < 0$, output is exactly zero. Neurons receiving consistently negative pre-activations become permanently inactive ("dying ReLU").

Leaky ReLU: A Simple Fix

Leaky ReLU modifies the inactive region to have a small slope instead of zero:

$$\text{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$$

where $\alpha \approx 0.01$ is a small constant. This ensures neurons remain responsive even for negative inputs.

3 The Limits of a Single Neuron: Why Networks Exist

What can a single neuron actually *accomplish*? This section reveals a fundamental limitation that makes neural *networks*—not just individual neurons—necessary.

3.1 The Geometry of a Single Neuron

Consider a neuron with sigmoid activation performing binary classification. Given input $\mathbf{x}$, it outputs:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

We classify as class 1 when $\hat{y} \geq 0.5$, which occurs when $\mathbf{w}^\top \mathbf{x} + b \geq 0$. The **decision boundary**—the set of points where the neuron is maximally uncertain—is defined by:

$$\mathbf{w}^\top \mathbf{x} + b = 0$$

This equation describes a **hyperplane**: a flat surface that divides input space into two half-spaces.

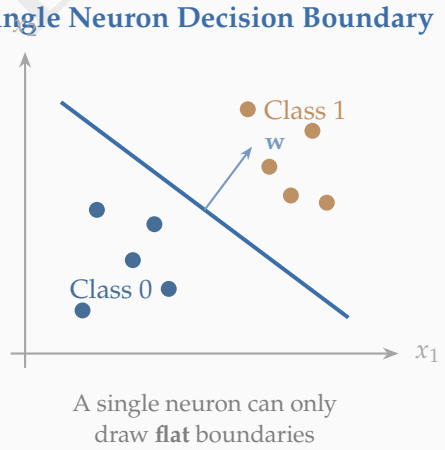

Figure 7: *A single neuron partitions input space with a hyperplane. In 2D, the boundary is a line; in higher dimensions, it remains flat. The weight vector* $\mathbf{w}$ *is perpendicular to the boundary.*

The critical geometric insight: **a single neuron can only draw flat decision boundaries**. This is intrinsic to the neuron's mathematical structure.

Linear Separability

A binary classification dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ is **linearly separable** if there exists a hyperplane that perfectly separates all class-0 points from all class-1 points. Equivalently, there exist $\mathbf{w}$ and b such that:

$$\mathbf{w}^\top \mathbf{x}^{(i)} + b > 0 \quad \text{for all } i \text{ where } y^{(i)} = 1$$
$$\mathbf{w}^\top \mathbf{x}^{(i)} + b < 0 \quad \text{for all } i \text{ where } y^{(i)} = 0$$

A single neuron can *only* solve linearly separable problems.

3.2 The XOR Problem: A Minimal Impossibility

The simplest demonstration of linear inseparability is the **exclusive-OR** (XOR) function. XOR returns 1 if *exactly one* input is 1:

x_1	x_2	$y = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: *The XOR truth table.*

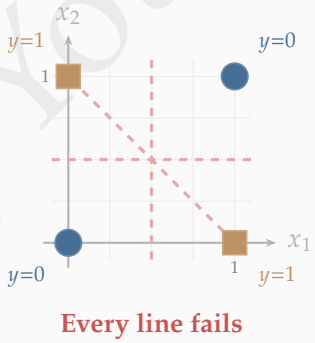

Figure 8: *XOR is not linearly separable.*

The geometry reveals why: class-0 points lie on the main diagonal while class-1 points lie on the anti-diagonal, making separation by any single line impossible.

The Deeper Pattern

XOR exposes when linear classification fails: **when the correct output depends on *interactions* between features rather than their individual values**.

In XOR, neither x_1 alone nor x_2 alone determines the output. Only their *relationship*—whether they agree or disagree—matters. A single neuron, computing only a weighted sum, cannot capture such interactions.

This pattern appears throughout machine learning: image classification requires interactions between pixels (edges, textures); text understanding requires interactions between words (context, grammar); medical diagnosis requires interactions between symptoms.

3.3 The Classical Escape: Feature Engineering

Before neural networks became practical, the standard approach to nonlinear problems was **feature engineering**: manually designing new input features that render the problem linearly separable.

For XOR, the key observation is that the output equals 1 when inputs *disagree*. We can capture this with a new feature $x_3 = x_1 \cdot x_2$, which equals 1 only when both inputs are 1:

x_1	x_2	$x_3 = x_1 x_2$	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 2: *XOR with an engineered feature. The product $x_3 = x_1 x_2$ captures the crucial interaction.*

In this augmented 3D space, the point $(1, 1)$ lifts to $(1, 1, 1)$—making the classes separable:

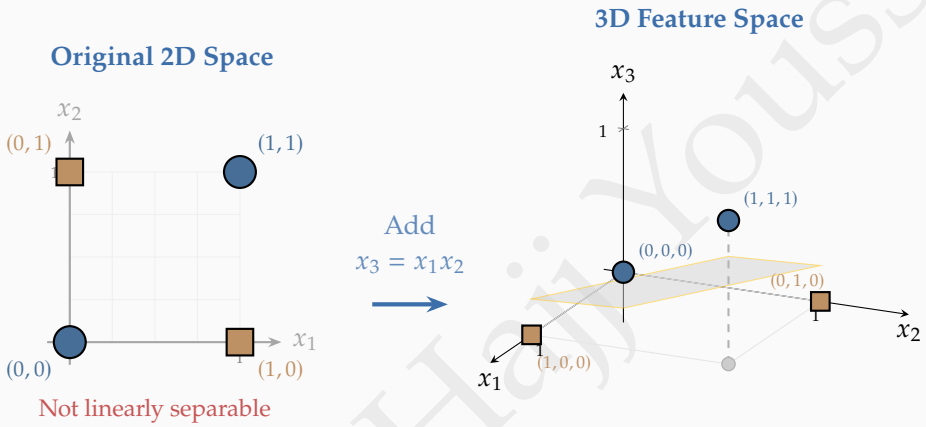

Figure 9: ***XOR becomes linearly separable via feature engineering. Left:*** *Original 2D problem.* ***Right:*** *Adding $x_3 = x_1 x_2$ lifts $(1, 1)$ to $(1, 1, 1)$. The plane $x_1 + x_2 - 2x_3 = 0.5$ (gray) separates the classes.*

The separating plane $x_1 + x_2 - 2x_3 = 0.5$ correctly classifies all four points:

$$\begin{aligned}
(0,0,0): &\quad 0 + 0 - 0 = 0 < 0.5 \implies \text{Class } 0 \quad \checkmark \\
(0,1,0): &\quad 0 + 1 - 0 = 1 > 0.5 \implies \text{Class } 1 \quad \checkmark \\
(1,0,0): &\quad 1 + 0 - 0 = 1 > 0.5 \implies \text{Class } 1 \quad \checkmark \\
(1,1,1): &\quad 1 + 1 - 2 = 0 < 0.5 \implies \text{Class } 0 \quad \checkmark
\end{aligned}$$

3.4 The Curse of Manual Features

Feature engineering solved XOR—but at a cost. We had to *know in advance* that the product feature would help. For two binary inputs, finding the right transformation is tractable. For real problems, it is not.

