Deep Learning (1): Basics of Neural Networks

(for high school and first-year university students)

From a model of a neuron to multilayer networks

Planned and edited by Yoshio Okamoto / Written by Claude Opus 5.5 (Anthropic)

Ver. 0.2 October 7, 2026

Note: The following content was generated by AI. The editor has checked it, but please be aware that it may still contain errors or mistaken assumptions.

© 2026 Yoshio Okamoto. Free to use for educational purposes (please cite the source).

Introduction

Telling whether a photo shows a dog or a cat, reading handwritten digits, writing the continuation of a text: inside the AI that does such things is a **neural network**, "a computing mechanism that takes numbers in and puts numbers out." Its parts are very simple, and by stacking the same parts in many stages, it becomes able to do surprisingly complicated things.

In the previous document, "Supervised and Unsupervised Learning," we organized the framework of supervised learning (model → loss function → minimization) using linear regression and logistic regression as examples.[a] Starting now, in four parts, we deal with **deep learning**, which extends that framework.

[a] Some of the other documents referred to in this booklet are currently available in Japanese only.

Part	Title	Main content
Part 1	**Basics of Neural Networks**	Model of a neuron, limits of one layer, multiple layers, activation functions, expressive power
Part 2	Backpropagation and Training in Practice	Chain rule, derivation of backpropagation, stochastic gradient descent, vanishing gradients, preventing overfitting
Part 3	Convolutional Neural Networks	Convolution, weight sharing, pooling, image recognition
Part 4	Sequence Data and Transformers	RNN, Attention, Transformer

Part 1 deals with **what kind of function a neural network is**. How to find the weights (training) is the topic of Part 2, so this time we concentrate on what a network computes when its weights are given.

How to read this booklet The main text is written so that it can be read with high-school mathematics (linear functions and inequalities, exponential functions, the summation sign $\sum$, and differentiation). Parts that need tools learned at university (notation with matrices and vectors, error estimates) are collected in the **footnotes** and in the **appendices** at the end. You can follow the main text even if you skip the appendices. All numerical values in the examples were actually computed (programs in Appendix C).

Notation

- $x_1, x_2, \ldots$ are inputs, $w_1, w_2, \ldots$ are weights, and b is a bias.
- $\sum_{j=1}^{d} w_j x_j$ means $w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$.
- $\hat{y}$ ("y hat") denotes the output (predicted value) of the network.

1 A model of a neuron

1.1 The artificial neuron

A nerve cell in the brain (a **neuron**) receives signals from many other neurons, and when their total exceeds a certain level, it sends out a signal itself. An extremely simplified mathematical model of this is the **artificial neuron** (Figure 1).

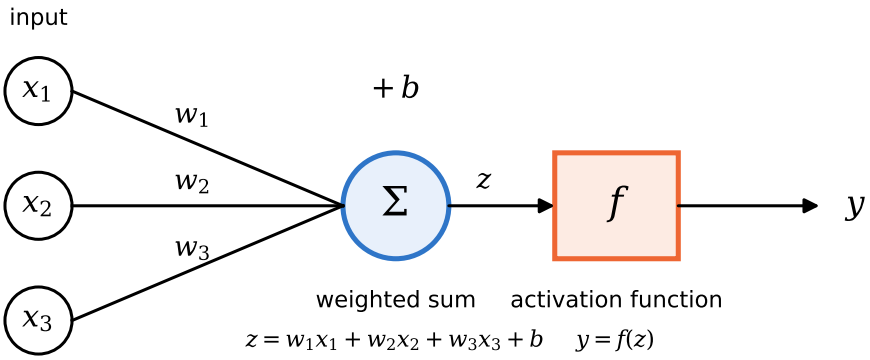


Figure 1 An artificial neuron. The inputs x_i are multiplied by the weights w_i and added, the bias b is added, and the resulting value z is passed through the activation function f to give the output y.

When there are d inputs $x_1, \ldots, x_d$ with weights $w_1, \ldots, w_d$, the neuron computes[1]

$$z = \sum_{j=1}^{d} w_j x_j + b = w_1 x_1 + \cdots + w_d x_d + b, \qquad y = f(z).$$

Here b is called the **bias** (it plays the role of a threshold), and f is called the **activation function**.

> Let us compare this with deciding who passes a university entrance exam. Each department multiplies the scores in English, mathematics and science (the inputs) by factors expressing "how much it values each subject" (the **weights**), adds them up, and compares the total with the passing line (the **bias**) to decide pass or fail (the output). A science department makes the factor for science large, and a literature department the factor for English. By changing the weights, the same mechanism can make different decisions. An artificial neuron performs exactly this computation.

