# Learning Deep Learning Chapter 1a

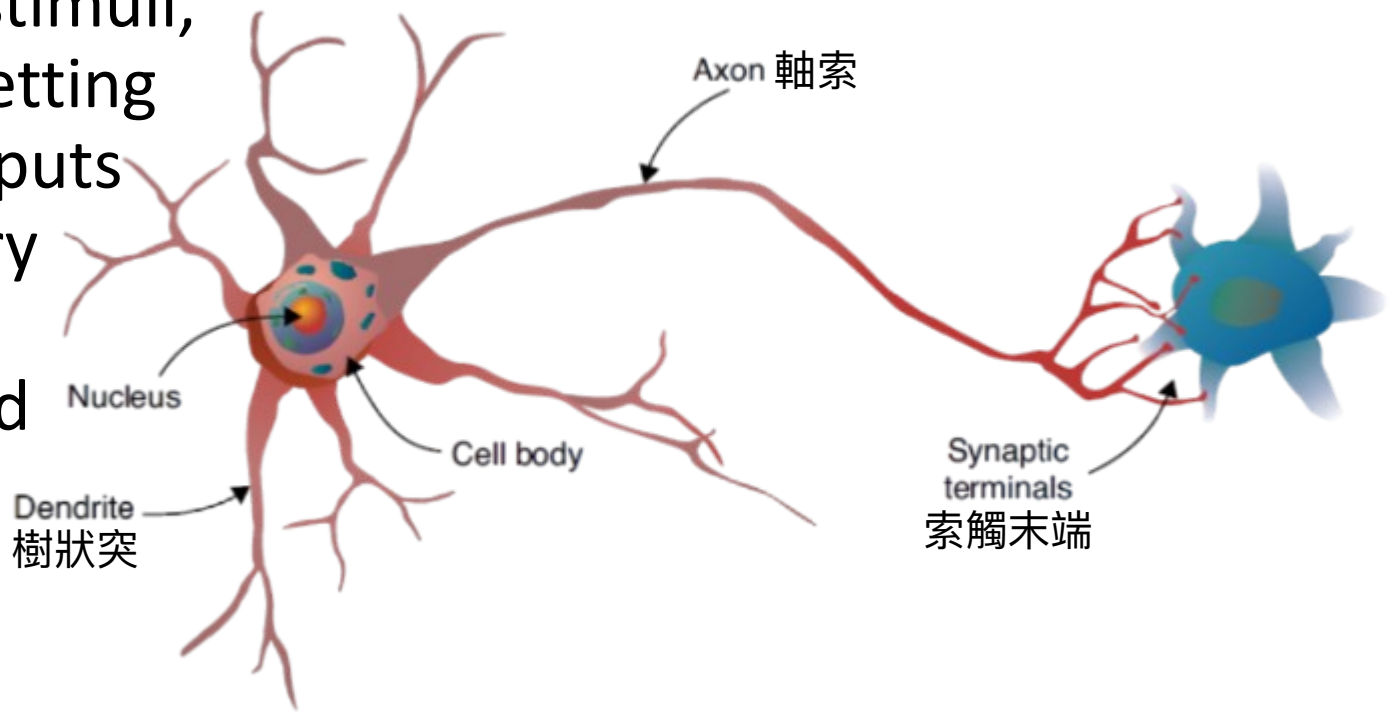
Chao-En Yu

National Tsing Hua University

# Biological Neuron

- The perceptron is an artificial neuron, i.e., a model of a biological neuron.

- The neuron receives stimuli on the dendrites, and in cases of sufficient stimuli, the neuron fires (getting activated), and outputs excitatory/inhibitory signals on its axon, that are transmitted to other neurons.