Consider applying this approach to image classification with $28 \times 28 = 784$ pixels:

- **Original features:** 784 pixel intensities
- **All pairwise products:** $\binom{784}{2} = 306{,}936$ features
- **All degree-3 products:** $\binom{784}{3} \approx 80$ million features

Even pairwise interactions would require handling over 300,000 features from perhaps 60,000

training examples—a recipe for poor generalization. Worse, polynomial features may be entirely wrong for images.

Feature engineering requires *knowing in advance* which transformations make the problem tractable. For complex domains like vision, language, and audio, human intuition fails. This leads to the central question: *Can we build models that automatically discover useful features from data?*

The answer is yes—by using multiple neurons working together, letting the network *learn* the feature transformations that feature engineering would require us to specify manually. The next section shows how.

4 From Neurons to Networks: The Source of Power

A single neuron is just logistic regression—limited to flat boundaries. But something remarkable happens when we connect multiple neurons together. This section develops the core insight that makes neural networks powerful.

4.1 The Key Insight: Building Complexity from Simplicity

Consider what happens when we add two sigmoid functions with different parameters:

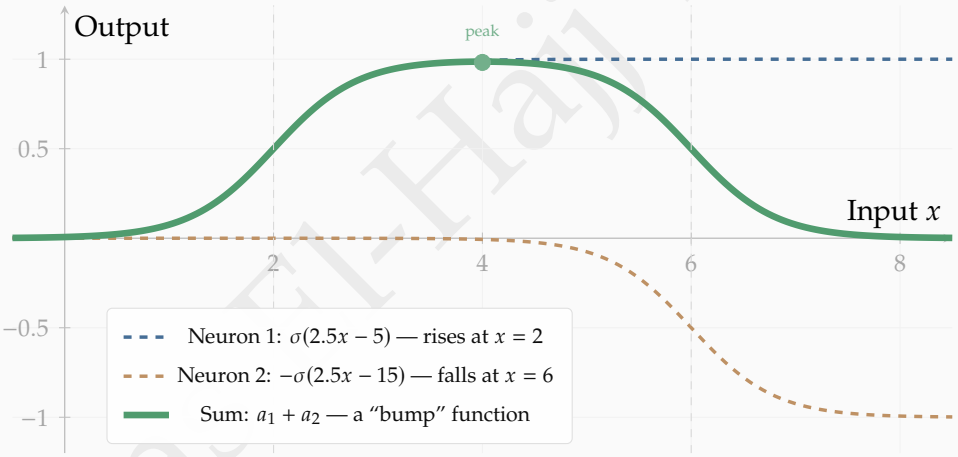

Figure 10: ***Constructing a bump from two neurons.*** *Neuron 1 (blue, dashed) rises around $x = 2$. Neuron 2 (orange, dashed) falls around $x = 6$ due to a negative output weight. Their sum (green, solid) rises then falls—a shape no single neuron could produce.*

The sigmoid function alone can only increase monotonically. But observe Figure 10: by combining two sigmoids, we create a function that rises, peaks, and falls. What happened?

1. Two neurons, each computing a sigmoid with different weights and biases
2. Neuron 1's parameters position its sigmoid to rise at $x \approx 2$
3. Neuron 2's parameters position its sigmoid to rise at $x \approx 6$, but we *negate* its contribution
4. Adding them produces a function that rises, plateaus, then falls

The neurons use the *same* sigmoid function—they differ only in their parameters. By combining differently-positioned pieces, we create shapes impossible for any single piece alone. When this combined output feeds into another layer, we get hierarchical composition—the foundation of deep learning. With enough neurons, each contributing a differently-positioned piece, the

network can approximate *any* continuous function to arbitrary precision. This is the **Universal Approximation Theorem**, which we examine in Section 10.

4.2 Network Architecture: Layers and Connections

We organize neurons into **layers**:

- **Input layer:** Receives raw features $\mathbf{x}$ (not technically neurons—just data holders)
- **Hidden layers:** Neurons that transform inputs; called "hidden" because they connect neither directly to input nor output
- **Output layer:** Produces the final prediction $\hat{\mathbf{y}}$

In a **fully connected** (or **dense**) layer, every neuron receives input from *every* neuron in the previous layer.

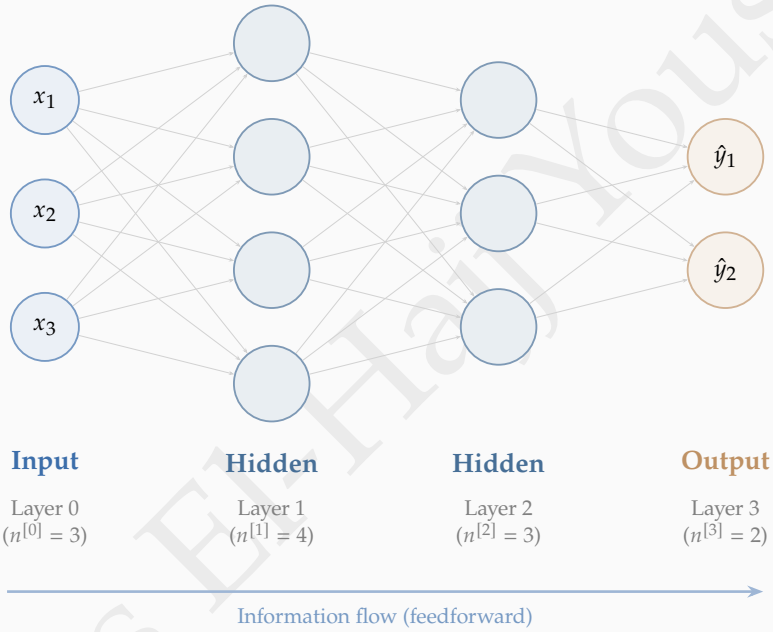

Figure 11: *A **multi-layer perceptron** (MLP) with two hidden layers. This network has $L = 3$ layers (excluding input). Each line represents a learnable weight. Information flows left to right without cycles.*

This architecture is called a **feedforward neural network** because information flows in one direction only—no cycles or feedback loops exist.