- A neuron in which f is the **step function** (1 if $z \geq 0$, 0 if $z < 0$) is called a **perceptron** (Rosenblatt, 1958). Pass is 1 and fail is 0.

[1] At university, the inputs and weights are collected into vectors $\mathbf{x} = (x_1, \ldots, x_d)^\mathsf{T}$ and $\mathbf{w} = (w_1, \ldots, w_d)^\mathsf{T}$, and this is written $z = \mathbf{w}^\mathsf{T}\mathbf{x} + b$ ($\mathbf{w}^\mathsf{T}\mathbf{x}$ is the dot product; Appendix A).

- A neuron in which f is the **sigmoid function** $\sigma(z) = 1/(1+e^{-z})$ is **exactly the logistic regression** of the previous document. Logistic regression was a model that passes a weighted sum of the inputs through the sigmoid function and outputs "the probability of passing."

> In other words, the logistic regression of the previous document was "**a neural network made of a single neuron**."

1.2 Making logic operations with one neuron

Let us make two-input logic operations with a step-function neuron. The inputs x_1, x_2 are 0 or 1.

x_1	x_2	AND	OR	NAND
0	0	0	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND is "1 only when both are 1," OR is "1 if either one is 1," and NAND is the opposite of AND (Not AND). In terms of electric circuits, AND is a circuit with two switches connected **in series** (the bulb lights only when both are on), and OR is one with them connected **in parallel** (it lights when either one is on).

Choosing the weights and bias as follows gives each logic operation.

Operation	w_1	w_2	b	$z = w_1x_1 + w_2x_2 + b$
AND	1	1	-1.5	$z = 0.5 \geq 0$ only when both are 1
OR	1	1	-0.5	$z \geq 0.5$ if either is 1
NAND	-1	-1	1.5	the signs of AND reversed

For example, for AND with $(x_1, x_2) = (1, 0)$, $z = 1 - 1.5 = -0.5 < 0$, so the output is 0; with $(1, 1)$, $z = 2 - 1.5 = 0.5 \geq 0$, so the output is 1. AND and OR have the same weights; only the passing line (the bias) differs. If the line is strict (-1.5), "both are 1" is needed; if it is lenient (-0.5), "at least one is 1" is enough to pass.

1.3 Seen in a figure: a straight line divides the plane in two

The boundary where the neuron's output switches is $z = w_1x_1 + w_2x_2 + b = 0$, that is, **a straight line in the plane**. For example, the boundary of AND is $x_1 + x_2 - 1.5 = 0$, that is, the line $x_2 = -x_1 + 1.5$.

> One neuron **draws a single straight line with a ruler** on the plane of inputs, and outputs 1 on one side and 0 on the other (Figure 2). This property is expressed by saying that "one neuron can only do **linear separation**."[2]

[2]With three inputs, the boundary $w_1x_1 + w_2x_2 + w_3x_3 + b = 0$ is a plane in three-dimensional space; in general, with d inputs it is a **hyperplane** in d-dimensional space.

2 The limit of a single neuron: the XOR problem

2.1 XOR cannot be separated by one straight line

The exclusive OR, XOR, outputs 1 when exactly one of the inputs is 1.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

A familiar example of XOR is **the light on a staircase**. There are switches at the bottom and the top of the stairs, and flipping either switch turns the light on or off. The light is on when the two switches are in "different" states and off when they are "the same": exactly XOR.

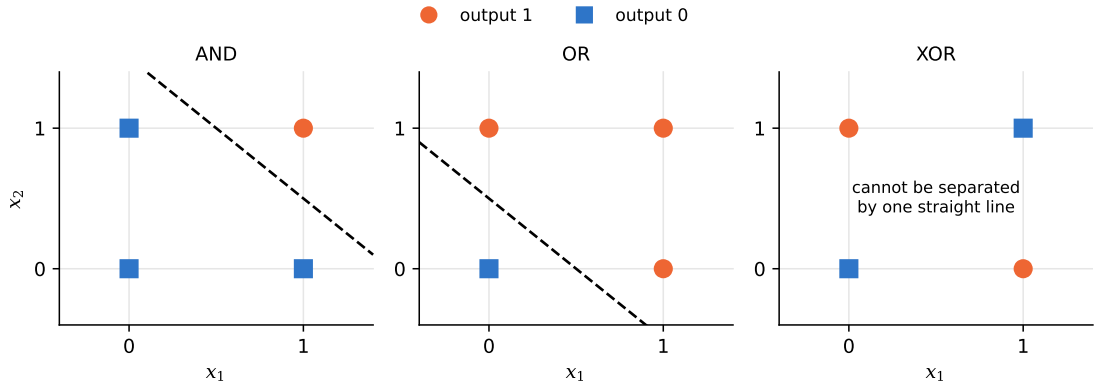


Figure 2 Inputs and outputs of AND, OR and XOR. Circles are output 1, squares are output 0, and the dashed lines are the boundary $z = 0$ of a neuron. AND and OR can be separated by one straight line, but XOR cannot.