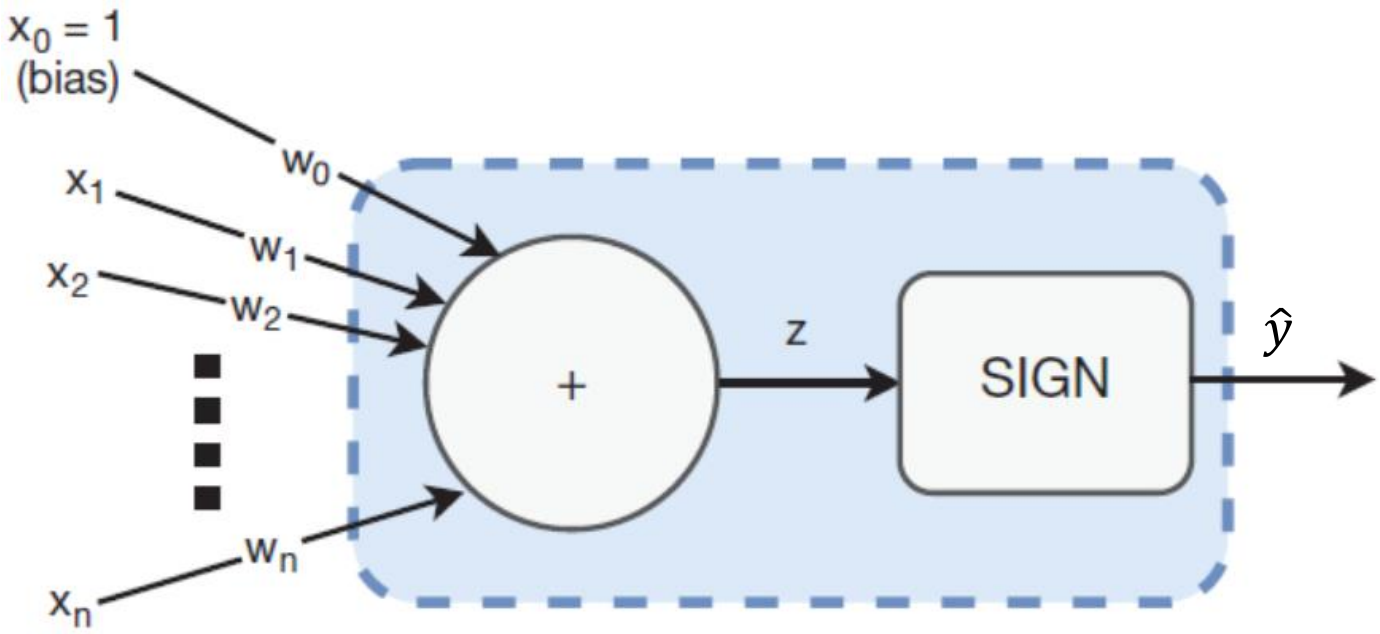


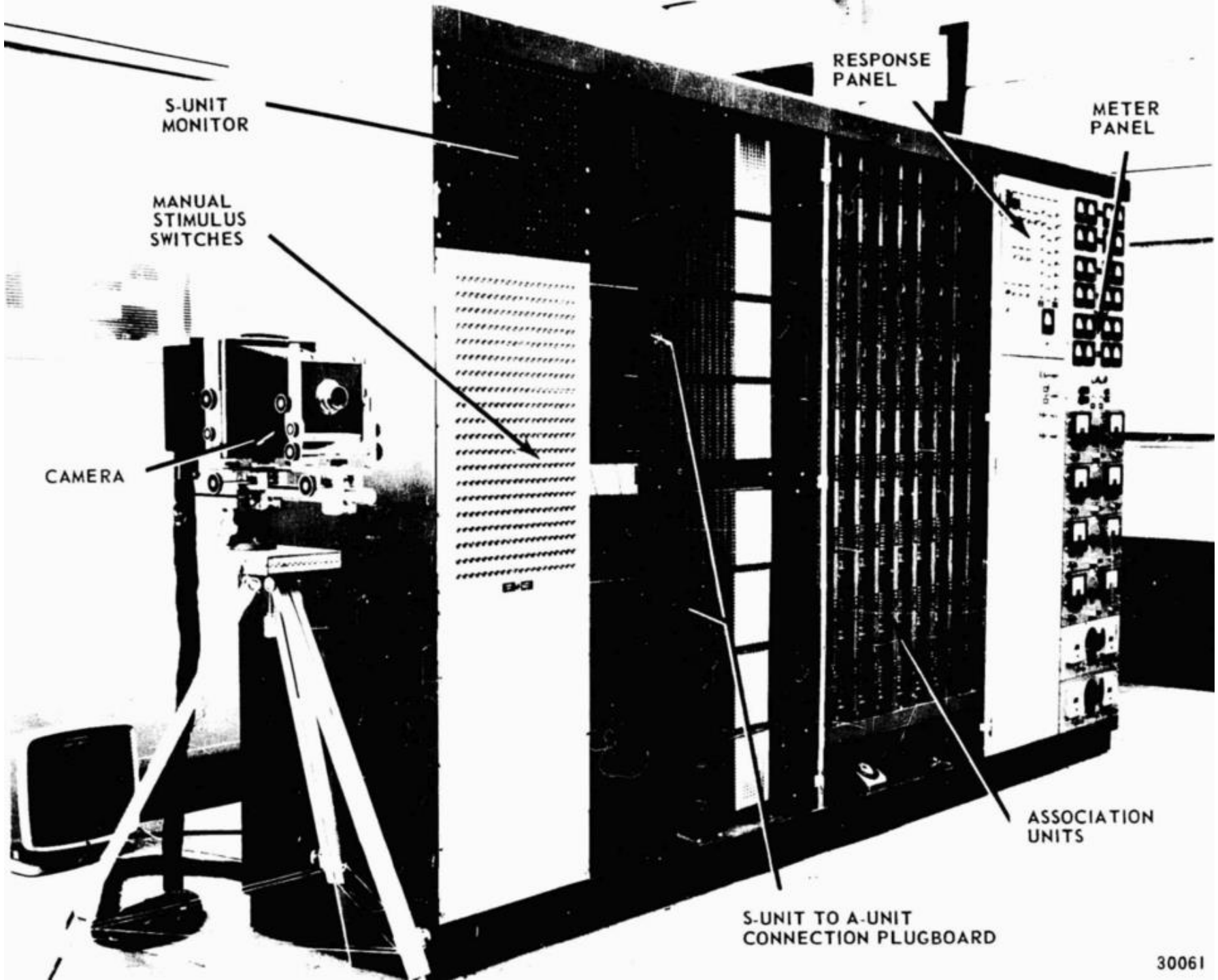
# Rosenblatt Perceptron

- The inputs are typically named $x_0, x_1, \ldots, x_n$ ($x_0 = 1$ being the bias input), each input having an associated weight ($w_i$), and the output is named $y$.

- The inputs and output loosely correspond to the dendrites and the axon.

- For the perceptron, the output can take on only −1 or 1.






Mark I Perceptron machine built in 1958 is the first implementation of the perceptron algorithm.
It was connected to a camera with 20×20 photocells to take a 400-pixel image.