5 Forward Propagation: The Computational Heart

We now formalize how information flows from input to output. This computation—**forward propagation**—is the fundamental operation that every neural network performs.

5.1 Notation for Networks

Precise notation is essential for understanding and implementing neural networks. The following table summarizes our conventions, which follow the appendix exactly.

Symbol	Meaning
L	Total number of layers (excluding input layer 0)
$n^{[\ell]}$	Number of units in layer ℓ; note $n^{[0]} = n$ (input dimension)
$\mathbf{W}^{[\ell]} \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}}$	Weight matrix for layer ℓ; row j contains weights *into* unit j
$\mathbf{b}^{[\ell]} \in \mathbb{R}^{n^{[\ell]}}$	Bias vector for layer ℓ
$\mathbf{z}^{[\ell]} \in \mathbb{R}^{n^{[\ell]}}$	Pre-activation vector (before applying $g^{[\ell]}$)
$\mathbf{a}^{[\ell]} \in \mathbb{R}^{n^{[\ell]}}$	Activation vector (output of layer ℓ); note $\mathbf{a}^{[0]} = \mathbf{x}$
$g^{[\ell]}(\cdot)$	Activation function for layer ℓ
$\boldsymbol{\theta}$	All learnable parameters: $\{\mathbf{W}^{[\ell]}, \mathbf{b}^{[\ell]}\}_{\ell=1}^{L}$

Table 3: *Standard neural network notation. Superscript $[\ell]$ denotes layer index. Vectors are lowercase bold; matrices are uppercase bold.*

By convention, $\mathbf{a}^{[0]} = \mathbf{x}$—the input serves as the "activation" of layer 0.

Understanding Weight Matrix Dimensions

Why is $\mathbf{W}^{[\ell]} \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}}$?
Each *row* contains weights for one unit in layer ℓ. That unit receives $n^{[\ell-1]}$ inputs (from all units in layer $\ell - 1$), so each row has $n^{[\ell-1]}$ entries. With $n^{[\ell]}$ units in layer ℓ, we need $n^{[\ell]}$ rows.
The element $W_{jk}^{[\ell]}$ is the weight on the connection *from* unit k in layer $\ell - 1$ *to* unit j in layer ℓ.

5.2 The Forward Propagation Equations

Each layer performs the same two-step computation we saw for individual neurons, now expressed in matrix form.

Forward Propagation

Starting with $\mathbf{a}^{[0]} = \mathbf{x}$, for each layer $\ell = 1, 2, \ldots, L$:

$$\boxed{\begin{aligned} \mathbf{z}^{[\ell]} &= \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]} && \text{(linear combination)} \\ \mathbf{a}^{[\ell]} &= g^{[\ell]}\left(\mathbf{z}^{[\ell]}\right) && \text{(element-wise activation)} \end{aligned}} \tag{6}$$

The network's output is $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$.

These two equations encode the entire forward pass. Despite the apparent simplicity, this recursive structure—each layer transforming the previous layer's output—creates the representational power that makes deep learning effective.

5.3 Worked Example: Computing a Prediction

Let us trace forward propagation through a minimal but complete network:

- **Architecture:** 1 input → 2 hidden units (sigmoid) → 1 output (linear)
- **Purpose:** Approximate a "bump" function

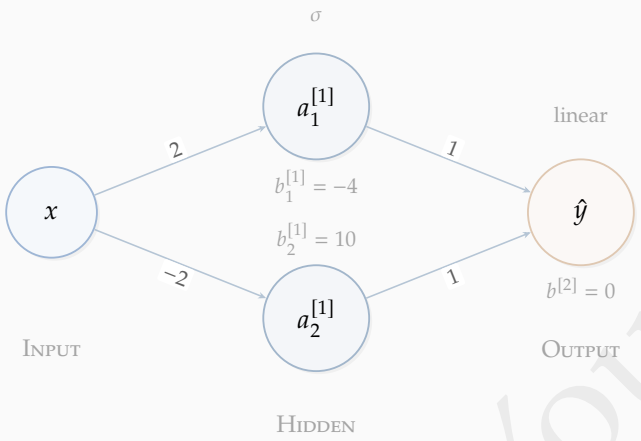

Figure 12: *A two-layer network with* $W^{[1]} = 2$ -2, $\mathbf{b}^{[1]} = -4$ 10, $\mathbf{W}^{[2]} = [1, 1]$, $b^{[2]} = 0$. *Hidden units activate at* $x \approx 2$ *(rising) and* $x \approx 5$ *(falling).*

Understanding the units geometrically:

- **Unit 1:** Computes $z_1^{[1]} = 2x - 4$, so the sigmoid transitions at $x = 2$. The positive weight means output rises with x.
- **Unit 2:** Computes $z_2^{[1]} = -2x + 10$, so the sigmoid transitions at $x = 5$. The negative weight means output falls with x.

The output layer combines them: $\hat{y} = a_1^{[1]} + a_2^{[1]}$.

Computing the Output for $x = 3.5$

Layer 1 (Hidden):

$$z_1^{[1]} = 2(3.5) - 4 = 3 \qquad a_1^{[1]} = \sigma(3) = \frac{1}{1 + e^{-3}} \approx 0.953$$

$$z_2^{[1]} = -2(3.5) + 10 = 3 \qquad a_2^{[1]} = \sigma(3) = \frac{1}{1 + e^{-3}} \approx 0.953$$

Layer 2 (Output):

$$z^{[2]} = 1 \cdot 0.953 + 1 \cdot 0.953 + 0 = 1.906$$

$$\hat{y} = a^{[2]} = z^{[2]} = 1.906 \quad \text{(linear activation)}$$

The network outputs $\hat{y} \approx 1.91$ for $x = 3.5$—near the peak of the bump function.