As on the right of Figure 2, the points with output 1 ($(0,1)$ and $(1,0)$) and those with output 0 ($(0,0)$ and $(1,1)$) lie diagonally, like the white and black squares of a chessboard, so **no single ruler line can separate them, however you draw it**.

Let us check If XOR could be made with a step-function neuron, then for the four inputs

$$\begin{aligned}(0,0):&\ b < 0\\(0,1):&\ w_2 + b \geq 0\\(1,0):&\ w_1 + b \geq 0\\(1,1):&\ w_1 + w_2 + b < 0\end{aligned}$$

would all have to hold. Adding the second and third inequalities gives $w_1 + w_2 + 2b \geq 0$, that is, $w_1 + w_2 + b \geq -b$. From the first, $-b > 0$, so $w_1 + w_2 + b > 0$, which contradicts the fourth. Therefore, **whatever w_1, w_2, b are chosen, XOR cannot be made**. Even with the sigmoid function or others, as long as the output is split at 0.5 the boundary is a straight line, so the conclusion is the same.

This limit was pointed out by Minsky and Papert in 1969, and it became one of the reasons why research on neural networks stalled for a while.

2.2 The solution: stack neurons

XOR can be made by combining AND, OR and NAND:

$$\mathrm{XOR}(x_1, x_2) = \mathrm{AND}\big(\,\mathrm{OR}(x_1, x_2),\ \mathrm{NAND}(x_1, x_2)\,\big).$$

In words, "**at least one is 1** (OR) **and not both are 1** (NAND)." That is, we only need a two-layer structure that **computes OR and NAND in the first stage and feeds the two results into AND in the second stage** (Figure 3(a)).

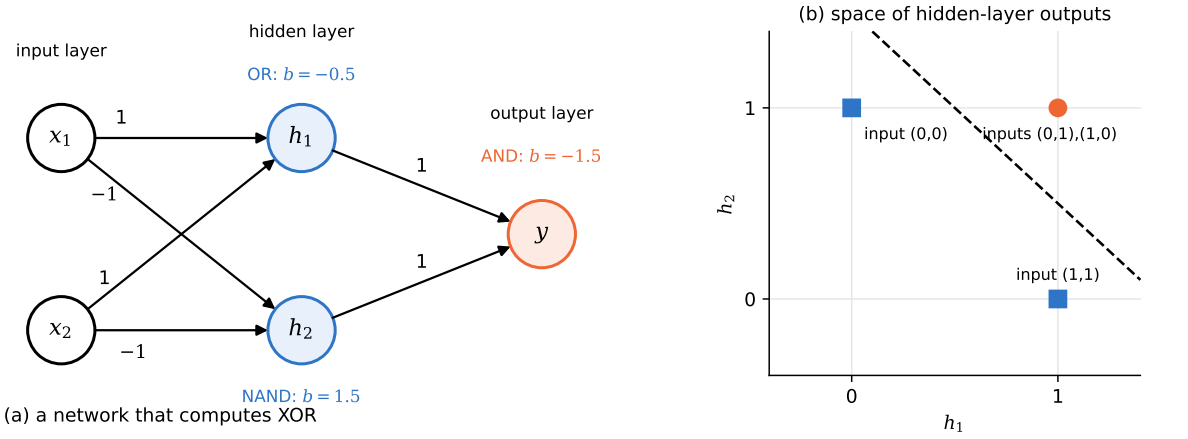


Figure 3 (a) A two-layer network that computes XOR. The numbers on the arrows are weights and b is the bias (the activation function is the step function). (b) The space of hidden-layer outputs (h_1, h_2). The four inputs are moved to three points, which can now be separated by one straight line (dashed).

The computation for each input is as follows (program in Appendix C.1).

x_1	x_2	$h_1 = \mathrm{OR}$	$h_2 = \mathrm{NAND}$	$y = \mathrm{AND}(h_1, h_2)$
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

2.3 The hidden layer transforms the "representation"

Why does using two layers solve the problem? Figure 3(b) replaces the input (x_1, x_2) with the output of the first layer, (h_1, h_2).