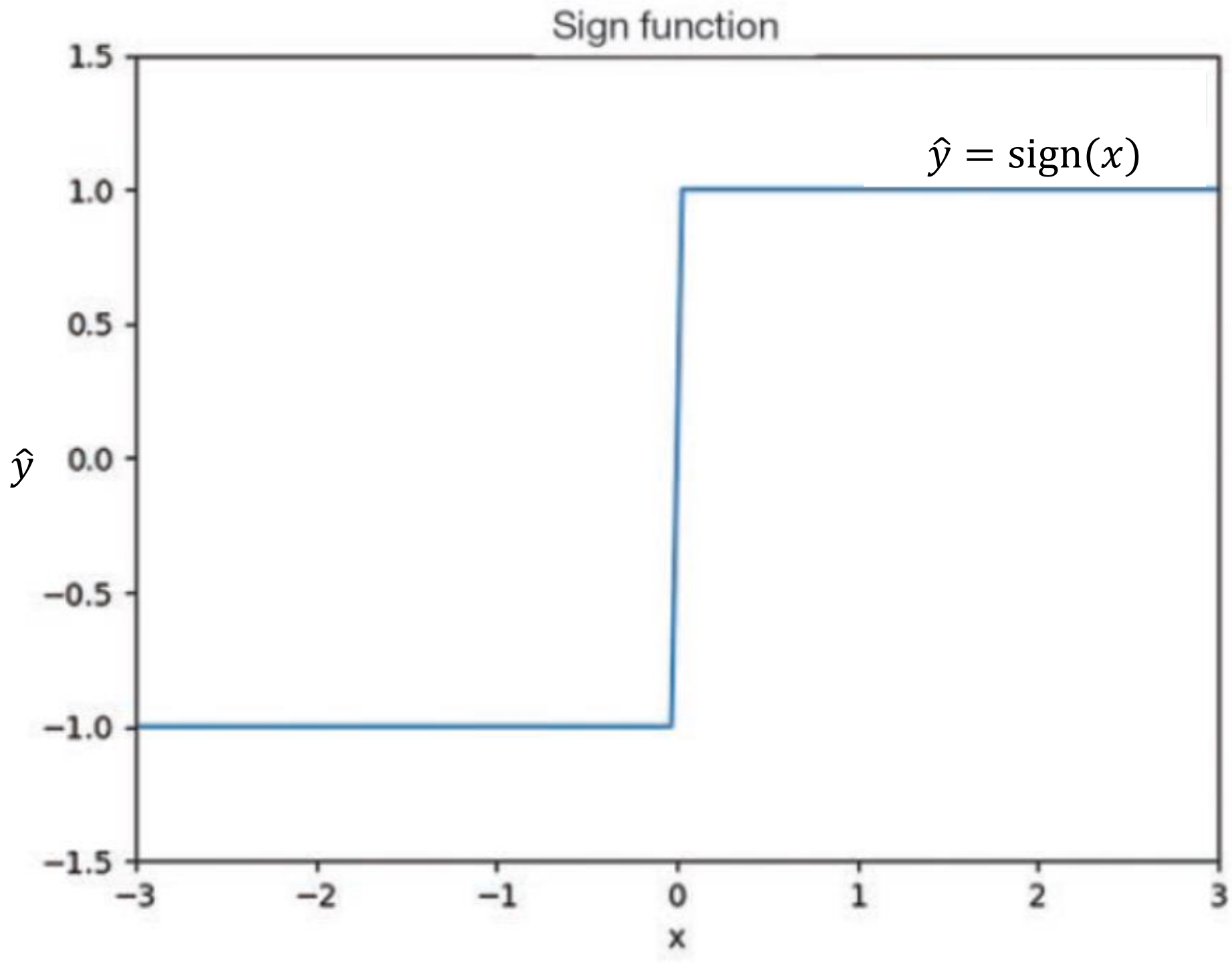
# Activation Function

- Each input value is multiplied by its corresponding weight before it is presented to the computational unit (the dashed rectangle).

- The computational unit computes the sum of the weighted inputs (denoted by $z$), and then applies a activation function,$\hat{y} = f(z)$.

$z$ is called intermediate representation.

- The activation function for a perceptron is the sign function (signum function), which evaluates to 1 if the input is 0 or higher and −1 otherwise.


Sign function
$\hat{y} = \text{sign}(x)$
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
$\hat{y}$
−3
−2
−1
0
1
2
3
x

# Mathematical Perceptron

- The perceptron uses the sign function as an activation function:

$$\hat{y} = f(z), \text{where } z = \sum_{i=0}^{n} w_i x_i \,,$$

$$f(z) = \begin{cases} -1, z < 0 \\ 1, z > 0 \end{cases}$$

$$x_0 = 1 \text{ (bias term)}.$$

Varying the bias weight is equivalent to adjusting the threshold later.

- Other artificial neurons may use other activation functions.

# Programming Perceptron

- The following code implements this function programmatically in Python:

```
z = 0.0; w=[0.9, -0.6, -0.5]; x=[1,-1,-1];
for i in range(len(w)):
    z = z + x[i] * w[i]  # sum of weighted inputs
if z < 0: # sign function
    y_hat = -1
else:
    y _hat = 1
print(y_hat)
```

← Any error here?

# Two-Input Perceptron

- Let us study a perceptron with two inputs in addition to the bias input. Without any justification, we set $w_0 = 0.9$, $w_1 = -0.6$, and $w_2 = -0.5$.

- Let us see how this perceptron behaves for all input combinations assuming that each of the two inputs can take on only the values −1.0 and 1.0.

