Scientific Machine Learning 01
Perceptrons, Sigmoid Neurons, and Artificial Neural Networks

Donghyun Ko

January 3, 2026

A *perceptron* is a hard on/off switch with a linear decision boundary, a *sigmoid neuron* is that switch with a smooth dimmer knob (so we can learn with calculus), and an *ANN* stacks these neurons in layers to approximate very complex functions.

Contents

1 Why this post? What you'll learn **1**

2 Perceptron **2**

3 Sigmoid neuron: make it smooth, make it learn! **4**

4 Artificial vs. Biological neurons: a tasteful analogy **5**

5 ANNs: Artificial Neural Network **6**

1 Why this post? What you'll learn

Today we connect three big ideas that show up everywhere in machine learning:

- What exactly is a perceptron and how does it make decisions?
- Why do sigmoid activations make learning smooth (literally) and effective?
- How do we assemble neurons into multilayer networks (ANNs) that scale to real tasks?

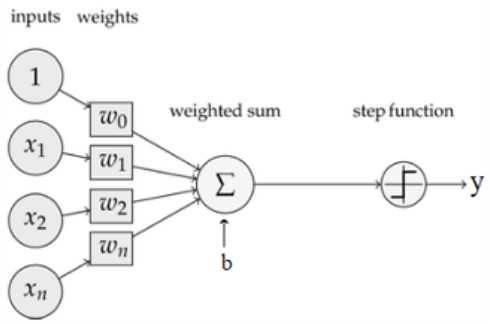
Learning goals — Build intuition for (i) linear decision boundaries, (ii) smooth activations and gradient-based learning, and (iii) the layer-by-layer view of ANNs with clean notation you can reuse in code.

2 Perceptron

The perceptron takes an input vector $x = [1, x_1, \ldots, x_n]$ (bias folded in as the leading 1), computes a weighted sum $z = w^\top x$, adds a bias b, and then applies a *step* activation.

A perceptron consists of the following parts:

1. **Input**: $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$, which are binary values[2], $x_i \in \{0, 1\}$
2. **Weights**: $\mathbf{w} = [w_0, w_1, \ldots, w_n]$, real numbers for the relative importance of each input
3. **Bias**: b, also called a "threshold" value
4. **Activation function**: f, the step function
5. **Output**: y, a single binary value, $y \in \{0, 1\}$



The perceptron makes a decision based on the comparison between the weighted sum after the activation function and the threshold('bias') such that:

$$y = f(w^\top x + b) = \begin{cases} 1, & w^\top x + b > 0, \\ 0, & \text{otherwise.} \end{cases}$$

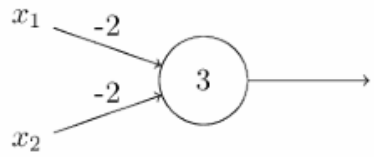
Geometrically, $w^\top x + b = 0$ is a hyperplane that splits space into "predict 1" and "predict 0." It's simple, fast, and perfect when your data are *linearly separable*. Bias, or threshold, is a measure of how easy it is to get the perceptron to output 1. It allows the perceptron to shift the decision boundary left or right, and this is why Perceptron is a linear binary classifier used in supervised learning. By varying the weights and the bias, we can get different models of decision-making. A single perceptron can only make simple decisions, whereas a complex network (with multiple layers) of perceptrons could make quite subtle decisions.

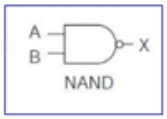
Bias is the game-changer. Without b, the decision boundary must pass through the origin. With b, you can *translate* the boundary and fit more realistic data.

Logic gates with perceptrons

Perceptrons can be used to compute the elementary logical functions that we usually think of as underlying computation, functions such as AND, OR, and NAND. For example, the following perceptron can compute the NAND logical gate, which is universal for logical computations.

x_1	x_2	$\mathbf{w}^\mathsf{T}\mathbf{x} + b$	output
0	0	$(-2) \times 0 + (-2) \times 0 + 3 = 3$	1
1	0	$(-2) \times 1 + (-2) \times 0 + 3 = 1$	1
0	1	$(-2) \times 0 + (-2) \times 1 + 3 = 1$	1
1	1	$(-2) \times 1 + (-2) \times 1 + 3 = -1$	0





NAND gate

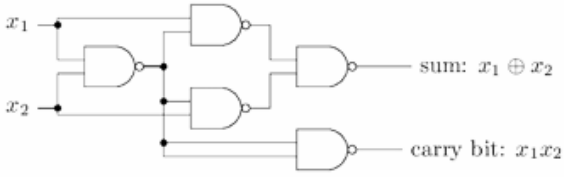
Input A	Input B	Output
0	0	1
1	0	1
0	1	1
1	1	0