- Input $(0, 0) \to (h_1, h_2) = (0, 1)$
- Inputs $(0, 1)$ and $(1, 0) \to$ both $(1, 1)$
- Input $(1, 1) \to (1, 0)$

The points, which lay diagonally in the original plane and could not be separated, are moved in the (h_1, h_2) plane to **an arrangement that can be separated by the single straight line** $h_1 + h_2 = 1.5$. The second-layer neuron only performs linear separation in this new plane.

> **The first layer transforms the data into a form that the second layer can separate with a straight line.**

You cannot divide the four squares of a chessboard drawn on paper into white and black with one ruler line. But if you fold the paper along the diagonal joining the two black squares, the two white squares lie exactly on top of each other and the two black squares line up on the fold. Now one line parallel to the fold can separate white and black. What the hidden layer does corresponds to this **"folding of the paper."**

This "transforming the input into another representation in which the problem is easier to solve" is the very essence of multilayer neural networks. The layers between the input layer and the output layer cannot be seen directly from outside, so they are called **hidden layers**.

Here a person thought out and decided the weights, but in real neural networks, **the weights are learned from data, including what transformation to make** (Part 2). Not needing people to design how features are extracted is a major characteristic of deep learning.

3 Multilayer neural networks

3.1 Stacking layers

When the hidden layer has m neurons, each neuron k ($k = 1, \ldots, m$) receives the same inputs $x_1, \ldots, x_d$ and computes, with its own weights and bias,

$$h_k = f\Big(\sum_{j=1}^{d} W_{kj}\, x_j + b_k\Big).$$

W_{kj} is "the weight from input j to neuron k"; it has two subscripts to distinguish "which neuron's" weight "for which input." The outputs $h_1, \ldots, h_m$ of the m neurons become the inputs to the next layer.

In a network with L layers stacked,

$$\text{input} \longrightarrow \text{layer 1} \longrightarrow \text{layer 2} \longrightarrow \cdots \longrightarrow \text{layer } L \text{ (output } \hat{y}),$$

the output of each layer becomes the input of the next, and the computation proceeds in order.

The computation is the same in every layer: **repeating the operation "take a weighted sum, add a bias, pass through the activation function."** Computing in order from the input is called **forward propagation**.
Compared to an assembly line in a factory, each layer is one process. It combines the parts (numbers) delivered from the previous process, processes them a little, and sends them on to the next. Each process is simple, but stacking many processes produces a complex product.

At university, the computation of one layer is written with matrices and vectors as a single formula, $\mathbf{h} = f(W\mathbf{x} + \mathbf{b})$. This is collected in Appendix A, including the XOR network of Figure 3(a) written with matrices. The function of the last layer (the output layer) is treated in Section 6.

3.2 What "deep" means

A network with many hidden layers stacked is called a **deep neural network**, and its training is called **deep learning**. "Deep" means that there are many layers (many processes).

3.3 The number of parameters

For example, consider a network whose input is an image of a handwritten digit with $28 \times 28 = 784$ pixels, with 100 hidden units and 10 outputs (digits 0–9). Each of the 100 hidden neurons has a weight for every one of the 784 inputs, and one bias each.

Layer	Weights	Biases	Computation
Input → hidden	78,400	100	784×100
Hidden → output	1,000	10	100×10
Total	**79,510**		

> Think of each weight and bias as a **knob** on a radio. Even this small network has about 80,000 knobs. In large language models (LLMs), the number of knobs (the number of parameters) reaches billions to hundreds of billions.[3] How to set these knobs is the "training" of Part 2.

4 Activation functions

4.1 Typical activation functions

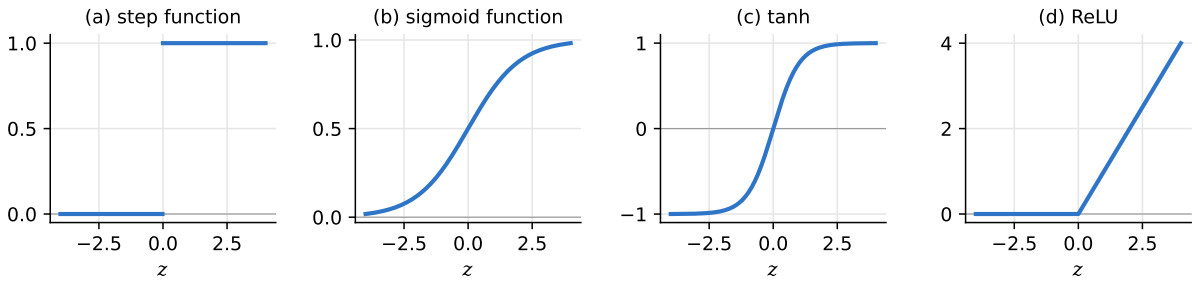


Figure 4 Typical activation functions.