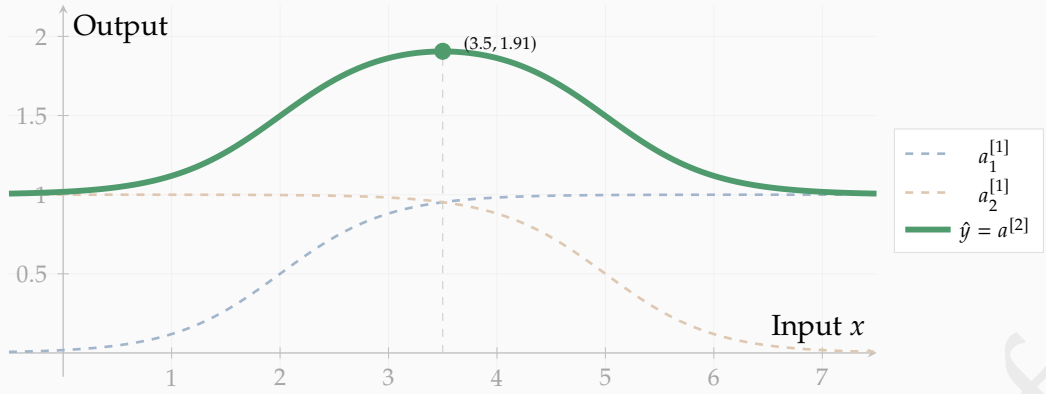

Figure 13: *The network's output (green) equals the sum of two hidden unit activations. The result is a bump—a shape impossible for any single neuron to produce.*

5.4 Batch Computation

For computational efficiency, we process many examples simultaneously. With m examples arranged as columns of $\mathbf{X} \in \mathbb{R}^{n^{[0]} \times m}$:

$$\begin{aligned} \mathbf{Z}^{[\ell]} &= \mathbf{W}^{[\ell]}\mathbf{A}^{[\ell-1]} + \mathbf{b}^{[\ell]}\mathbf{1}^\top \\ \mathbf{A}^{[\ell]} &= g^{[\ell]}\left(\mathbf{Z}^{[\ell]}\right) \end{aligned} \tag{7}$$

Here $\mathbf{A}^{[\ell]} \in \mathbb{R}^{n^{[\ell]} \times m}$ stores activations for all examples (each column is one example), and the term $\mathbf{b}^{[\ell]}\mathbf{1}^\top$ broadcasts the bias vector across all m columns.

Figure 14: *Batch computation dimensions. Each column of $\mathbf{A}^{[\ell-1]}$ is one example's activations; the matrix multiplication processes all m examples simultaneously.*

This vectorized formulation exploits highly optimized linear algebra libraries, making batch processing orders of magnitude faster than iterating over examples individually.

6 Solving XOR: Neural Networks in Action

We now return to the XOR problem that defeated every single neuron. With multiple neurons working together, we can finally solve it—and in doing so, reveal the profound mechanism by which neural networks succeed.

6.1 Architecture for XOR

We construct a minimal network:

- **Input:** 2 features (x_1, x_2)
- **Hidden:** 2 units with sigmoid activation
- **Output:** 1 unit with sigmoid activation (binary classification)

The following weights solve XOR:

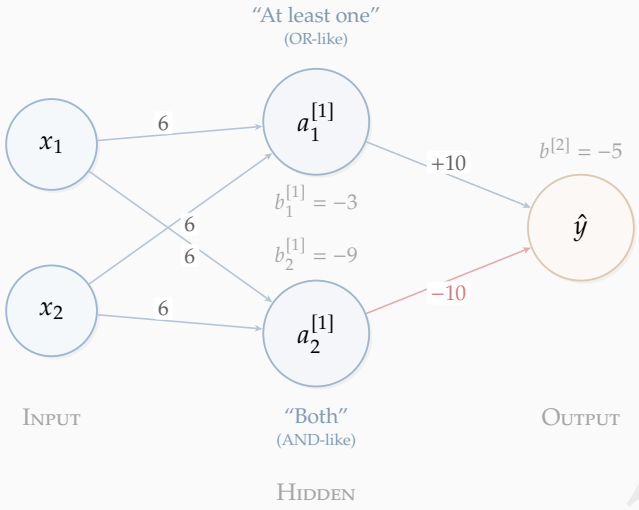

Figure 15: *A neural network solving XOR. Hidden unit $a_1^{[1]}$ fires when at least one input is 1; $a_2^{[1]}$ fires only when both are 1. The output computes "OR but not AND," yielding XOR.*

6.2 Why These Weights Work

With these weights, the sigmoid produces outputs close to 0 or 1, creating effective thresholds. Let us trace each input through the network:

Input		Hidden $a_1^{[1]}$		Hidden $a_2^{[1]}$		Output		Result
x_1	x_2	$z_1^{[1]}$	$a_1^{[1]}$	$z_2^{[1]}$	$a_2^{[1]}$	$z^{[2]}$	$\hat{y}$	
0	0	−3	≈ 0.05	−9	≈ 0	−4.5	≈ 0.01	✓
0	1	+3	≈ 0.95	−3	≈ 0.05	+4.0	≈ 0.98	✓
1	0	+3	≈ 0.95	−3	≈ 0.05	+4.0	≈ 0.98	✓
1	1	+9	≈ 1	+3	≈ 0.95	−4.5	≈ 0.01	✓

Table 4: *Forward propagation through the XOR network. Pre-activations z are computed as weighted sums; activations $a = \sigma(z)$. All four cases classify correctly.*

6.3 The Hidden Layer as Learned Representation

Examine the hidden layer's outputs for each input:

x_1	x_2	$a_1^{[1]}$	$a_2^{[1]}$	y	Interpretation
0	0	0	0	0	Neither input active
0	1	1	0	1	One input active, not both
1	0	1	0	1	One input active, not both
1	1	1	1	0	Both inputs active

Table 5: *Hidden layer representations (rounded). The two class-1 points* $(0,1)$ *and* $(1,0)$ *both map to the same hidden representation* $(1,0)$.