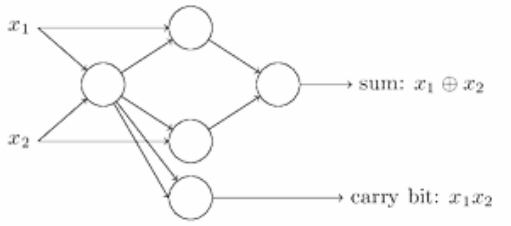
NAND gate is universal for logical computation, that is, we can build any logical computation out of NAND gates. Because a perceptron with the previous weights/bias implements the NAND gate, we can use networks of such a perceptron to compute any logical function. Furthermore, even more complex computations can be implemented by the perceptron, but this time with the multiple layer of perceptrons like MLP (Multi-Layer-Perceptron). Let's start with a bite-size digital task: **add two bits** $x_1, x_2 \in \{0, 1\}$. A half-adder needs:

- *Sum* bit $x_1 \oplus x_2$ (XOR), and
- *Carry* bit, which is 1 only when both inputs are 1 (i.e., $x_1 \cdot x_2$)

You can wire this up entirely with **NAND** gates because NAND is *universal*, meaning any logic can be built from it. One simple choice is two inputs with weights $w_1 = w_2 = -2$ and bias $b = 3$. Using the step activation $f(z) = \mathbb{I}[z > 0]$, the neuron outputs 1 except when both inputs are 1, which is exactly NAND.

x_1	x_2	$x_1 \oplus x_2$	$x_1 \cdot x_2$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1



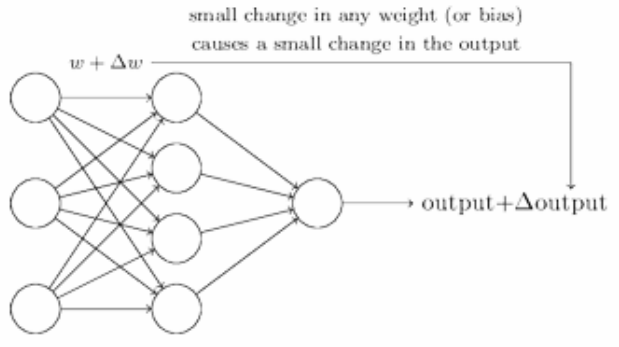


But to have this representation power, we should choose the right combination of (w, b) with the hands. However, the representation power is not the same as *learnability from data*. Because the step function jumps from 0 to 1, tiny weight changes can flip output entirely, which makes gradient-based learning awkward (the gradient is zero almost everywhere and undefined at the jump).

For machine learning we need more. A network of step-output perceptrons is basically a programmable logic circuit—great for hand-designed gates, not so great for *learning from data*. Tiny weight tweaks can flip outputs from 0 to 1 in a discontinuous way, making gradient-based training brittle. That's why we soon switch to *smooth* units (e.g., sigmoid) and differentiable training rules.

3 Sigmoid neuron: make it smooth, make it learn!

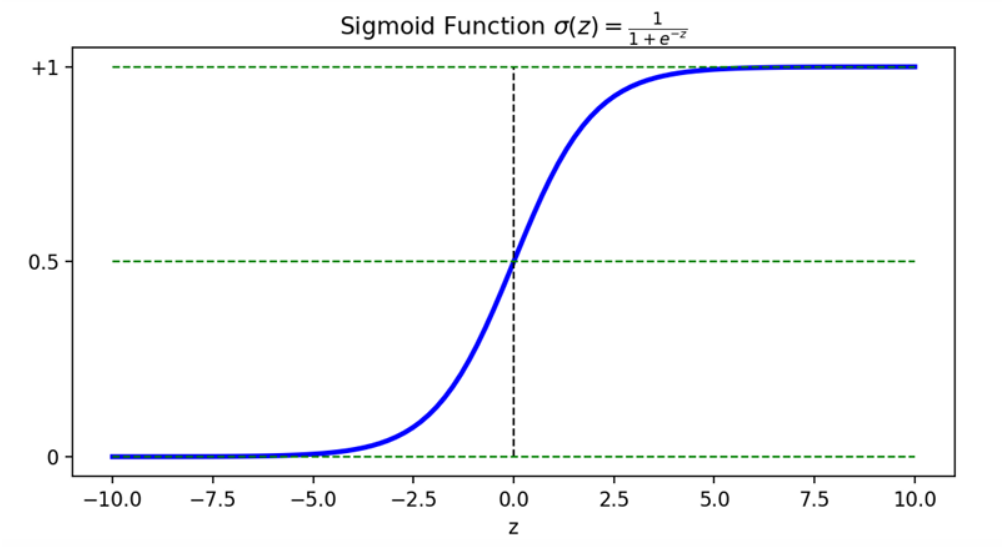
To build learning algorithms that automatically tune a network's weights and biases, we need *smoothness*: a small change in some weights (or bias) should cause only tiny, predictable changes in the output from the network. When this continuity holds, we can use gradients and the chain rule via backpropagation to make small, targeted updates that steadily steer behavior toward our goals. As we discussed in the previous session, with hard step activations, an infinitesimal change can flip an output from 0 to 1 and derail training, so choosing differentiable activations (e.g., sigmoid, tanh, or ReLU almost everywhere) is what makes "the network would be learning" true in practice.