Name	Formula	Range	Derivative
Step function	$1\ (z \geq 0),\ 0\ (z < 0)$	0 or 1	0 (discontinuous at $z = 0$)
Sigmoid function	$\sigma(z) = \dfrac{1}{1+e^{-z}}$	$0 < \sigma < 1$	$\sigma(z)\big(1-\sigma(z)\big)$
tanh (hyperbolic tangent)	$\tanh z = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$-1 < \tanh z < 1$	$1 - \tanh^2 z$
ReLU	$\max(0, z)$	0 or more	$1\ (z > 0),\ 0\ (z < 0)$

$\max(0, z)$ means "the larger of 0 and z": it is z if $z > 0$ and 0 if $z \leq 0$.

- The **step function** has a derivative that is always 0 (except at $z = 0$), so gradient descent cannot be used.[4] It is like looking for "the downhill direction" on a perfectly flat plateau and not finding one. It is not used in networks that are trained.

[3] For example, an LLM called an "8B model" has about 8 billion parameters (B stands for billion), 100,000 times the scale of this example.

[4] Gradient descent was the method of "moving the weights a little in the direction opposite to the slope of the loss." If the slope is 0 everywhere, you cannot tell which way to move.

- The **sigmoid function** and **tanh** are smooth and have been used for a long time.[5] They round off the corner of the step function, turning "a steep step" into "a gentle slope."
- **ReLU** (Rectified Linear Unit) is easy to compute, and since its derivative is always 1 for $z > 0$, training proceeds easily; it is the most widely used in today's deep learning. The reason is explained in Part 2 (vanishing gradients).

> The "rectified" of ReLU means "made to flow in one direction." It has the same shape as **half-wave rectification by a diode** in an electric circuit, which lets current flow in only one direction and makes the reverse direction 0. Positive values pass through unchanged, and negative values become 0.

4.2 Why a nonlinear activation function is needed

What happens if we remove the activation function ($f(z) = z$) and stack layers? Consider the simple case of one input and one hidden neuron:

$$h = w_1 x + b_1, \qquad \hat{y} = w_2 h + b_2.$$

Substituting the first into the second gives

$$\hat{y} = w_2(w_1 x + b_1) + b_2 = (w_2 w_1)\, x + (w_2 b_1 + b_2),$$

a linear function with slope $w_2 w_1$ and intercept $w_2 b_1 + b_2$; that is, it becomes **the same as a one-layer network**. However many layers are stacked, a composition of linear functions is a linear function (the same holds with many inputs and neurons; Appendix A.3).

> If you enlarge something to 200% on a copy machine and then enlarge that to 150%, the result is the same as a single enlargement to 300%. However many enlargements or reductions (linear functions) you stack, you only get a single enlargement or reduction, and **you can never bend a straight line**.
> **Only by inserting nonlinear activation functions does stacking layers become meaningful.**

XOR could be solved because a nonlinear function, the step function, was inserted. The "folding of the paper" above cannot be done with linear functions (enlargement, reduction and translation). Making the fold is the role of the nonlinear activation function.

5 The expressive power of neural networks

5.1 Can any function be represented?

> It is known that even with only one hidden layer, if the hidden layer has enough neurons, any continuous function (on a finite interval) can be approximated to any desired accuracy. This is called the **universal approximation theorem** (Cybenko, 1989, and others).

5.2 Making a curve by adding ReLUs

To get a feel for the universal approximation theorem, let us approximate $y = \sin 2\pi x$ $(0 \le x \le 1)$ using ReLU.

[5] tanh has the shape of the sigmoid function stretched twice vertically and lowered by 1 ($\tanh z = 2\sigma(2z) - 1$).

The shifted ReLU $\max(0,\, x - c)$ is 0 for $x < c$ and a line with slope 1 for $x > c$: a "broken line" bent at $x = c$. Adding such functions with weights makes **a broken line bent at any positions by any angles**:

$$\hat{y}(x) = b_0 + \sum_{k=0}^{S-1} a_k \max(0,\, x - c_k).$$

This is nothing but a one-input, one-output network with S ReLU neurons in the hidden layer (neuron k has weight 1 and bias $-c_k$, and its weight to the output layer is a_k).

How to choose the weights Divide the interval $[0, 1]$ into S equal parts, and let $y_k = \sin(2\pi k/S)$ be the function value at the division point $c_k = k/S$. Suppose we want the slope of the broken line on interval k ($c_k \le x \le c_{k+1}$) to be $s_k = (y_{k+1} - y_k) \div \frac{1}{S}$. Since $\max(0,\, x - c_k)$ has slope 1 for $x > c_k$, each time x passes c_k, the slope of the broken line increases by a_k. Therefore, if we set

$$b_0 = y_0, \qquad a_0 = s_0, \qquad a_k = s_k - s_{k-1} \quad (k = 1, \ldots, S-1),$$

the slope on each interval is exactly s_k, and the broken line passes through all the division points.