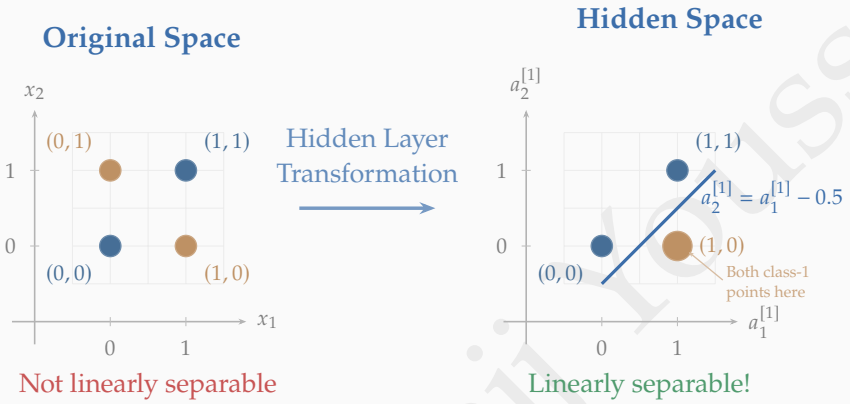

Figure 16: *The hidden layer transforms the input space so that XOR becomes linearly separable. Both class-1 inputs map to* $(1,0)$ *in hidden space. The decision boundary* $a_2^{[1]} = a_1^{[1]} - 0.5$ *separates the classes perfectly.*

The Fundamental Insight of Neural Networks

Hidden layers do not merely "process" data—they **learn new representations** in which problems become tractable.
For XOR, the hidden layer learned to answer two questions:

- $a_1^{[1]}$: "Is *at least one* input equal to 1?" (the OR function)
- $a_2^{[1]}$: "Are *both* inputs equal to 1?" (the AND function)

In this new coordinate system, XOR has a trivial solution: output 1 when $a_1^{[1]} = 1$ AND $a_2^{[1]} = 0$.
This is the core mechanism of deep learning: transform data into representations where the target relationship becomes simple—ideally, linear.

6.4 Two Complementary Perspectives

We have developed two complementary ways to understand neural networks. The **function approximation** perspective (Section 4) shows how networks approximate complex functions by combining simple pieces—each neuron contributes a positioned and scaled activation, and their combination can match any continuous target. The **representation learning** perspective (this section) shows how networks transform inputs into new spaces where tasks become easier—each hidden layer builds more abstract features from the previous layer's outputs.

These are complementary views of the same phenomenon: function approximation describes *what* the network computes; representation learning describes *how* it achieves expressiveness.

7 Designing Output Layers for Different Tasks

While hidden layers learn representations, the **output layer** must produce predictions in a format appropriate for the specific task. Different problems require different output designs.

7.1 Regression: Predicting Continuous Values

For regression tasks (predicting real-valued quantities such as price, temperature, or distance), the output layer uses **no activation function** (equivalently, the identity function):

$$\hat{y} = \mathbf{w}^{[L]\top}\mathbf{a}^{[L-1]} + b^{[L]} \tag{8}$$

This allows the network to output any real number, matching the unbounded nature of continuous targets.

7.2 Binary Classification: Probability Output

For binary classification, we use **sigmoid activation**, exactly as in logistic regression:

$$\hat{y} = \sigma\left(\mathbf{w}^{[L]\top}\mathbf{a}^{[L-1]} + b^{[L]}\right) = P(y = 1 \mid \mathbf{x}) \tag{9}$$

The output $\hat{y} \in (0, 1)$ represents the probability of class 1. We predict class 1 if $\hat{y} \geq 0.5$.

7.3 Multi-Class Classification: The Softmax Function

For problems with $K > 2$ mutually exclusive classes (such as digit recognition with $K = 10$), we need a probability distribution over all classes. The **softmax** function generalizes sigmoid to multiple classes.

The Softmax Function

Given pre-activation vector $\mathbf{z}^{[L]} = [z_1, z_2, \ldots, z_K]^\top$ (one value per class), softmax produces:

$$\text{softmax}(\mathbf{z}^{[L]})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} = P(y = k \mid \mathbf{x}) \tag{10}$$

for $k = 1, 2, \ldots, K$.

Key properties:

- **Valid probability distribution:** All outputs are positive and sum to exactly 1
- **Order-preserving:** Larger z_k yields larger probability
- **Amplifying:** Differences in z values become more pronounced in probabilities
- **Generalizes sigmoid:** For $K = 2$, softmax reduces to sigmoid applied to $(z_1 - z_2)$

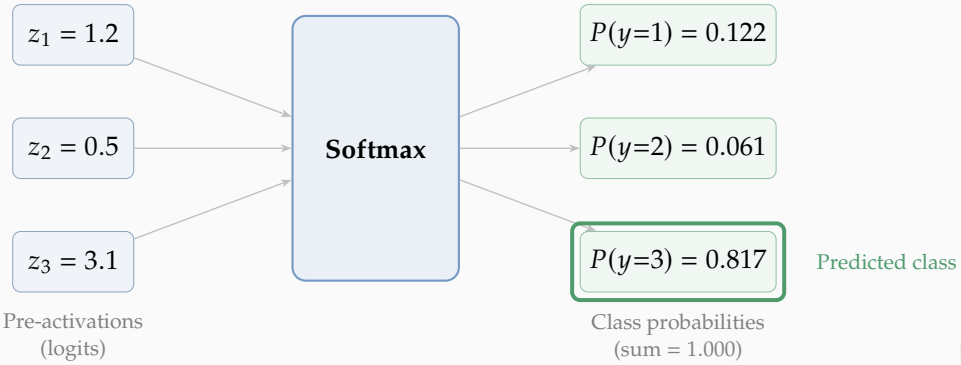

Figure 17: *Softmax converts raw scores (logits) into a probability distribution. The highest logit ($z_3 = 3.1$) produces the highest probability (81.7%), and the network predicts class 3.*

Softmax Calculation

For logits $\mathbf{z}^{[L]} = [1.2, 0.5, 3.1]$:

Step 1: Compute exponentials

$$e^{1.2} = 3.32, \quad e^{0.5} = 1.65, \quad e^{3.1} = 22.20$$

Step 2: Compute the normalization constant (sum of exponentials)

$$\sum_j e^{z_j} = 3.32 + 1.65 + 22.20 = 27.17$$

Step 3: Divide each exponential by the sum

$$\begin{aligned} P(y = 1) &= 3.32/27.17 = 0.122 \quad (12.2\%) \\ P(y = 2) &= 1.65/27.17 = 0.061 \quad (6.1\%) \\ P(y = 3) &= 22.20/27.17 = 0.817 \quad (81.7\%) \end{aligned}$$

The network predicts class 3 with 81.7% confidence.

In practice, computing e^{z_k} for large z_k causes numerical overflow. The standard solution is to subtract $\max_j z_j$ from all logits before exponentiating—this is mathematically equivalent but numerically stable.[1]

[1] The numerically stable softmax computes $\text{softmax}(\mathbf{z})_k = e^{z_k - \max_j z_j} / \sum_j e^{z_j - \max_j z_j}$. All modern deep learning frameworks implement this automatically.