A sigmoid (or logistic) neuron is very similar to a perceptron, but instead of a hard step that only outputs 0 or 1, it uses a smooth activation function that can return any value between 0 and 1. So, small changes in the weights or bias cause only small changes in the output. Another good property, which we will discuss with more details later, is that its derivative $\sigma'(z) = \sigma(z)\Big(1 - \sigma(z)\Big)$ is well-behaved, and thus we can apply gradient-based learning while also reading the output as a probability-like confidence. The sigmoid function(also called 'logistic function') is defined as

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

where $z = w^\top x + b$ is the weighted sum of inputs. If z is a large positive number, then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$; if z is a large negative number, then e^{-z} grows very large and $\sigma(z) \approx 0$; for moderate values of z, the output lies smoothly in between, giving a 'softened step'.



Because the output $y = \sigma(w^\top x + b)$ changes gradually in the moderate range, small parameter adjustments $(\Delta w, \Delta b)$ lead to small output changes such that:

$$\Delta y \approx \sum_j \frac{\partial y}{\partial w_j} \Delta w_j + \frac{\partial y}{\partial b} \Delta b$$

This linear approximation is what enables learning with gradient descent. Even better, the derivative of the sigmoid is very simple:

$$\sigma'(z) = \frac{d\sigma(z)}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \sigma(z) - \sigma^2(z) = \sigma(z)\Big(1-\sigma(z)\Big)$$

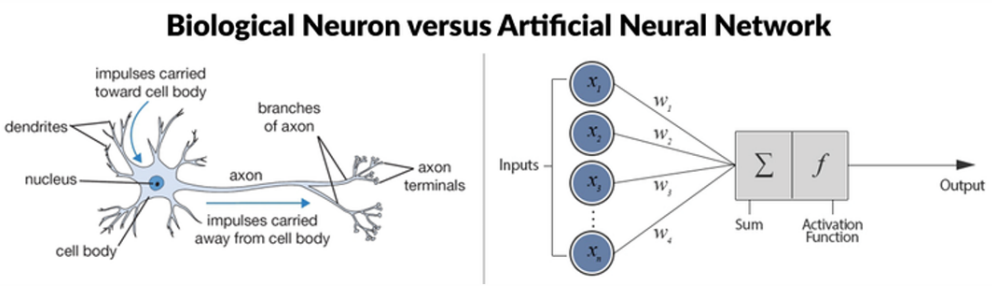
Thanks to this smoothness and easy derivative, sigmoid neurons behave like perceptrons in extreme cases, but unlike perceptrons, they allow us to train networks effectively using calculus.

Aside: other activations — Sigmoid is classic and probabilistic-friendly; modern nets often use ReLU/tanh/variants for better gradient flow. The big idea is the same: *differentiable nonlinearity* enables learning.

4 Artificial vs. Biological neurons: a tasteful analogy

The human brain is an astonishingly complex system, hosting billions of neurons that constantly process and transmit chemical and electrical signals. Each neuron has dendrites, which act like branches that receive information from other neurons; this information is then processed in the cell nucleus, also called the soma. If the processed signal is strong enough, it travels down the axon, a long cable-like structure, to other neurons. The actual transmission of information occurs in the synapse, which is the connection point between the axon of one neuron and the dendrites of another neuron. This intricate web of dendrites, soma, axons, and synapses is what powers human thought, memory, and perception.

Artificial intelligence researchers, inspired by this biological model, created simplified versions known as artificial neurons. Back in 1943, Warren McCulloch and Walter Pitts introduced what is now called the McCulloch-Pitts (MCP) neuron. They treated neurons as simple logic gates with binary output: Multiple signals enter through the dendrites, are integrated in the nucleus, and if they exceed a threshold, the axon "fires" a signal forward. The artificial neuron mirrors this idea mathematically: it receives several inputs, applies individual weights to each, sums them up, and passes the result through a nonlinear activation function to produce an output. In short, dendrites become inputs, synapses become weights, the soma becomes the processing node, and the axon is the output channel.