The broken lines made this way are shown in Figure 5 (program in Appendix C.2).

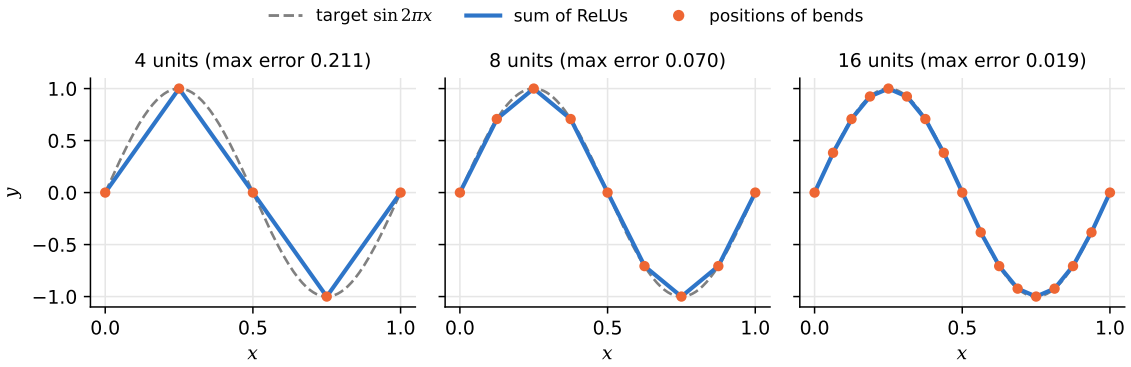


Figure 5 Approximating $\sin 2\pi x$ by adding ReLUs. The number in each title is the number of hidden units (neurons). The gray dashed line is the target function, the blue line is the output of the network, and the dots are the positions of the bends.

Number of hidden units S	4	8	16
Maximum error	0.211	0.070	0.019

Doubling the number of units makes the error about 1/4, so the accuracy can be raised as much as we like.[6]

This is the same as approximating a circle with regular polygons. For a regular hexagon inscribed in a circle of radius 1, the largest gap between a side and the circle is 0.134; for a regular 12-gon it is 0.034, and for a 24-gon 0.0086: each doubling of the number of sides makes it about 1/4. This is the idea Archimedes used to find π. **Even with straight parts, increasing their number brings you as close to a curve as you like.**

[6]The error decreases in proportion to $1/S^2$. The reason (an error estimate for broken-line approximation) is given in Appendix B. The gap for a regular polygon is $1 - \cos(\pi/n)$, which is proportional to $1/n^2$ for the same reason.

5.3 Then why make it "deep"?

The universal approximation theorem seems to say that "one hidden layer is enough." However, the theorem guarantees nothing about **how many units are needed**.

> In practice, representing a complicated function with one layer often needs an enormous number of units, and it is known that there are cases in which **stacking layers can represent the same function with far fewer parameters**.

Intuitively, this is because, as in the XOR example of Section 2, each layer can "combine the outputs of the previous layer to build gradually more complex features." Just as letters are combined into words and words into sentences, building up combinations of parts step by step is much more efficient than making everything at once. For example, in image recognition it has been observed that shallow layers come to represent edges and textures, deeper layers parts such as eyes and wheels, and still deeper layers whole objects (Part 3).

Also, the universal approximation theorem only says that weights that approximate the function **exist**; whether those weights **can be found from data** is a separate question. Even if you are guaranteed that treasure is buried somewhere, you cannot get it if you do not know how to dig it up. This is the subject of Part 2.

6 The output layer and the loss function

For the function of the last layer (the output layer) and the loss function, the same combinations as in the previous document are used, depending on the task.

Task	Output-layer function	Loss function	Method in the previous document
Regression	identity ($g(z) = z$)	squared error	linear regression
Two-class classification	sigmoid function	cross-entropy	logistic regression
Multi-class classification	softmax function	cross-entropy	multi-class logistic regression

> In other words, a neural network can be seen as **the linear regression or logistic regression of the previous document, with many hidden layers that transform the input added in front**. The hidden layers transform the data into a representation that is easy to handle (fold the paper), and the last layer uses that representation for regression or classification (draws a line with a ruler).