7.4 Summary: Matching Output to Task

Task	Output Units	Activation	Output Interpretation
Regression	1 (or d for vector output)	None (linear)	Predicted value(s)
Binary classification	1	Sigmoid	$P(y = 1 \mid \mathbf{x})$
Multi-class (K classes)	K	Softmax	$[P(y = k \mid \mathbf{x})]_{k=1}^{K}$

Table 6: *Output layer design for common machine learning tasks.*

8 The Complete Picture: From Input to Prediction

Let us consolidate the forward propagation pipeline into a unified algorithmic view.

Forward Propagation Algorithm

Input: Data point $\mathbf{x}$, network parameters $\boldsymbol{\theta} = \{\mathbf{W}^{[\ell]}, \mathbf{b}^{[\ell]}\}_{\ell=1}^{L}$
Output: Prediction $\hat{\mathbf{y}}$

1. **Initialize:** Set $\mathbf{a}^{[0]} = \mathbf{x}$
2. **For each layer** $\ell = 1, 2, \ldots, L$:
   1. Compute pre-activation: $\mathbf{z}^{[\ell]} = \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}$
   2. Apply activation: $\mathbf{a}^{[\ell]} = g^{[\ell]}(\mathbf{z}^{[\ell]})$
3. **Return:** $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$

Complexity: $O\left(\sum_{\ell=1}^{L} n^{[\ell]} \cdot n^{[\ell-1]}\right)$ — linear in the total number of weights.

This algorithm is remarkably simple: initialize with input, loop through layers applying the two-step transformation, and return the final activation. The entire forward pass reduces to alternating matrix multiplications and element-wise nonlinearities.

9 Loss Functions: Measuring Prediction Error

To train a neural network, we need to measure how wrong its predictions are. This measurement is provided by a **loss function** (also called a cost function or objective function).

9.1 Mean Squared Error for Regression

For regression, we use the familiar mean squared error (MSE):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 \tag{11}$$

where $\boldsymbol{\theta}$ represents all the weights and biases in the network, $y^{(i)}$ is the true value, $\hat{y}^{(i)}$ is the network's prediction, and m is the number of training examples. The factor of $\frac{1}{2}$ simplifies the gradient computation during backpropagation.

MSE penalizes large errors more heavily than small ones (due to the squaring), encouraging the network to avoid predictions far from the truth.

9.2 Cross-Entropy for Classification

For classification, we use **cross-entropy loss**. The key intuition is **surprise**: if the network predicts a high probability for the correct class, we are not surprised—low loss. If it predicts a low probability for the correct class, we are shocked—high loss.

9.2.1 Binary Cross-Entropy

For binary classification with sigmoid output:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] \tag{12}$$

This is exactly the loss function from logistic regression. The loss is:

- $-\log(\hat{y})$ when the true label is 1 (penalizes low predicted probability)
- $-\log(1-\hat{y})$ when the true label is 0 (penalizes high predicted probability)

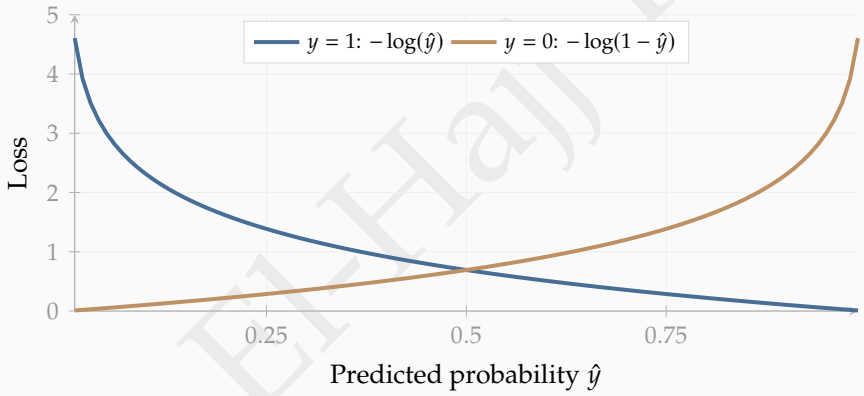

Figure 18: *Binary cross-entropy loss. When the true label is 1 (blue), the loss increases sharply as $\hat{y} \to 0$—predicting low probability for a true positive is heavily penalized. The reverse holds for true label 0 (orange).*

9.2.2 Categorical Cross-Entropy

For multi-class classification with softmax output:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log\hat{y}_k^{(i)} \tag{13}$$

where $y_k^{(i)} = 1$ if example i belongs to class k and 0 otherwise (one-hot encoding), and $\hat{y}_k^{(i)}$ is the predicted probability for class k.
Since only one $y_k^{(i)} = 1$ for each example (the correct class), this simplifies to:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\log\hat{y}_{c^{(i)}}^{(i)} \tag{14}$$

where $c^{(i)}$ is the correct class for example i. The loss is simply the negative log-probability assigned to the correct class—our "surprise" at getting the answer right.

9.3 The Loss Landscape

The loss function $\mathcal{L}(\boldsymbol{\theta})$ defines a **loss surface** over the space of all possible parameter values. Training seeks to find parameters that minimize this surface:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{15}$$

Unlike linear regression, neural network loss surfaces are **non-convex**. This creates several challenges:

- **Local minima:** Points where the loss is lower than all nearby points, but not the global minimum. Gradient descent can get "trapped" in these valleys.
- **Saddle points:** Points where the surface curves upward in some directions and downward in others—like a mountain pass. These can slow optimization significantly.
- **Flat regions:** Areas where the loss changes very little, making it difficult to determine which direction improves the objective.

Despite this complexity, gradient-based optimization works remarkably well in practice. Research suggests that most local minima in large networks have loss values close to the global minimum, and saddle points (rather than local minima) are the primary obstacle to fast convergence.