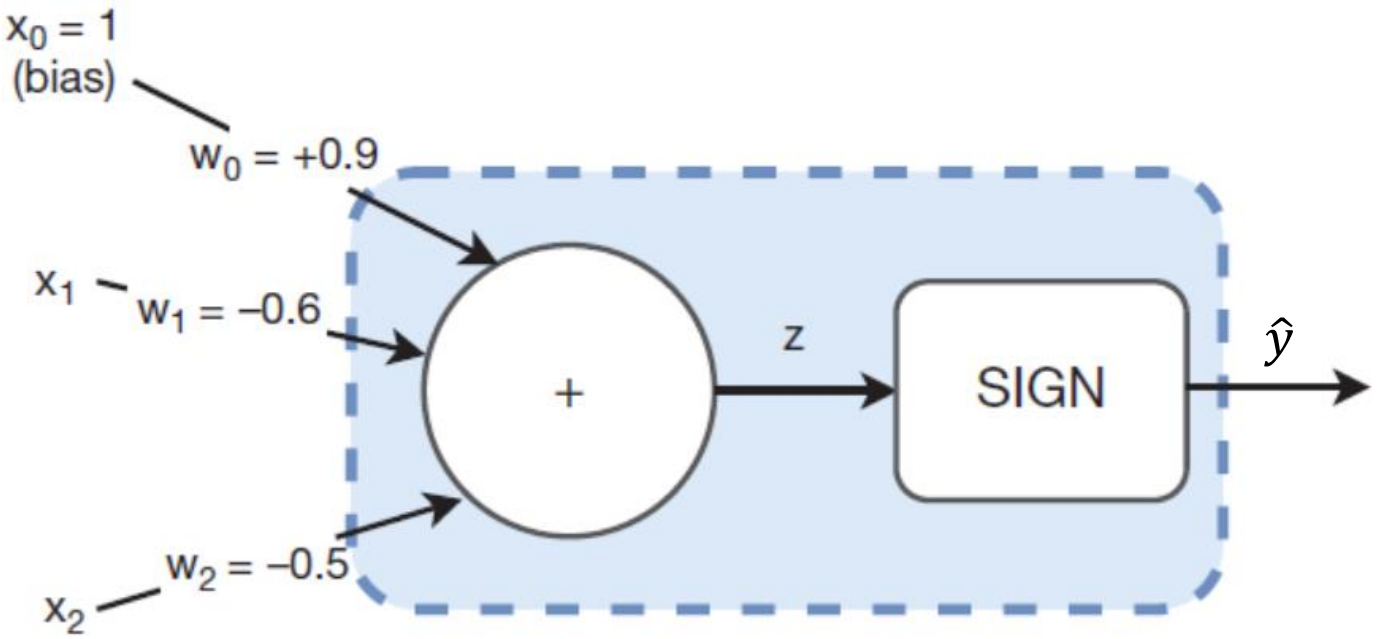


*Table 1-1* Behavior of a Perceptron with Two Inputs **(NAND Gate)**

| $X_0$ | $X_1$ | $X_2$ | $W_0*X_0$ | $W_1*X_1$ | $W_2*X_2$ | $Z$ | $\hat{Y}$ |
|---|---|---|---|---|---|---|---|
| 1 | –1 (False) | –1 (False) | 0.9 | 0.6 | 0.5 | 2.0 | 1 (True) |
| 1 | 1 (True) | –1 (False) | 0.9 | –0.6 | 0.5 | 0.8 | 1 (True) |
| 1 | –1 (False) | 1 (True) | 0.9 | 0.6 | –0.5 | 1.0 | 1 (True) |
| 1 | 1 (True) | 1 (True) | 0.9 | –0.6 | –0.5 | –0.2 | –1 (False) |

# Perceptron Learning Algorithm

- The chosen weights are *not* the only ones that result in a perceptron which behaves like a NAND gate.
- Whether there is a general approach for determining the weights is where the perceptron learning algorithm comes into play.
- The perceptron learning algorithm is what is called a *supervised learning algorithm*: the model is trained with both the input and the output data (ground truth).

A step-by-step procedure for solving a problem. →

# Procedure

- The perceptron learning algorithm works as follows:

  1. Randomly initialize the weights.

  2. Select one input/output pair at random.

  3. Present the values $x_0, x_1, \ldots, x_n$ to the perceptron to compute the output $y$.

  4. If the output $\hat{y}$ is different from the ground truth for this input/output pair, adjust the weights:
     a. If $\hat{y} < 0$, add $\eta x_i$ to each $w_i$.
     b. If $\hat{y} > 0$, subtract $\eta x_i$ from each $w_i$.

  5. Repeat steps 2, 3, and 4 until the perceptron predicts all examples correctly.

# Limitations

- For some sets of input/output pairs, the algorithm will not converge.
- If there exists a set of weights that enables the perceptron to represent the set of input/output pairs, then the algorithm is guaranteed to converge.
- The constant $\eta$ is the learning rate which is an example of a *hyperparameter*: It is not a parameter that is adjusted by the learning algorithm but can still be adjusted.

# Initialization

- To implement the algorithm, we first initialize variables for the weights and training examples.

```
import numpy as np
np.random.seed(7) # To make repeatable
LEARNING_RATE = 0.1
index_list = [0, 1, 2, 3] # Used to randomize order
x_train = [[1.0, -1.0, -1.0], [1.0, -1.0, 1.0],[1.0, 1.0, -
1.0], [1.0, 1.0, 1.0]] # 4 Inputs
y_train = [1.0, 1.0, 1.0, -1.0] # Output (ground truth)
w = [0.2, -0.6, 0.25] # Initialized weights
print(w)
```

← Positive constant

# Code with a Nested Loop

- The inner loop runs through all four *randomly-ordered* training examples.
- The value of true $y$ will either be −1 or +1, and consequently results in selecting between addition and subtraction for the update.
- The outer loop tests whether the perceptron provided correct output for all four examples and, if so, terminates the program.

# Perceptron Training Loop

- For simplicity, suppose that we know that the adjusted weights will converge within 10 loops. Code Snippet 1-4 can be trimmed as follows.

```
for epoch in range(10):
    np.random.shuffle(index_list) # Randomize order
    for i in index_list:
        x = x_train[i]
        y = y_train[i]
        z = 0.0
        for i in range(len(w)):
            z = z + x[i] * w[i] # sum of weighted inputs
```

# Perceptron Training Loop

- (Continuing)

```
if z < 0: # Apply sign function
    y_hat = -1
else:
    y_hat = 1
if y != y_hat: # Update weights when wrong
    for j in range(len(w)):
        w[j] = w[j] + (y * LEARNING_RATE * x[j])
    print(np.round(w, 2))
```

- Q: What is the smallest number of epochs to have the same results?

