Chapter 06: Deep Learning

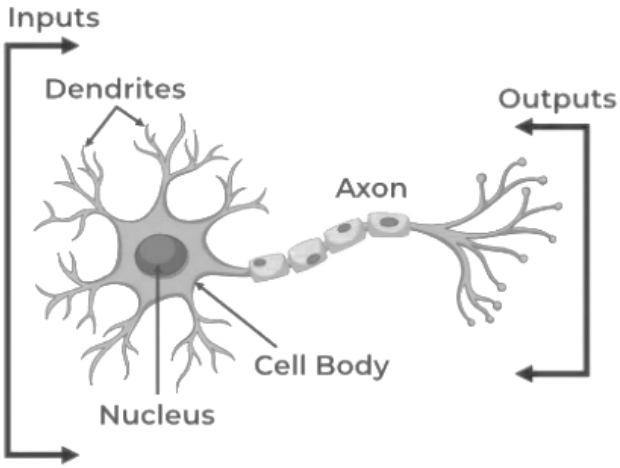
🧠 Biological Neuron vs 🤖 Artificial Perceptron

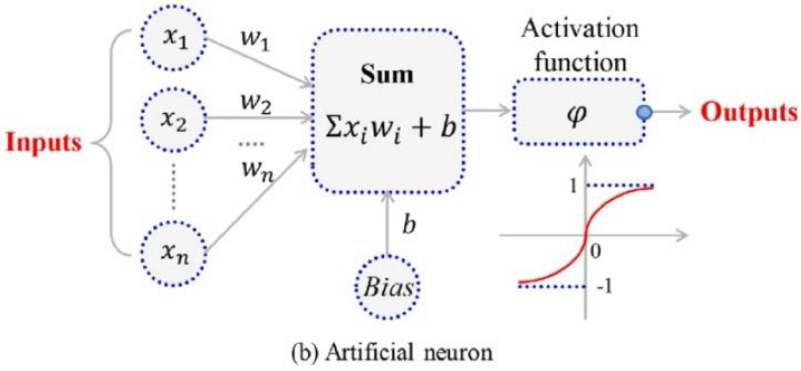
1. Biological Neuron (Real Brain Cell)

A **biological neuron** is a cell in the nervous system that processes and transmits information using electrical and chemical signals.

🔷 Structure

- **Dendrites** → receive signals from other neurons
- **Cell body (soma)** → processes signals
- **Axon** → sends output signal
- **Synapses** → connections to other neurons





(b) Artificial neuron

🔷 Function

- Receives multiple input signals
- Each signal has a **strength (synaptic weight)**
- If total signal exceeds a **threshold**, neuron fires

👉 This behavior relates to the concept of a threshold activation.

2. Perceptron (Single Neurone Model)

🔷 Definition

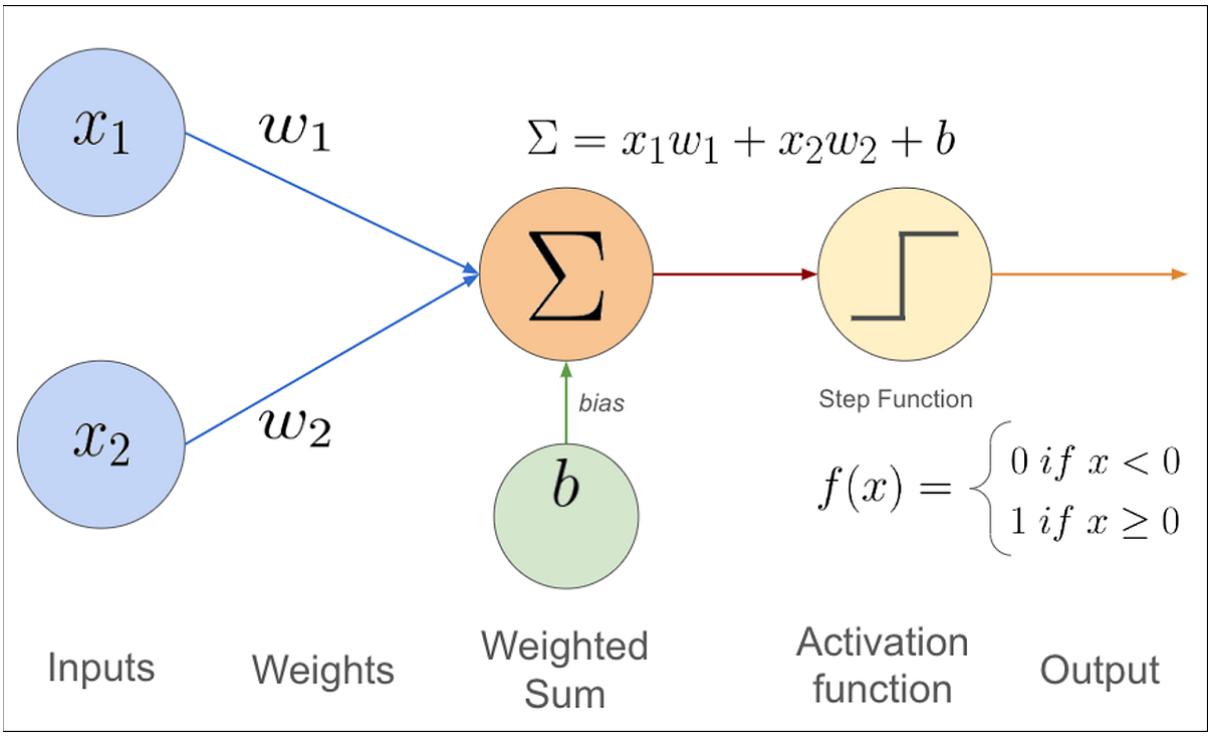
The **Perceptron** is the simplest neural unit:

$$z = w^T x + b$$

$$y = \phi(z)$$

Where:

- x: input vector
- w: weights
- b: bias
- ϕ: activation function



- **Weights (w):** Importance of each input
- **Bias (b):** Shifts activation (like offset voltage)

👉 Electrical analogy:

- Weight → amplifier gain
- Bias → DC offset

Biological Neuron	Perceptron
Dendrites receive signals	Inputs x_i
Synapse strength	Weights w_i
Soma processes signals	Summation $\sum w_i x_i$
Firing threshold	Bias b + activation
Axon sends output	Output y