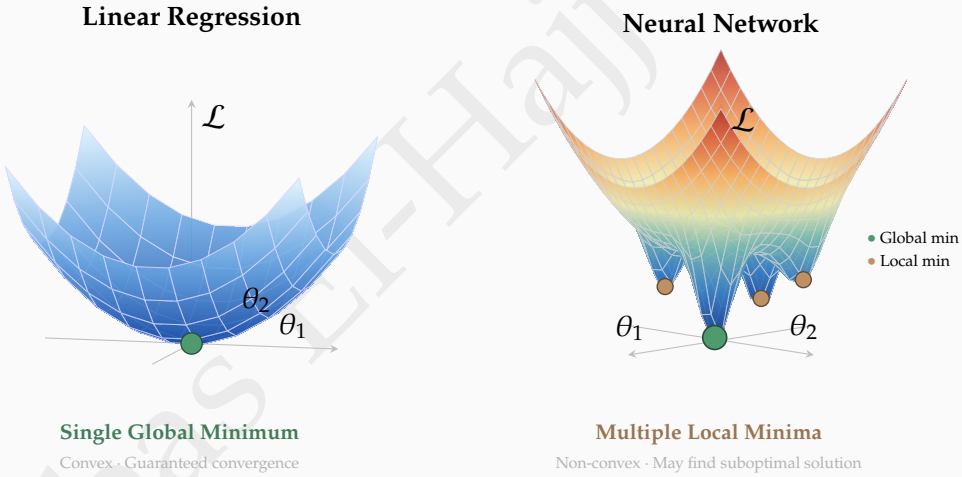

Figure 19: *Optimization landscapes.* ***Left:*** *Linear regression has a convex surface with a unique global minimum.* ***Right:*** *Neural networks have non-convex surfaces with multiple local minima; gradient descent may converge to different solutions depending on initialization.*

9.4 Summary: Matching Loss to Task

Task	Output Activation	Loss Function
Regression	Identity (none)	Mean Squared Error
Binary Classification	Sigmoid	Binary Cross-Entropy
Multi-class Classification	Softmax	Categorical Cross-Entropy

Table 7: *Standard combinations of output activation and loss function for common machine learning tasks.*

10 The Expressive Power of Neural Networks

We have seen that a two-layer network can solve XOR—something impossible for a single neuron. We have seen how combining sigmoids can create bump functions. This raises a natural question: what are the ultimate limits of neural network computation?

10.1 The Universal Approximation Theorem

A remarkable theorem, first proven by Cybenko (1989) for sigmoid networks and extended by Hornik et al. (1989), establishes that neural networks are extraordinarily powerful function approximators:

Universal Approximation Theorem (Informal)

Let $f : K \to \mathbb{R}$ be any continuous function on a compact (closed and bounded) set $K \subset \mathbb{R}^n$. For any desired accuracy $\epsilon > 0$, there exists a feedforward neural network with a single hidden layer containing a finite number of units such that the network output $g(\mathbf{x})$ satisfies:

$$\sup_{\mathbf{x} \in K} |f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$$

provided the hidden layer uses a non-constant, bounded, continuous activation function (such as sigmoid or tanh).

In plain terms: *neural networks with one hidden layer can approximate any continuous function to arbitrary precision, uniformly over any bounded region.*

10.2 What the Theorem Does and Does Not Guarantee

The universal approximation theorem is powerful but often misunderstood. It is crucial to understand its limitations:

What It Guarantees	What It Does NOT Guarantee
A network with enough units *exists* that approximates any continuous function	How to *find* the right parameters $\boldsymbol{\theta}$
Approximation is possible with a single hidden layer	The required number of units is practical (may be exponential)
Uniform approximation on bounded domains	Generalization to unseen data
Theoretical representational power	Computational tractability of training

The Existence-Learning Gap

The universal approximation theorem is *non-constructive*: it proves that good parameters exist but provides no algorithm to find them. A network might theoretically be able to represent the perfect classifier for your task, yet gradient descent might never discover the required θ.

Moreover, even if gradient descent finds parameters that fit the training data perfectly, there is no guarantee the network will generalize to new examples. The theorem says nothing about learning or generalization—only about representational capacity.

10.3 Why Depth Matters

Although a single hidden layer suffices in theory, practice strongly favors *deep* networks (many layers) over *wide* networks (many units in one layer). Several compelling reasons explain this preference:

1. **Efficiency:** Some functions require exponentially many units with one hidden layer but only polynomially many with multiple layers. For example, computing the parity of n bits requires 2^{n-1} units in one layer, but only $O(n)$ units across $O(\log n)$ layers.
2. **Compositionality:** Many real-world problems have hierarchical structure. Images are composed of objects, which are composed of parts, which are composed of edges. Language is composed of documents, sentences, phrases, words. Deep networks naturally exploit this hierarchy.
3. **Feature reuse:** Each layer can reuse features computed by previous layers, enabling more efficient representations.
4. **Empirical dominance:** Across virtually every domain—vision, language, speech, games—deeper networks outperform shallower ones with the same parameter count.

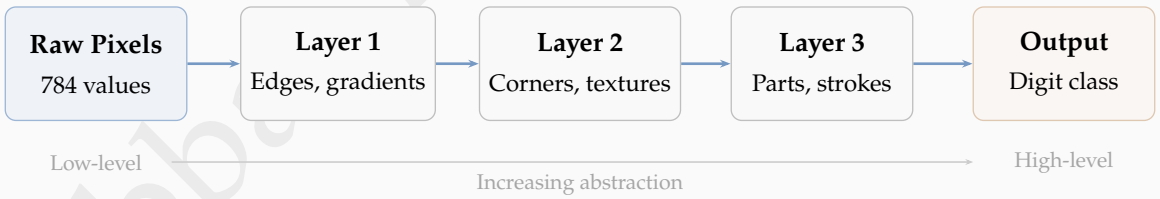

Figure 20: *Hierarchical feature learning in a deep network. Early layers detect simple patterns (edges); deeper layers combine these into complex concepts (digit identity). This hierarchy emerges from training—it is not manually designed.*

The intuition is that depth enables the network to build complex features **compositionally**. A deep network does not need to learn every possible digit directly from pixels; instead, it learns to detect edges, then uses edges to detect curves and corners, then uses those to detect strokes, and finally combines strokes into digit identities. Each layer reuses the representations computed by previous layers.

11 Counting Parameters

The **number of parameters** in a neural network determines both its representational capacity and its computational cost. Understanding parameter counts helps us reason about model complexity.

11.1 Parameters in a Fully Connected Layer

For a fully connected layer ℓ connecting $n^{[\ell-1]}$ inputs to $n^{[\ell]}$ outputs:

- **Weights:** $n^{[\ell]} \times n^{[\ell-1]}$ (one weight per connection)
- **Biases:** $n^{[\ell]}$ (one per unit)
- **Total for layer ℓ:** $n^{[\ell]} \times n^{[\ell-1]} + n^{[\ell]} = n^{[\ell]}(n^{[\ell-1]} + 1)$

11.2 Total Parameters in a Network

For a network with layers of sizes $n^{[0]}, n^{[1]}, \ldots, n^{[L]}$:

$$|\boldsymbol{\theta}| = \sum_{\ell=1}^{L} n^{[\ell]} \left(n^{[\ell-1]} + 1 \right) \tag{16}$$

where $|\boldsymbol{\theta}|$ denotes the total number of learnable parameters.