# Weight Adjustments

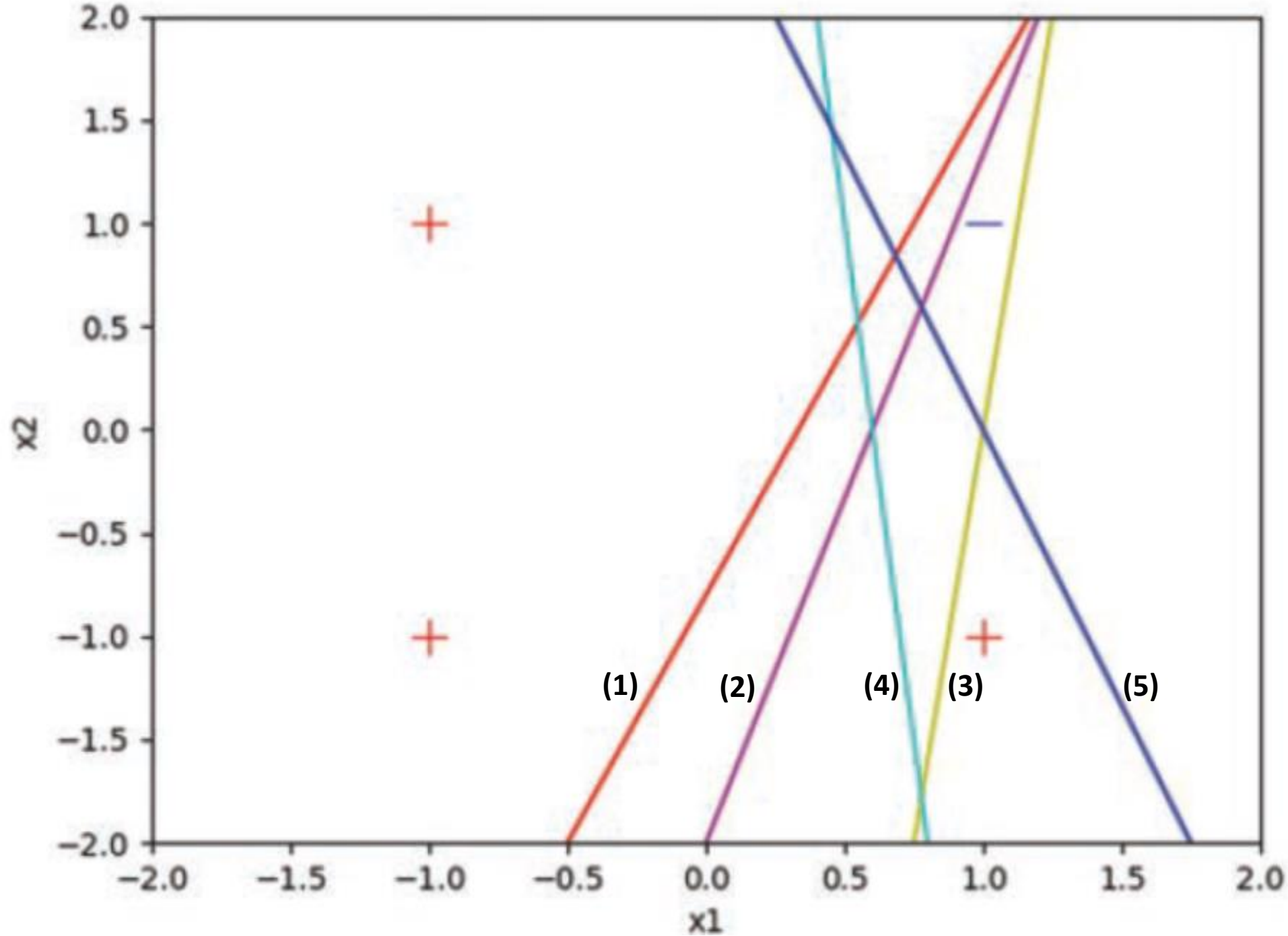
- The weights are gradually adjusted from the initial values to arrive at weights that produce the correct output.

```
w0 = 0.20 , w1 = -0.60 , w2 =  0.25
w0 = 0.30 , w1 = -0.50 , w2 =  0.15
w0 = 0.40 , w1 = -0.40 , w2 =  0.05
w0 = 0.30 , w1 = -0.50 , w2 = -0.05
w0 = 0.40 , w1 = -0.40 , w2 = -0.15
```