7 Summary

> 1. An artificial neuron computes $y = f(w_1x_1 + \cdots + w_dx_d + b)$: a weighted sum of the inputs plus a bias, passed through an activation function. With the sigmoid function, it is the same as logistic regression.
> 2. The boundary of one neuron is a straight line (in general, a hyperplane), so problems that cannot be linearly separated, such as XOR, cannot be solved.
> 3. Stacking layers lets hidden layers transform the input into another representation, so that problems that cannot be linearly separated can be solved.
> 4. A multilayer network repeats "take a weighted sum, add a bias, pass through a nonlinear

activation function." Without nonlinearity, any number of layers is the same as one layer.
5. With enough hidden units, any continuous function can be approximated (the universal approximation theorem). How to find those weights, however, is a separate question.

Coming next This time a person decided the weights. But real networks have tens of thousands to billions of weights (knobs), which no person can set. Part 2 deals with **backpropagation**, which computes the gradient of the loss function efficiently and learns the weights.

Appendix A Writing with matrices and vectors

This appendix uses matrices and vectors (first-year university linear algebra).

Notation Vectors are written in bold lowercase ($\mathbf{x}$, $\mathbf{h}$) and matrices in uppercase (W). Vectors are column vectors. The superscript T denotes the **transpose**, not a power. Transposing swaps the rows and columns of a matrix; attached to a row $(x_1, \dots, x_d)$, it makes a column vector. The transpose A^{T} of a matrix A is the matrix whose (i, j) entry is A_{ji}. $\mathbf{w}^{\mathsf{T}}\mathbf{x} = w_1x_1 + \cdots + w_dx_d$ is the dot product (row vector $\times$ column vector = one number). Numbers written as superscripts **in parentheses**, as in $W^{(1)}$ and $\mathbf{h}^{(2)}$, are **layer numbers**, not powers. When a function acts on a vector, as in $f(\mathbf{z})$, it acts on each component separately ($f(\mathbf{z}) = (f(z_1), \dots, f(z_m))^{\mathsf{T}}$).

A.1 One neuron

With input $\mathbf{x} = (x_1, \dots, x_d)^{\mathsf{T}}$ and weights $\mathbf{w} = (w_1, \dots, w_d)^{\mathsf{T}}$,

$$z = \sum_{j=1}^{d} w_j x_j + b = \mathbf{w}^{\mathsf{T}}\mathbf{x} + b, \qquad y = f(z).$$

The boundary $\mathbf{w}^{\mathsf{T}}\mathbf{x} + b = 0$ is a **hyperplane** in d-dimensional space, and $\mathbf{w}$ is a vector perpendicular to this hyperplane (a normal vector).

A.2 Writing a layer with a matrix

When the hidden layer has m neurons, the equations of Section 3.1,

$$h_k = f\Big(\sum_{j=1}^{d} W_{kj}\, x_j + b_k\Big), \qquad k = 1, \dots, m,$$

can be written together with a matrix as

$$\mathbf{h} = f(W\mathbf{x} + \mathbf{b}).$$

W is an $m \times d$ matrix, and its entry in row k, column j, W_{kj}, is "the weight from input j to neuron k." $\mathbf{b}$ is an m-dimensional vector, and f acts on each component of the vector separately.

A network with L layers stacked is expressed as the repetition of the operation "multiply by a matrix, add a bias, pass through the activation function":

$$\mathbf{h}^{(1)} = f\big(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big), \quad \mathbf{h}^{(2)} = f\big(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\big), \quad \dots, \quad \hat{\mathbf{y}} = g\big(W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}\big)$$

(g is the function of the output layer, treated in Section 6).

The XOR network of Figure 3(a) written with matrices is

$$W^{(1)} = \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} -0.5 \\ 1.5 \end{pmatrix}, \qquad W^{(2)} = \begin{pmatrix} 1 & 1 \end{pmatrix}, \quad b^{(2)} = -1.5.$$

The first row of $W^{(1)}$ holds the weights of OR and the second row those of NAND. In the parameter-count example of Section 3.3, $W^{(1)}$ is a 100×784 matrix and $W^{(2)}$ is a 10×100 matrix.

A.3 Stacking linear layers is the same as one layer

If we remove the activation function ($f(\mathbf{z}) = \mathbf{z}$) and stack two layers, substituting

$$\mathbf{h} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \qquad \hat{\mathbf{y}} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

gives

$$\hat{\mathbf{y}} = W^{(2)}W^{(1)}\mathbf{x} + \left(W^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}\right) = W'\mathbf{x} + \mathbf{b}',$$

which is the same as a one-layer network with the matrix $W' = W^{(2)}W^{(1)}$ and the vector $\mathbf{b}'$. However many layers are stacked, a composition of linear transformations is a linear transformation. The one-input example of Section 4.2 is the case of this formula in which all the matrices are 1×1.