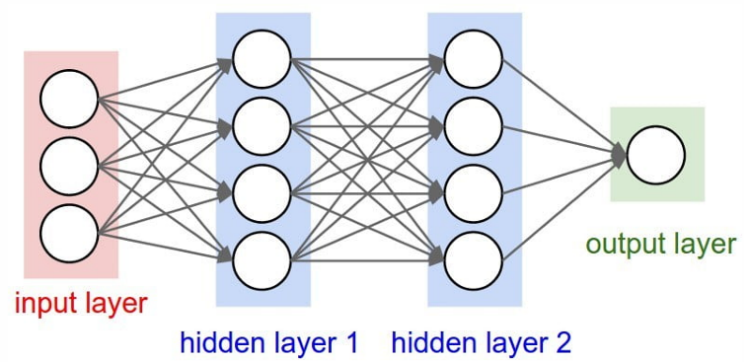
Artificial neuron	Biological neuron
Node	Soma (cell body)
Inputs	Dendrites
Weights	Synaptic strengths
Output	Axon signal

Although artificial neurons borrow heavily from biological terminology, the two worlds are vastly different in scale. A human brain contains about 86 billion neurons and an estimated 100 trillion synapses. By comparison, even the largest artificial neural networks are tiny. In fact, as of 2018, the largest ANNs built on supercomputers were comparable only to the size of a frog's brain. Progress has been rapid, though: In 2020, OpenAI unveiled GPT-3 with 175 billion parameters; By 2023, GPT-4 grew to 1.7 trillion parameters, and GPT-4o (released in 2024) expanded slightly further to 1.8 trillion. Although OpenAI has not publicly disclosed the number of parameters for GPT-5 (as of August 2025), it is clear that artificial networks, although still far less complex than the human brain, are scaling at an astonishing pace. This parallel between natural and artificial neurons is less about replication and more about inspiration: biological neurons offer a metaphor that guides the design of artificial ones, even as technology charts its own trajectory.

5 ANNs: Artificial Neural Network

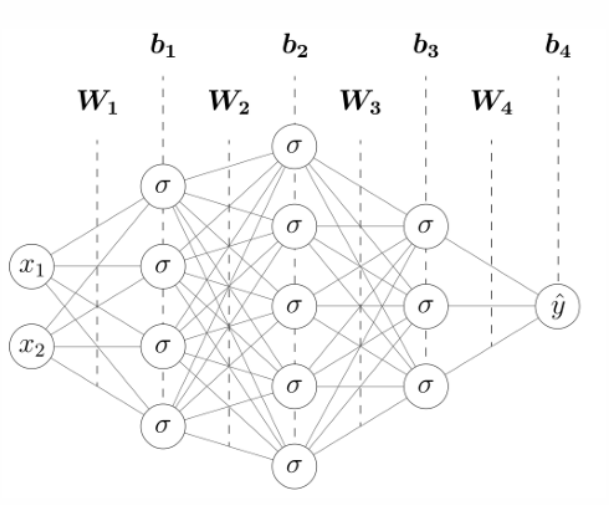
Artificial Neural Networks (ANNs) are built from the artificial neurons that we have already introduced: perceptrons that use a step function and sigmoid neurons that use smooth sigmoid activation. The structure of an ANN follows a layered design: the leftmost layer is *input layer*, containing the input neurons that simply pass features into the network; the rightmost layer is *output layer*, containing the output neurons that produce predictions; and

the layers between are called *hidden layers*, where their neurons are neither input nor output. The *depth* of a network typically means the number of hidden layers plus one. Historically, networks with multiple layers are often called *Multi-Layer Perceptrons (MLPs)*, although their neurons are usually sigmoids, not strict perceptrons.



In the ANNs, the outputs of one layer become the inputs to the next, which is why they are also known as *Feedforward Neural Networks (FNNs)* - information always flows forward without loops. In contrast, *Recurrent Neural Networks (RNNs)*, which we will discuss later, allows feedback loops, enabling temporal or sequential processing. In practice, MLP, ANN, and FNN are often used interchangeably, and when a network has two or more hidden layers, it is called a *Deep Neural Network (DNN)*. However, it is worth noting that even a shallow ANN, if properly trained, can be surprisingly powerful. Mathematically, we can express a regular feedforward ANN as

$$\hat{y}(x) = \sigma\Big(\cdots \sigma\Big(\sigma(xW_1 + b_1)W_2 + b_2\Big) \cdots\Big)W_L + b_L,$$



where L is the depth of the network, σ is the activation function, W_ℓ and b_ℓ are the weights and biases of the ℓ-th layer, and $\hat{y}(x)$ is the network output for input x. This compact formula captures the essence of how information flows forward through layers, each step applying a weighted linear transformation followed by a nonlinear activation, culminating in the final prediction.

Universal approximation—intuition only. Even a shallow network with enough hidden units can approximate many functions arbitrarily well. Depth often buys you efficiency, better inductive bias, and cleaner feature hierarchies.

This is a posting that I summarized with study-purpose and is adapted from lecture notes of NE-795(Scientific Machine Learning), given by professor Xu Wu, NC State University.