🔷 Example (EE: Fault Detection)

Suppose:

- x_1: voltage
- x_2: current

We classify:

- 0 → Normal
- 1 → Fault

Given:

$$w = [0.6, 0.4], \quad b = -0.5$$

$$x = [1, 1]$$

Solution:

$$z = 0.6(1) + 0.4(1) - 0.5 = 0.5$$

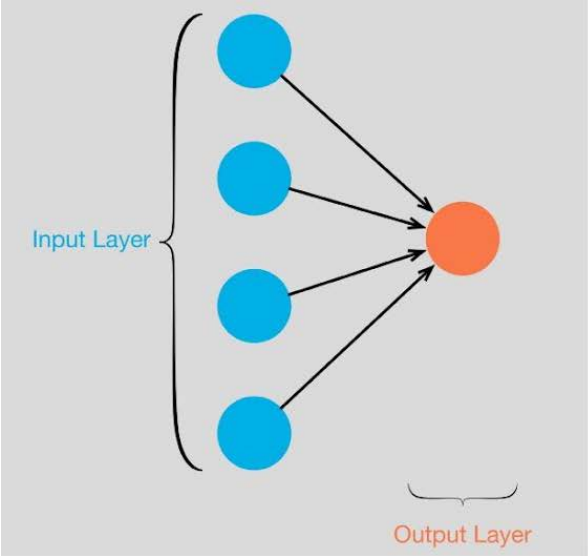
If we use a step function:

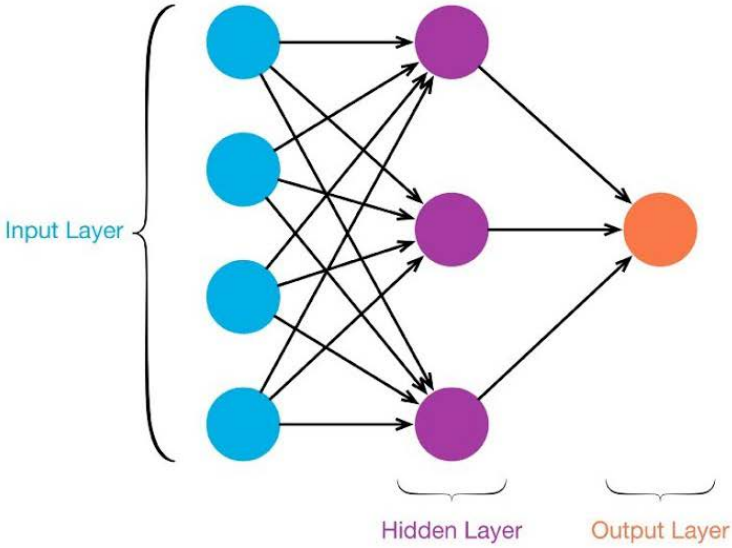
- $z > 0 \rightarrow$ Fault $\rightarrow$ ✅ 1

Single-Layer Perceptron vs. Multilayer Perceptron

1. Single-layer perceptron (only input → output).

2. Multilayer perceptron with one hidden layer (input → hidden → output). Hidden layers add non-linearity and power.





🔷 Structure

A neural network is composed of:

1. Input Layer

- Receives raw data
- No computation

2. Hidden Layers

- Perform transformations
- Extract features

3. Output Layer

- Produces final result

🔄 3. Types of Architectures

🔷 3.1 Feedforward Neural Network (FNN)

- Simplest architecture
- Data flows in one direction

Structure:

Input → Hidden → Output

🔷 3.2 Multilayer Perceptron (MLP)

- Multiple hidden layers
- Fully connected

🔷 3.3 Convolutional Neural Network (CNN)

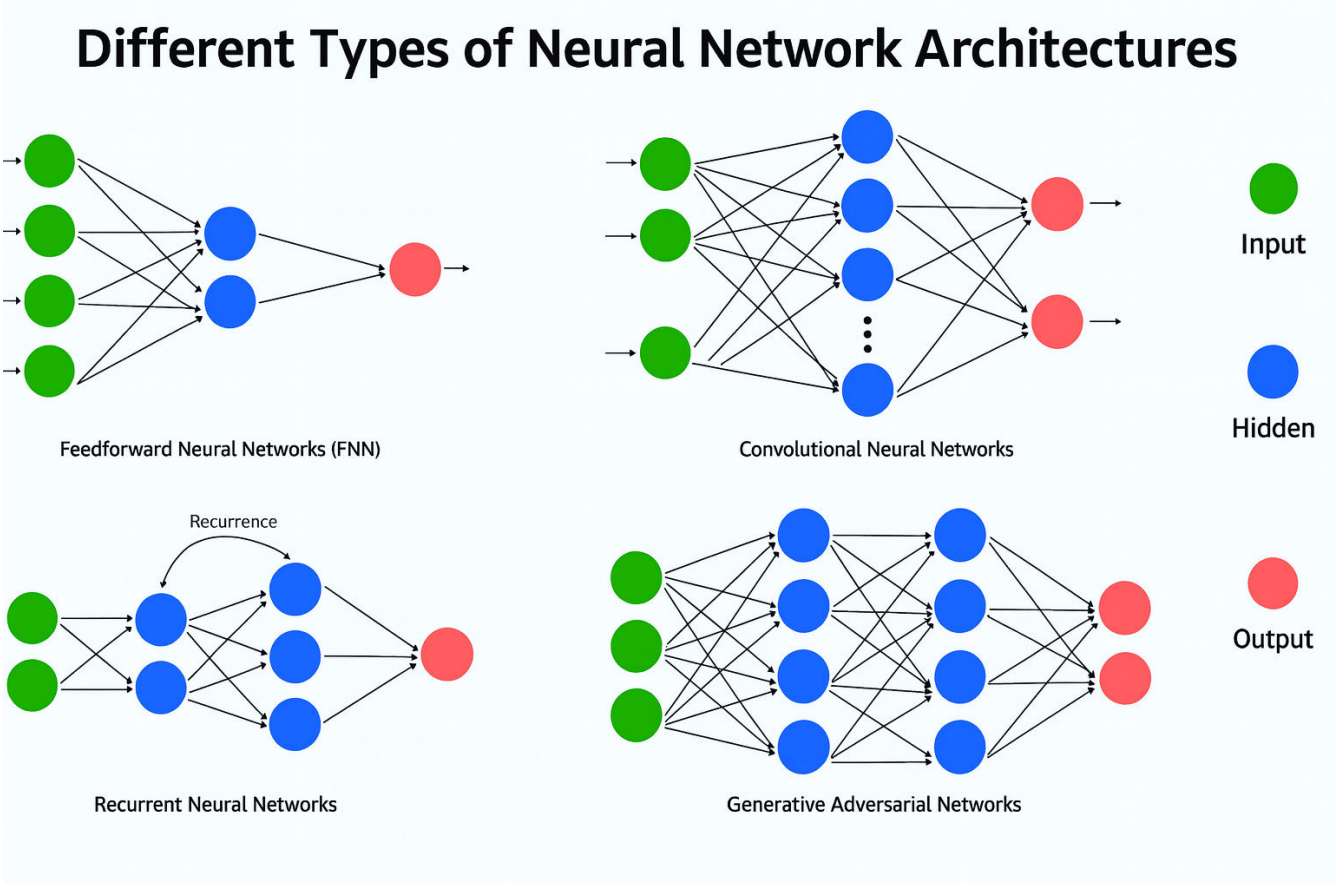
- Used for images and signals
- Local feature extraction