Parameter Count for MNIST

Consider a network for digit classification with architecture:
784 $\rightarrow$ **256** $\rightarrow$ **128** $\rightarrow$ **10** (input $\rightarrow$ hidden $\rightarrow$ hidden $\rightarrow$ output)

$$\begin{aligned} \text{Layer 1:} \quad & 256 \times (784 + 1) = 256 \times 785 = 200{,}960 \\ \text{Layer 2:} \quad & 128 \times (256 + 1) = 128 \times 257 = 32{,}896 \\ \text{Layer 3:} \quad & 10 \times (128 + 1) = 10 \times 129 = 1{,}290 \\ \text{Total:} \quad & |\boldsymbol{\theta}| = 235{,}146 \text{ parameters} \end{aligned}$$

This modest network has over 235,000 learnable parameters—all of which must be discovered from data.

The first hidden layer typically dominates the parameter count because it connects to the high-dimensional input. In the MNIST example, Layer 1 contains 85% of all parameters. This observation motivates architectures (like convolutional networks) that reduce input dimensionality before applying dense layers.

12 Looking Ahead: The Training Problem

This chapter has defined **what** a neural network computes. Given an input $\mathbf{x}$, we can propagate it forward through layers to compute a prediction $\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x})$ and compare it to the true label using a loss function $\mathcal{L}$. But we have not addressed the central question: **how do we find good parameter values $\boldsymbol{\theta}$?**

Training a neural network means solving $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$. The XOR network in Section 6 used carefully hand-picked weights; for realistic problems with hundreds of thousands of

parameters, manual selection is impossible. The next chapter develops **gradient descent** and **backpropagation**—the algorithms that make neural network training possible by efficiently computing the gradient $\nabla_\theta \mathcal{L}$ and using it to iteratively improve parameters.

Chapter Summary

1. **A single sigmoid neuron is logistic regression:** It computes $\sigma(\mathbf{w}^\top \mathbf{x} + b)$, defining a linear decision boundary. This is your conceptual anchor.
2. **Linear classifiers cannot solve XOR:** When the correct output depends on feature interactions rather than individual features, flat boundaries fail.
3. **Combining neurons enables nonlinearity:** By adding differently-positioned activation functions, networks can approximate any continuous function (Universal Approximation Theorem).
4. **Hidden layers learn representations:** They transform inputs into new spaces where problems become linearly separable—this is the core mechanism of deep learning.
5. **Output design matches the task:** Sigmoid for binary classification, softmax for multi-class, identity for regression; paired with cross-entropy or MSE loss respectively.

We have fully specified the **forward pass**: input → prediction → loss. The next chapter addresses the **backward pass**: using the loss to update parameters.

Appendix: Notation

Network Architecture

L	Number of layers (excl. input)
ℓ	Layer index, $\ell \in \{0, 1, \ldots, L\}$
$n^{[\ell]}$	Number of units in layer ℓ
n	Input features (equiv. $n^{[0]}$)
$n_{\text{in}}, n_{\text{out}}$	Input/output dimensions

Data and Indexing

m	Number of training examples
K	Number of classes (classification)
$(\cdot)^{(i)}$	Quantity for training example i
$(\cdot)^{[\ell]}$	Quantity for layer ℓ
$(\cdot)_j$	j-th component of a vector

Inputs, Outputs, and Targets

$\mathbf{x} \in \mathbb{R}^n$	Input vector
$\mathbf{X}$	Data matrix (examples as columns)
y	Target value (scalar)
$\mathbf{y}$	Target vector
$\hat{y}$	Predicted output (scalar)
$\hat{\mathbf{y}}$	Predicted output (vector)

Parameters

$\mathbf{W}^{[\ell]}$	Weight matrix for layer ℓ
$\mathbf{b}^{[\ell]}$	Bias vector for layer ℓ
$w_{jk}^{[\ell]}$	Weight: unit k (layer $\ell-1$) to j (layer ℓ)
w, b	Scalar weight and bias (single unit)
$\boldsymbol{\theta}$	All parameters: $\{\mathbf{W}^{[\ell]}, \mathbf{b}^{[\ell]}\}_{\ell=1}^{L}$

Forward Propagation

$\mathbf{z}^{[\ell]}$	Pre-activation at layer ℓ
$\mathbf{a}^{[\ell]}$	Activation at layer ℓ
$\mathbf{a}^{[0]} = \mathbf{x}$	Input layer convention
$g^{[\ell]}(\cdot)$	Activation function for layer ℓ
$\sigma(z)$	Sigmoid: $1/(1+e^{-z})$
$\tanh(z)$	Hyperbolic tangent
$\text{ReLU}(z)$	Rectified linear unit: $\max(0, z)$
$\text{softmax}(\mathbf{z})_k$	Softmax: $e^{z_k}/\sum_j e^{z_j}$

Loss and Optimization

$\mathcal{L}(\boldsymbol{\theta})$	Loss function
$\nabla_{\boldsymbol{\theta}} \mathcal{L}$	Gradient of loss w.r.t. parameters
$\partial \mathcal{L}/\partial \theta$	Partial derivative w.r.t. θ
α	Learning rate

Backpropagation

$\delta^{[\ell]}$	Error signal at layer ℓ: $\partial \mathcal{L}/\partial \mathbf{z}^{[\ell]}$
$\delta_j^{[\ell]}$	Error signal for unit j in layer ℓ

General Mathematics

$\mathbb{R}$	Set of real numbers
$\mathbb{R}^n$	n-dimensional Euclidean space
$\mathbf{v}^\top$	Transpose of vector $\mathbf{v}$
$\|\mathbf{v}\|$	Euclidean norm of vector $\mathbf{v}$
$\|\mathbf{v}\|_2$	Euclidean (L_2) norm
$\odot$	Element-wise (Hadamard) product

Conventions

- Vectors are lowercase bold: $\mathbf{x}, \mathbf{z}, \mathbf{a}$
- Matrices are uppercase bold: $\mathbf{W}, \mathbf{X}$
- Scalars are italic: m, L, α
- Layer indices as superscripts in brackets: $\mathbf{W}^{[\ell]}$
- Example indices as superscripts in parentheses: $\mathbf{x}^{(i)}$
- Vector components as subscripts: x_j, $a_k^{[\ell]}$