Weights are adjusted by different amounts

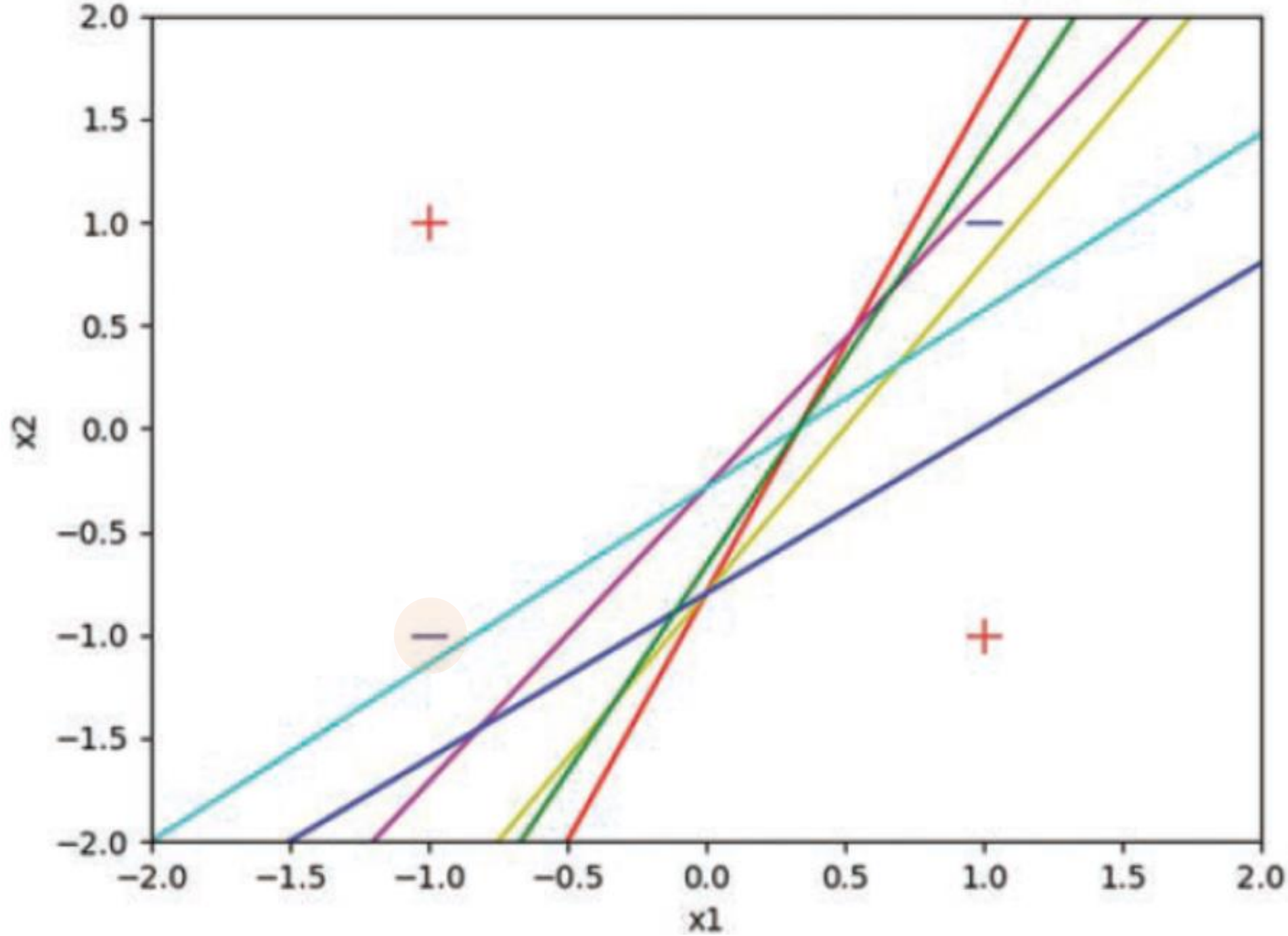
# 2D Coordinate Illustration

- The perceptron divides the 2D space into two regions, separated by a straight line.
- All inputs on one side of the line produce the output -1, and all inputs on the other side of the line produce the output +1.
- Learning process progressing in the following order: red, magenta, yellow, cyan, blue.


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
x2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x1
(1)
(2)
(4)
(3)
(5)

# Limitations of the Perceptron

- What happens if a straight line cannot correctly classify all the data points?
- The following figure shows four data points on the same type of chart that is trivial to solve the problem with a curved line.
- But it is not possible to solve the problem with a straight line. This is one of the key limitations of the perceptron.


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
x2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x1
+
−
−
+