🔷 3.4 Recurrent Neural Network (RNN)

- Handles time-dependent signals
- Has memory (feedback loop)

👉 Useful in EE for:

- Signal processing
- Fault prediction



4. Common Activation Functions

✅ 1. Sigmoid Function

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

✔ **Properties:**

- Output range: $(0, 1)$
- Smooth and differentiable

✔ **Derivative:**

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

✔ **Use:**

- Binary classification
- Output layer (probabilities)

✅ 2. Hyperbolic Tangent (Tanh)

$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

✔ **Properties:**

- Output range: $(-1, 1)$
- Zero-centered

✔ **Derivative:**

$$\tanh'(z) = 1 - \tanh^2(z)$$

✔ **Use:**

- Hidden layers (better than sigmoid in many cases)

✅ 3. ReLU (Rectified Linear Unit)

$f(z) = \max(0, z)$

✔ **Properties:**

- Output: $[0, +\infty)$
- Very simple and efficient

✔ **Derivative:**

$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

✔ **Use:**

- Most popular for hidden layers

✅ 4. Leaky ReLU

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

✔ Properties:

- Fixes "dying ReLU" problem
- α is small (e.g., 0.01)

✅ 5. Softmax Function (Multiclass)

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j-1}^{K} e^{z_j}}$$

✔ Properties:

- Converts vector into probability distribution
- Sum = 1

✔ Use:

- Output layer for multi-class classification

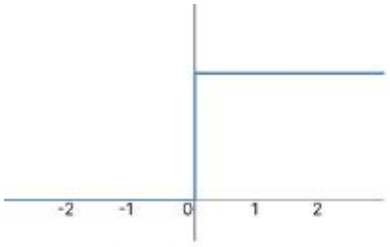
✅ 6. Linear Activation (Identity)

$$f(z) = z$$

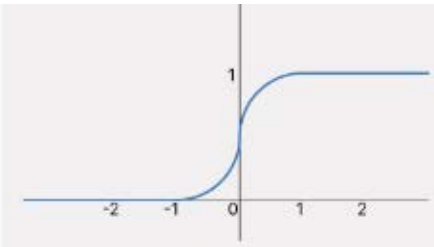
✔ Use:

- Regression problems (e.g., voltage prediction, system output in EE)

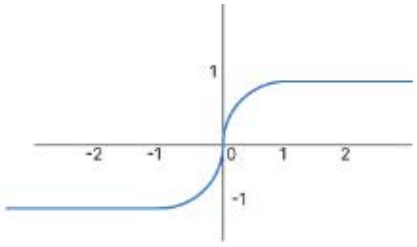
Function	Formula	Range	Typical Use
Sigmoid	$\frac{1}{1+e^{-z}}$	(0,1)	Binary output
Tanh	$\tanh(z)$	(-1,1)	Hidden layers
ReLU	$\max(0, z)$	[0,∞)	Hidden layers
Leaky ReLU	piecewise	(-∞,∞)	Deep networks
Softmax	normalized exp	(0,1)	Multiclass output
Linear	z	(-∞,∞)	Regression



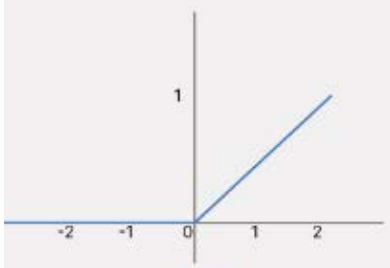
Step Function



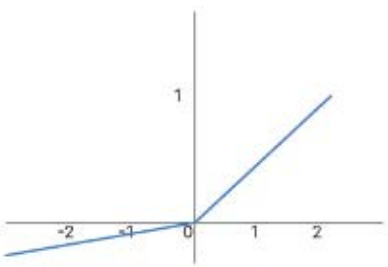
Sigmoid



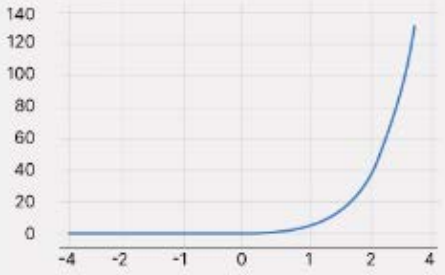
tanh



ReLU



Leaky ReLU



Softmax Function

🟢 Problem: Fault Detection in a Circuit

We measure:

- Voltage V
- Current I

Goal:

Classify system:

- 0 → Normal
- 1 → Fault

🟢 Network Architecture

- Input layer: 2 neurons (V, I)
- Hidden layer: 2 neurons
- Output layer: 1 neuron

🟢 Given:

Weights:

Hidden layer:

$$W_1 = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}$$

Bias:

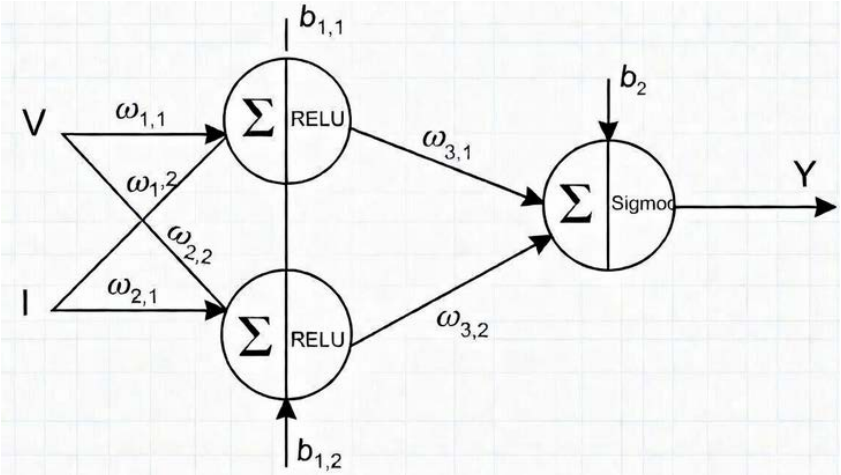
$$b_1 = [0.1, -0.1]$$

Output layer:

$$W_2 = [1.2, -0.7], \quad b_2 = 0.05$$

Input:

$$V = 2, \quad I = 1$$



🟢 Step-by-Step Solution

Step 1: Hidden Layer

Neuron 1:

$$z_1 = (2 \times 0.5) + (1 \times -0.3) + 0.1 = 1 - 0.3 + 0.1 = 0.8$$

Neuron 2:

$$z_2 = (2 \times 0.8) + (1 \times 0.2) - 0.1 = 1.6 + 0.2 - 0.1 = 1.7$$

Apply ReLU:

$$a_1 = 0.8, \quad a_2 = 1.7$$

Step 2: Output Layer

$$z = (0.8 \times 1.2) + (1.7 \times -0.7) + 0.05$$

$$z = 0.96 - 1.19 + 0.05 = -0.18$$

Apply Sigmoid:

$$y = \frac{1}{1 + e^{0.18}} \approx 0.455$$

🟢 Final Decision

- Output ≈ 0.455
- If threshold = 0.5 → **Normal system**

```python
import numpy as np

# Input
x = np.array([2, 1])

# Weights and bias (hidden layer)
W1 = np.array([[0.5, -0.3],
               [0.8,  0.2]])
b1 = np.array([0.1, -0.1])

# Weights and bias (output layer)
W2 = np.array([1.2, -0.7])
b2 = 0.05

# Activation functions
def relu(z):
    return np.maximum(0, z)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Forward pass
z1 = np.dot(W1, x) + b1
a1 = relu(z1)

z2 = np.dot(W2, a1) + b2
y_hat = sigmoid(z2)

# Classification
prediction = 1 if y_hat >= 0.5 else 0

print("z1:", z1)
print("a1 (ReLU):", a1)
print("z2:", z2)
print("y_hat (Sigmoid):", y_hat)
print("Prediction:", prediction)
```

```
                    z1: [0.8 1.7]
             a1 (ReLU): [0.8 1.7]
          z2: -0.1800000000000001
y_hat (Sigmoid): 0.45512110762641994
                    Prediction: 0
```

Single sigmoid perceptron trained on the AND gate

1. Dataset (AND gate)

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

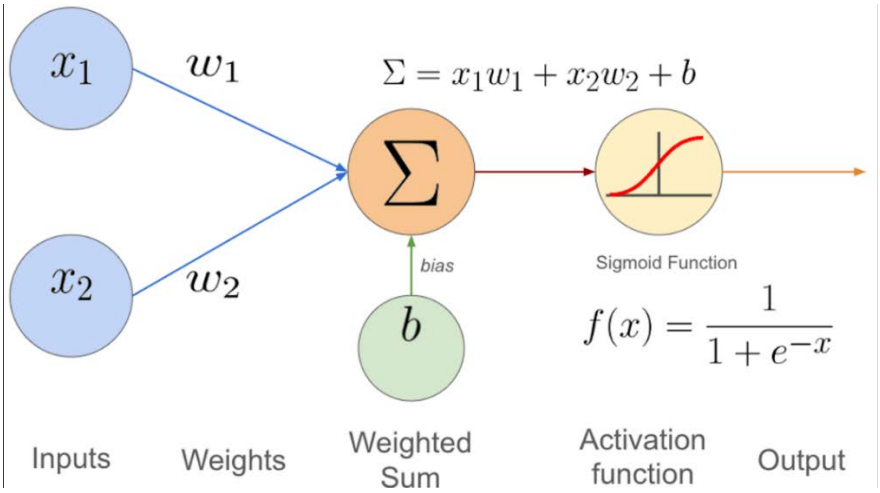
2. Model (Single Neuron)

A single neuron with sigmoid:

Equations:

$$z = w_1 x_1 + w_2 x_2 + b$$

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



3. Loss Function

$$E = \frac{1}{2}(y - \hat{y})^2$$

4. Detailed Gradient Derivation

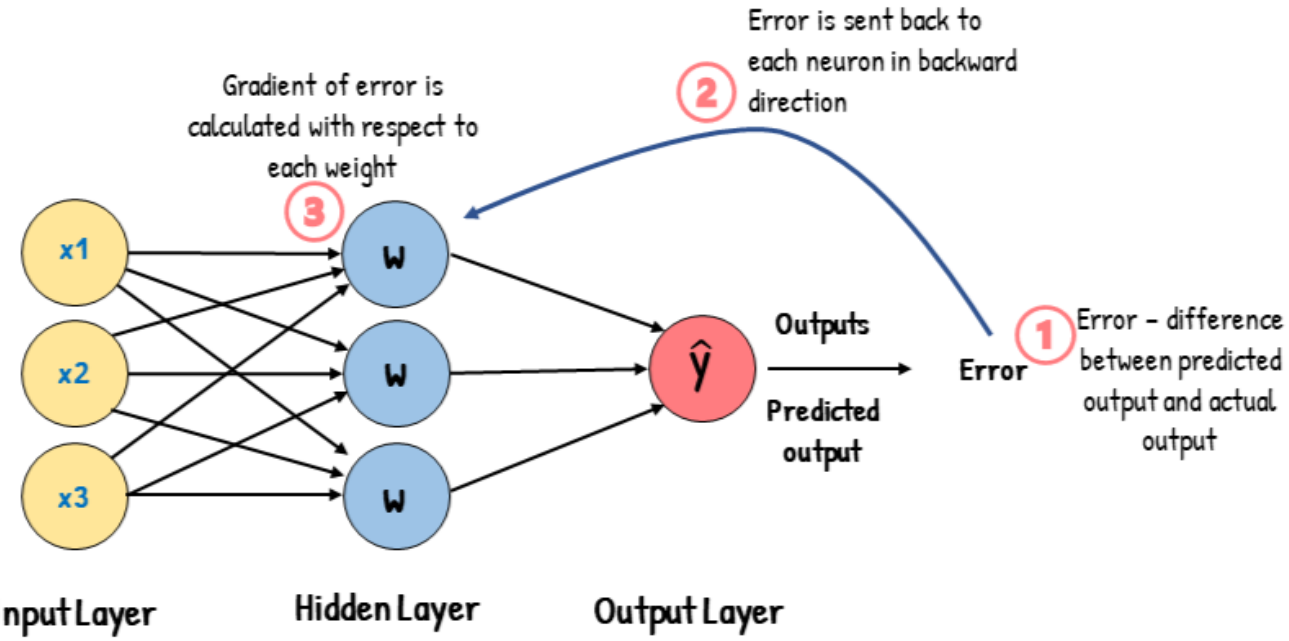
:ompute:

$$\frac{\partial E}{\partial w_i}$$

Step 1: Apply Chain Rule

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Backpropagation



✅ Step 2: Compute Each Term

1. Loss derivative

$$E = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

2. Sigmoid derivative

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$$

✅ Step 3: Final Gradient Formula

$$\frac{\partial E}{\partial w_i} = (\hat{y} - y)\hat{y}(1 - \hat{y})x_i$$

$$\frac{\partial E}{\partial b} = (\hat{y} - y)\hat{y}(1 - \hat{y})$$

🔷 Define Delta

$$\delta = (\hat{y} - y)\hat{y}(1 - \hat{y})$$

So:

$$\frac{\partial E}{\partial w_1} = \delta x_1, \quad \frac{\partial E}{\partial w_2} = \delta x_2, \quad \frac{\partial E}{\partial b} = \delta$$

🔥 6. Forward + Backpropagation (All Samples)

✅ Sample 1: (0,0), y=0

Forward

$$z = 0.1 \Rightarrow \hat{y} = 0.5250$$

Gradient step-by-step

$$(\hat{y} - y) = 0.5250$$

$$\hat{y}(1 - \hat{y}) = 0.5250 \times 0.4750 = 0.2494$$

$$\delta_1 = 0.5250 \times 0.2494 = 0.1310$$

✅ Sample 2: (0,1), y=0

Forward

$$z = -0.2 \Rightarrow \hat{y} = 0.4502$$

Gradient

$$(\hat{y} - y) = 0.4502$$

$$\hat{y}(1 - \hat{y}) = 0.4502 \times 0.5498 = 0.2475$$

$$\delta_2 = 0.4502 \times 0.2475 = 0.1114$$

✅ Sample 3: (1,0), y=0

Forward

$$z = 0.3 \Rightarrow \hat{y} = 0.5744$$

Gradient

$$(\hat{y} - y) = 0.5744$$

$$\hat{y}(1 - \hat{y}) = 0.5744 \times 0.4256 = 0.2445$$

$$\delta_3 = 0.5744 \times 0.2445 = 0.1405$$

✅ Sample 4: (1,1), y=1

Forward

$$z = 0 \Rightarrow \hat{y} = 0.5$$

Gradient

$$(\hat{y} - y) = -0.5$$

$$\hat{y}(1 - \hat{y}) = 0.5 \times 0.5 = 0.25$$

$$\delta_4 = -0.5 \times 0.25 = -0.125$$

🔷 7. Gradient Aggregation (Batch Mode)