Appendix B The error of broken-line approximation

We explain why, in the broken-line approximation of Section 5.2, the error decreases in inverse proportion to the square of the number of units.

When a function $y(x)$ is approximated on each interval of width h by the segment joining its two ends, it is known that the error is at most

$$|\text{error}| \;\le\; \frac{h^2}{8} M_2,$$

where M_2 is the maximum of the absolute value of the second derivative of the function.[7]

For $y = \sin 2\pi x$, $y'' = -4\pi^2 \sin 2\pi x$, so $M_2 = 4\pi^2$. The interval width is $h = 1/S$, so the upper bound of the error is

$$\frac{1}{8S^2} \times 4\pi^2 = \frac{\pi^2}{2S^2}.$$

For $S = 4$, 8, 16 this is 0.31, 0.077, 0.019, and the actual errors in the table (0.211, 0.070, 0.019) are below these.

> Since the upper bound of the error is proportional to $1/S^2$, doubling the units makes the error 1/4.

The regular-polygon example in the main text has the same structure. The largest gap between a side of the regular n-gon inscribed in a circle of radius 1 and the circle is $1 - \cos(\pi/n)$, which, from $\cos\theta \approx 1 - \theta^2/2$ (for small θ), is about $\pi^2/(2n^2)$: again proportional to $1/n^2$.

[7] Let $\ell(x)$ be the segment joining the ends of the interval $[\alpha, \beta]$. Then $e(x) = y(x) - \ell(x)$ is 0 at both ends. Using Rolle's theorem (a special case of the mean value theorem) repeatedly, there is a ξ in the interval with $e(x) = -\frac{1}{2}y''(\xi)(x-\alpha)(\beta-x)$, and the maximum of $(x-\alpha)(\beta-x)$ is $h^2/4$ at the midpoint, so $|e(x)| \le \frac{h^2}{8}M_2$.

Appendix C Python code (standard library only)

These programs compute the examples in the main text. They use no external libraries, only the standard features of Python 3.

C.1 Logic gates and the XOR network (Sections 1–2)

```
# Make logic gates with a single neuron (step function),
# and compute XOR with two layers
def step(z):
    return 1 if z >= 0 else 0

def neuron(x, w, b):
    z = sum(wi * xi for wi, xi in zip(w, x)) + b
    return step(z)

AND  = lambda x: neuron(x, [1, 1], -1.5)
OR   = lambda x: neuron(x, [1, 1], -0.5)
NAND = lambda x: neuron(x, [-1, -1], 1.5)

def XOR(x):
    h = [OR(x), NAND(x)]          # hidden layer
    return AND(h), h              # output layer

print(" x1 x2 | AND OR NAND | h1 h2 | XOR")
for x in [(0, 0), (0, 1), (1, 0), (1, 1)]:
    y, h = XOR(x)
    print(f"  {x[0]}  {x[1]} |  {AND(x)}   {OR(x)}   {NAND(x)}   |  {h[0]}  {h[1]} |  {y}")
```

Output:

```
 x1 x2 | AND OR NAND | h1 h2 | XOR
  0  0 |  0   0   1   |  0  1 |  0
  0  1 |  0   1   1   |  1  1 |  1
  1  0 |  0   1   1   |  1  1 |  1
  1  1 |  1   1   0   |  1  0 |  0
```

C.2 Function approximation by adding ReLUs (Section 5)

```
# Approximate sin(2 pi x) with a broken line made by adding ReLUs
import math

def relu(z):
    return max(0.0, z)

def build(S):
    """Divide [0,1] into S equal parts and find the weights of the broken-line approximation"""
    knots = [k / S for k in range(S + 1)]
    ys = [math.sin(2 * math.pi * t) for t in knots]
    slopes = [(ys[k + 1] - ys[k]) * S for k in range(S)]
    # hidden unit k: relu(x - knots[k]); weight to the output: change of slope
    a = [slopes[0]] + [slopes[k] - slopes[k - 1] for k in range(1, S)]
    return knots[:S], a, ys[0]

def net(x, c, a, b0):
    return b0 + sum(ak * relu(x - ck) for ck, ak in zip(c, a))

for S in (4, 8, 16):
    c, a, b0 = build(S)
```

```
    err = max(abs(net(i / 10000, c, a, b0) - math.sin(2 * math.pi * i / 10000))
              for i in range(10001))
    print(f"{S:2d} hidden units: max error = {err:.4f}")
```

Output:

```
 4 hidden units: max error = 0.2105
 8 hidden units: max error = 0.0704
16 hidden units: max error = 0.0188
```