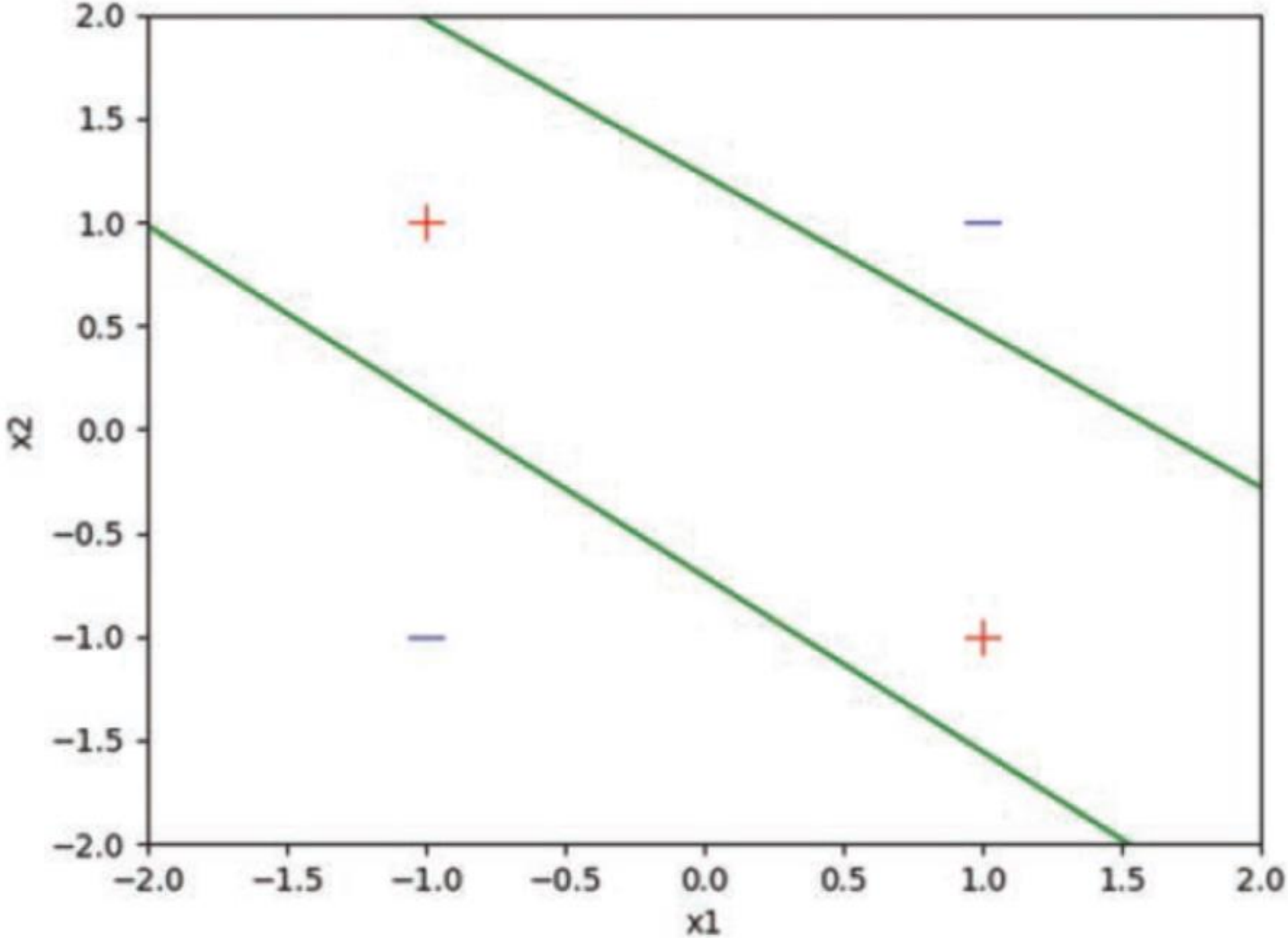
# The XOR Problem

- The XOR (exclusive OR) problem shows that a *single-layer* neural network cannot separate data when inputs are not on opposite sides of a straight line.

| $X_0$ | $X_1$ | $Y$ |
|---|---|---|
| False | False | False |
| True | False | True |
| False | True | True |
| True | True | False |

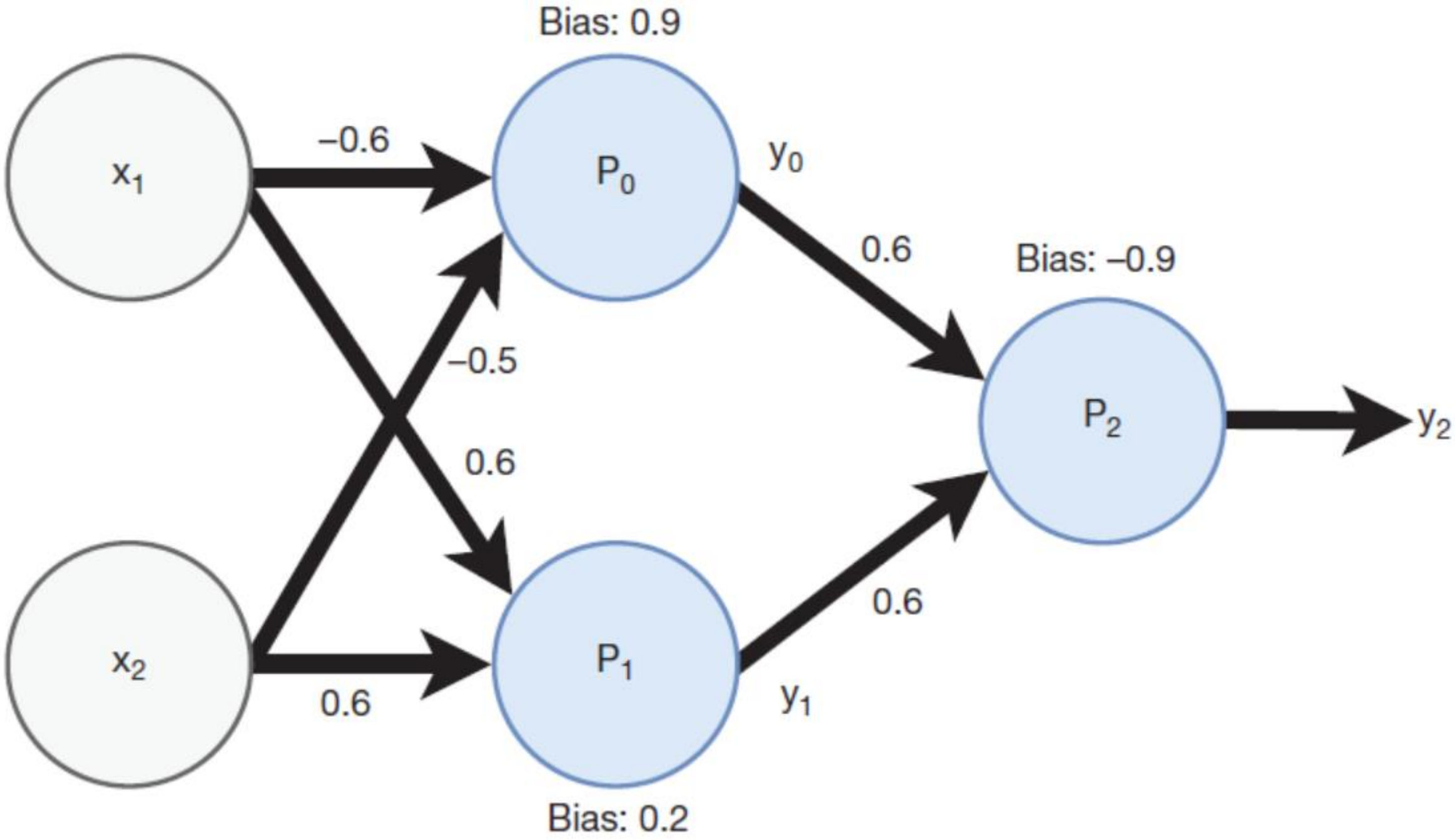
# Combining Perceptrons

- If we add another perceptron, we can draw another straight line.
- Each of the two perceptrons will fire correctly for three out of four data points. The data points with $y = 1$ are between the two lines.
- If we could combine the output of the two, and output 1 only when both of them compute the output as a 1, then we would get the right result.