✅ Weight w_1

$$\frac{\partial E}{\partial w_1} = 0 + 0 + 0.1405 - 0.125 = 0.0155$$

✅ Weight w_2

$$\frac{\partial E}{\partial w_2} = 0 + 0.1114 + 0 - 0.125 = -0.0136$$

✅ Bias

$$\frac{\partial E}{\partial b} = 0.1310 + 0.1114 + 0.1405 - 0.125 = 0.2579$$

🔷 8. Parameter Update

$$w_1^{new} = 0.2 - 0.01(0.0155) = 0.199845$$

$$w_2^{new} = -0.3 - 0.01(-0.0136) = -0.299864$$

$$b^{new} = 0.1 - 0.01(0.2579) = 0.097421$$

✅ 9. Final Results

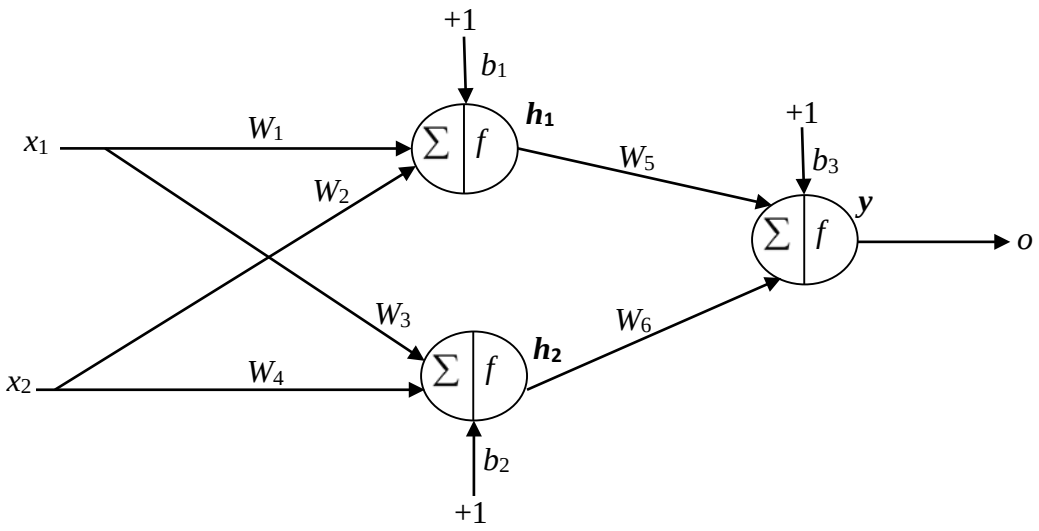
$$w_1 \approx 0.199845 \quad , \quad w_2 \approx -0.299864 \quad , \quad b \approx 0.097421$$

```python
import numpy as np
import matplotlib.pyplot as plt

# ========================================
# 1. PARAMETERS (UPDATED)
# ========================================
w1 = 0.2
w2 = -0.3
b  = 0.1

learning_rate = 0.01   # 🔥 updated
epochs = 1000          # 🔥 updated

# ========================================
# 2. DATASET (AND gate)
# ========================================
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

Y = np.array([0, 0, 0, 1])

# ========================================
# 3. SIGMOID
# ========================================
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ========================================
# 4. TRAINING
# ========================================
loss_history = []

for epoch in range(epochs):

    grad_w1 = 0
    grad_w2 = 0
    grad_b  = 0
    total_loss = 0

    for i in range(len(X)):
        x1, x2 = X[i]
        y = Y[i]

        # Forward
        z = w1 * x1 + w2 * x2 + b
        y_hat = sigmoid(z)

        # Loss (MSE)
        loss = 0.5 * (y - y_hat) ** 2
        total_loss += loss

        # Backprop
        dE_dyhat = (y_hat - y)
        dyhat_dz = y_hat * (1 - y_hat)
        delta = dE_dyhat * dyhat_dz

        grad_w1 += delta * x1
        grad_w2 += delta * x2
        grad_b  += delta

    # Update
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
    b  -= learning_rate * grad_b

    loss_history.append(total_loss)

    # Print only first epoch
    if epoch == 0:
        print("=== AFTER FIRST EPOCH ===")
        print(f"w1 = {w1:.6f}")
        print(f"w2 = {w2:.6f}")
        print(f"b  = {b:.6f}")

# ========================================
# FINAL PARAMETERS
# ========================================
print("\n=== FINAL PARAMETERS ===")
print(f"w1 = {w1:.6f}")
print(f"w2 = {w2:.6f}")
print(f"b  = {b:.6f}")

# ========================================
# 5. TEST RESULTS
# ========================================
print("\n=== TEST RESULTS ===")

for i in range(len(X)):
    x1, x2 = X[i]

    z = w1 * x1 + w2 * x2 + b
    y_hat = sigmoid(z)
    pred = 1 if y_hat >= 0.5 else 0

    print(f"Input {X[i]} -> y_hat = {y_hat:.4f}, pred = {pred}, true = {Y[i]}")

# ========================================
# 6. LOSS PLOT
# ========================================
plt.figure()
plt.plot(loss_history)
plt.title("Loss Convergence")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid()
plt.show()
```

```
# ========================================
# 7. DECISION BOUNDARY
# ========================================
plt.figure()

# Grid
x1_vals = np.linspace(-0.5, 1.5, 100)
x2_vals = np.linspace(-0.5, 1.5, 100)
xx, yy = np.meshgrid(x1_vals, x2_vals)

Z = sigmoid(w1 * xx + w2 * yy + b)

plt.contourf(xx, yy, Z, levels=50, alpha=0.6)

# Data points
for i in range(len(X)):
    x1, x2 = X[i]
    if Y[i] == 0:
        plt.scatter(x1, x2, marker='o')
    else:
        plt.scatter(x1, x2, marker='x', s=100)

# Decision boundary
x_vals = np.linspace(-0.5, 1.5, 100)
y_vals = -(w1 * x_vals + b) / w2
plt.plot(x_vals, y_vals)

plt.title("Decision Boundary (AND Gate)")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid()
plt.show()
```

Exercice:

Let it be a **multilayer perceptron neural network** with an input layer, one hidden layer, and an output layer. The activation function used in this case is of the **sigmoid type**, the diagram of the neural network is represented in the following figure:



For the following dataset:

x_1	x_2	o_d
0.1	0.3	0.03

- Determine the value of the **quadratic error**
- Using the **backpropagation algorithm**, compute the weights $W_1 \ldots W_6$ and $b_1 \ldots b_3$ after one training epoch, given the following initial conditions:

W_1	W_2	W_3	W_4	W_5	W_6	b_1	b_2	b_3
0.5	0.1	0.62	0.2	-0.2	0.3	0.4	-0.1	1.83

And a learning rate η=0.01.

Sol:

✅ 1. Problem Setup

We have a **Multilayer Perceptron (MLP)**:

- Inputs: $x_1 = 0.1, x_2 = 0.3$
- Hidden layer: 2 neurons h_1, h_2
- Output layer: 1 neuron y
- Activation: **Sigmoid**

$$f(x) = \frac{1}{1 + e^{-x}}$$

Initial Weights & Biases:

Parameter	Value
W1	0.5
W2	0.1
W3	0.62
W4	0.2
W5	-0.2
W6	0.3
b1	0.4
b2	-0.1
b3	1.83

- Learning rate: $\eta = 0.01$
- Desired output: $o_d = 0.03$

✅ 2. Forward Propagation

Hidden Layer

Neuron h_1:

$$z_1 = W1 \cdot x_1 + W2 \cdot x_2 + b1$$

$$z_1 = (0.5)(0.1) + (0.1)(0.3) + 0.4 = 0.05 + 0.03 + 0.4 = 0.48$$

$$h_1 = \sigma(0.48) = \frac{1}{1 + e^{-0.48}} \approx 0.6177$$

Neuron h_2:

$$z_2 = W3 \cdot x_1 + W4 \cdot x_2 + b2$$

$$z_2 = (0.62)(0.1) + (0.2)(0.3) - 0.1 = 0.062 + 0.06 - 0.1 = 0.022$$

$$h_2 = \sigma(0.022) \approx 0.5055$$

Output Layer

$$z_3 = W5 \cdot h_1 + W6 \cdot h_2 + b3$$

$$z_3 = (-0.2)(0.6177) + (0.3)(0.5055) + 1.83$$

$$z_3 = -0.1235 + 0.1516 + 1.83 = 1.8581$$

$$y = \sigma(1.8581) \approx 0.8651$$

✅ 3. Error (Quadratic)

$$E = \frac{1}{2}(o_d - y)^2$$

$$E = \frac{1}{2}(0.03 - 0.8651)^2$$

$$E = \frac{1}{2}(-0.8351)^2 \approx 0.3487$$

✅ 4. Backpropagation

Gradient Descent Update Rule

Once gradients are computed:

$$W_{new} = W - \eta \cdot \frac{\partial E}{\partial W}$$

Goal of Backpropagation

We want to minimize the error:

$$E = \frac{1}{2}(o_d - y)^2$$

👉 So we compute:

$$\frac{\partial E}{\partial W}$$

for every weight in the network.