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
x2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x1

# Two-Layer Neural Network

- The two-layer neural network is one of the simplest examples of a fully connected feedforward network.
- Fully connected means that the output of each neuron in one layer is connected to all neurons in the next layer.
- Feedforward means that there are no backward connections, or, using graph terminology, it is a directed acyclic graph (DAG).




Input layer Hidden layer Output layer

# Justification of Bias Term

- Recall the activation function:

$$f(z) = \begin{cases} -1, z < 0 \\ 1, z > 0 \end{cases}$$

where the threshold of 0 has been chosen arbitrarily.

- To make our perceptron more general, the threshold is replaced by a parameter $\theta$:

$$f(z) = \begin{cases} -1, z < \theta \\ 1, z > \theta \end{cases}$$

# Adjustable Threshold Value

- For the output to take on the value 1, we have

$$\mathrm{z} \geq \theta,$$

  which can be rewritten as

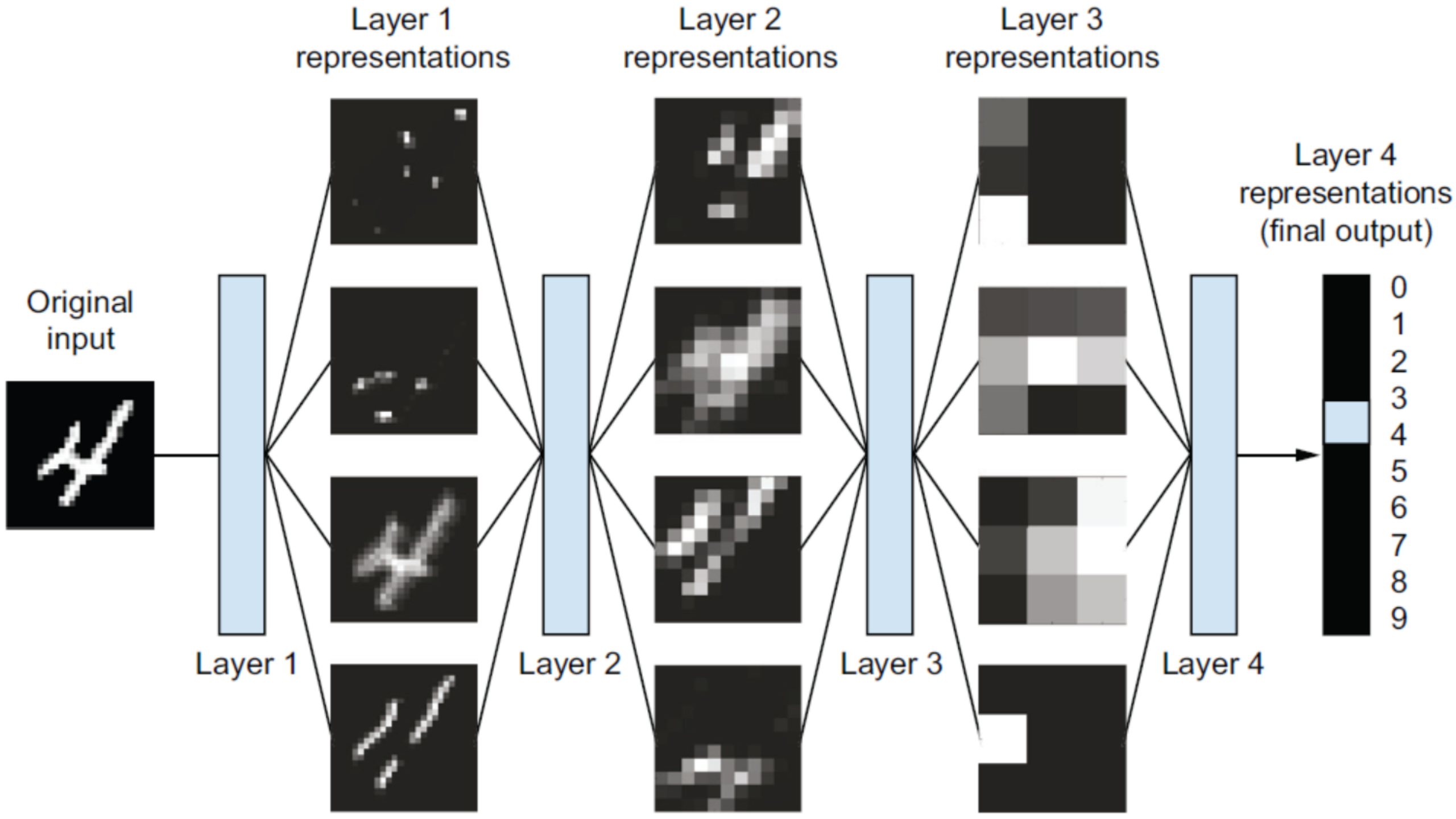
$$\mathrm{z} - \theta \geq 0.$$

- If we want the perceptron to use a threshold of $\theta$, then we set $w_0$ to be $-\theta$ (to move the plane *down*).

- The rationale for the bias term is that our perceptron can be made to implement any arbitrary threshold $\theta$.

# Concept of Layers

- A multilayer neural network has an input layer, one or more hidden layers, and an output layer.
- The input layer does not contain neurons and can take any real numbers as inputs. The output layer can consist of more than one neuron.
- The network has only a single hidden layer (with two neurons), but a deep neural network (DNN) has more than one hidden layer and typically many more neurons in each layer.


Layer 1 representations
Layer 2 representations
Layer 3 representations
Layer 4 representations (final output)
Original input
Layer 1
Layer 2
Layer 3
Layer 4
0
1
2
3
4
5
6
7
8
9