Backpropagation is just repeated use of the **chain rule:**

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial W}$$

Output layer:

- $z_3 = W5 \cdot h_1 + W6 \cdot h_2 + b3$
- $y = \sigma(z_3)$

Hidden layer:

- $z_1 = W1x_1 + W2x_2 + b1$
- $h_1 = \sigma(z_1)$
- $z_2 = W3x_1 + W4x_2 + b2$
- $h_2 = \sigma(z_2)$

🔴 STEP 1: Output Layer Gradient

We compute:

$$\frac{\partial E}{\partial W5}$$

Apply chain rule:

$$\frac{\partial E}{\partial W5} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z_3} \cdot \frac{\partial z_3}{\partial W5}$$

1. Error derivative:

$$\frac{\partial E}{\partial y} = (y - o_d)$$

2. Sigmoid derivative:

$$\frac{\partial y}{\partial z_3} = y(1 - y)$$

3. Linear derivative:

$$\frac{\partial z_3}{\partial W5} = h_1$$

✅ **Final result:**

$$\frac{\partial E}{\partial W5} = (y - o_d) \cdot y(1 - y) \cdot h_1$$

👉 Define:

$$\delta_3 = (y - o_d) \cdot y(1 - y)$$

Then:

$$\frac{\partial E}{\partial W5} = \delta_3 \cdot h_1$$
$$\frac{\partial E}{\partial W6} = \delta_3 \cdot h_2$$
$$\frac{\partial E}{\partial b3} = \delta_3$$

🔵 STEP 2: Hidden Layer Gradient

Now we compute:

$$\frac{\partial E}{\partial W1}$$

$$\frac{\partial E}{\partial W1} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W1}$$

Compute each term:

1. $\frac{\partial E}{\partial y} = (y - o_d)$
2. $\frac{\partial y}{\partial z_3} = y(1 - y)$
3. $\frac{\partial z_3}{\partial h_1} = W5$
4. $\frac{\partial h_1}{\partial z_1} = h_1(1 - h_1)$
5. $\frac{\partial z_1}{\partial W1} = x_1$

$$\frac{\partial E}{\partial W1} = (y - o_d) \cdot y(1 - y) \cdot W5 \cdot h_1(1 - h_1) \cdot x_1$$

👉 Define:

$$\delta_1 = \delta_3 \cdot W5 \cdot h_1(1 - h_1)$$

Then:

$$\frac{\partial E}{\partial W1} = \delta_1 \cdot x_1$$
$$\frac{\partial E}{\partial W2} = \delta_1 \cdot x_2$$

For neuron h_2:

$$\delta_2 = \delta_3 \cdot W6 \cdot h_2(1 - h_2)$$

$$\frac{\partial E}{\partial W3} = \delta_2 \cdot x_1$$

$$\frac{\partial E}{\partial W4} = \delta_2 \cdot x_2$$

Weight Updates

$$\delta_3 = (y - o_d) \cdot y(1 - y)$$

$$\delta_3 = (0.8651 - 0.03)(0.8651)(1 - 0.8651)$$

$$\delta_3 = 0.8351 \cdot 0.8651 \cdot 0.1349 \approx 0.0973$$

$$\delta_2 = \delta_3 \cdot W6 \cdot h_2(1 - h_2)$$

$$\delta_2 = 0.0973 \cdot 0.3 \cdot (0.5055)(0.4945)$$

$$\delta_2 \approx 0.0073$$

$$\delta_1 = \delta_3 \cdot W5 \cdot h_1(1 - h_1)$$

$$\delta_1 = 0.0973 \cdot (-0.2) \cdot (0.6177)(0.3823)$$

$$\delta_1 \approx -0.0046$$

$$W5_{new} = W5 - \eta \cdot \delta_3 \cdot h_1$$

$$W5_{new} = -0.2 - 0.01(0.0973)(0.6177) \approx -0.2006$$

$$W6_{new} = 0.3 - 0.01(0.0973)(0.5055) \approx 0.2995$$

$$b3_{new} = 1.83 - 0.01(0.0973) \approx 1.8290$$

$W1$

$$W1_{new} = 0.5 - 0.01(-0.0046)(0.1) \approx 0.5000046$$

$W2$

$$W2_{new} = 0.1 - 0.01(-0.0046)(0.3) \approx 0.100014$$

$W3$

$$W3_{new} = 0.62 - 0.01(0.0073)(0.1) \approx 0.619993$$

$W4$

$$W4_{new} = 0.2 - 0.01(0.0073)(0.3) \approx 0.199978$$

$$b1_{new} = 0.4 - 0.01(-0.0046) \approx 0.400046$$

$$b2_{new} = -0.1 - 0.01(0.0073) \approx -0.100073$$

Python

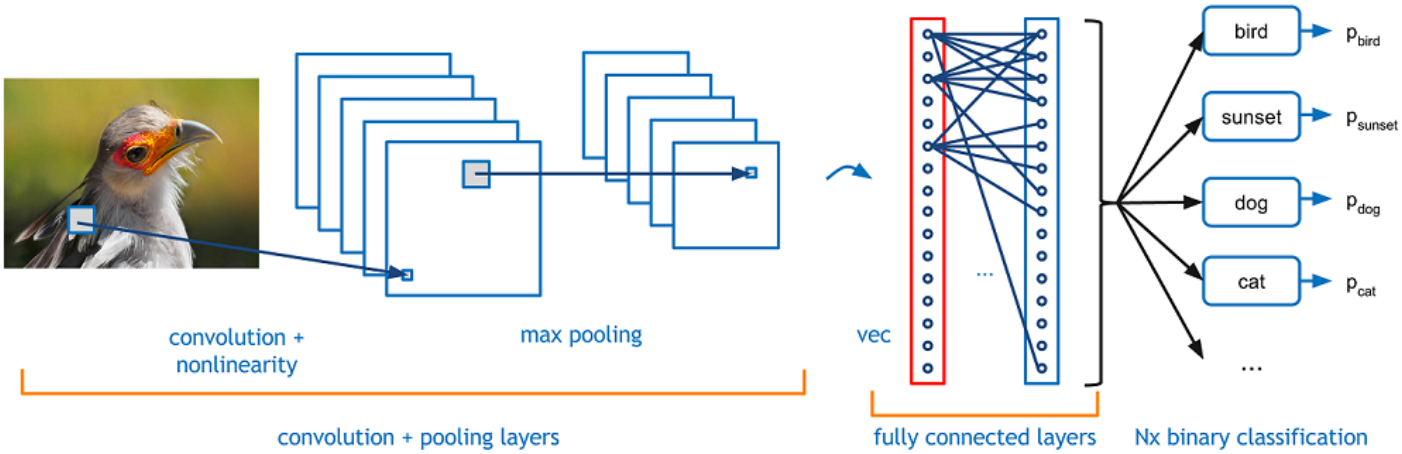
```python
import numpy as np

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Inputs
x = np.array([0.1, 0.3])
target = 0.03

# Weights
W1, W2, W3, W4 = 0.5, 0.1, 0.62, 0.2
W5, W6 = -0.2, 0.3

# Biases
b1, b2, b3 = 0.4, -0.1, 1.83

lr = 0.01

# ===== Forward =====
h1_in = W1*x[0] + W2*x[1] + b1
h1 = sigmoid(h1_in)

h2_in = W3*x[0] + W4*x[1] + b2
h2 = sigmoid(h2_in)

y_in = W5*h1 + W6*h2 + b3
y = sigmoid(y_in)

# Error
E = 0.5 * (target - y)**2

# ===== Backward =====
delta3 = (y - target) * sigmoid_derivative(y)

delta1 = delta3 * W5 * sigmoid_derivative(h1)
delta2 = delta3 * W6 * sigmoid_derivative(h2)

# ===== Update =====
W5 -= lr * delta3 * h1
W6 -= lr * delta3 * h2
b3 -= lr * delta3

W1 -= lr * delta1 * x[0]
W2 -= lr * delta1 * x[1]
b1 -= lr * delta1

W3 -= lr * delta2 * x[0]
W4 -= lr * delta2 * x[1]
b2 -= lr * delta2

# Print results
print("Output:", y)
print("Error:", E)
print("Updated weights:")
print(W1, W2, W3, W4, W5, W6)
print("Updated biases:")
print(b1, b2, b3)
```

```
Output: 0.8650753756786712

Error: 0.3486754415324369

Updated weights:

0.5000046032229463 0.10001380966883887 0.6199926906365043 0.19997807190951294 -0.20060211868586833
0.2995072895034133

Updated biases:

0.4000460322294629 -0.10007309363495694 1.8290253002714203
```

🟦 Definition of a Convolutional Neural Network (CNN)

A **Convolutional Neural Network (CNN)** is a type of **deep neural network** specifically designed to process data that has a **grid-like structure**, such as:

- Images (2D grids of pixels) 🖼️
- Signals or time series (1D sequences) 🔊



🔷 Formal Mathematical Definition

A CNN is a function:

$$y = f(x; W, b)$$

Where:

- x: input data (image or signal)
- y: output (class, prediction, feature)
- W: set of convolution kernels (filters)
- b: biases

The core operation is **convolution**, defined as:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

🔷 Simple Explanation

👉 A CNN works like a **smart filter bank** that:

1. Scans the input
2. Detects important patterns (edges, shapes, signal changes)
3. Combines them to make decisions

📘 Example 1: 2D Convolution (Image / Signal Map)

🔷 Given

Input (3×3 signal/image):

$$I = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

Kernel (2×2):

$$K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

📐 🔷 Step 1: Output Size

Formula:

$$\text{Output size} = (N - F + 1)$$

Where:

- $N = 3$ (input size)
- $F = 2$ (kernel size)

$$3 - 2 + 1 = 2$$

👉 Output is **2 × 2**

🔷 Convolution Formula

$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$

🔷 Step-by-Step Solution

1. Position (0,0)

$$(1 \times 1 + 2 \times 0 + 0 \times 0 + 1 \times (-1)) = 1 - 1 = 0$$

2. Position (0,1)

$$(2 \times 1 + 0 \times 0 + 1 \times 0 + 3 \times (-1)) = 2 - 3 = -1$$

3. Position (1,0)

$$(0 \times 1 + 1 \times 0 + 2 \times 0 + 1 \times (-1)) = -1$$

4. Position (1,1)

$$(1 \times 1 + 3 \times 0 + 1 \times 0 + 0 \times (-1)) = 1$$

🔷 Output Feature Map

$$\begin{bmatrix} 0 & -1 \\ -1 & 1 \end{bmatrix}$$

🔷 Apply ReLU

$$\text{ReLU}(x) = \max(0, x)$$

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Solution with NumPy (Manual CNN)

```
import numpy as np

# Input matrix
I = np.array([
    [1, 2, 0],
    [0, 1, 3],
    [2, 1, 0]
])

# Kernel
K = np.array([
    [1, 0],
    [0, -1]
])

# Output size
n, f = I.shape[0], K.shape[0]
output_size = n - f + 1

# Initialize output
output = np.zeros((output_size, output_size))

# Convolution
for i in range(output_size):
    for j in range(output_size):
        region = I[i:i+f, j:j+f]
        output[i, j] = np.sum(region * K)

# ReLU
output_relu = np.maximum(0, output)

print("Convolution Output:\n", output)
print("After ReLU:\n", output_relu)
```

```
Convolution Output:
[[ 0. -1.]
 [-1.  1.]]

After ReLU:
[[0. 0.]
 [0. 1.]]
```

Solution with PyTorch (https://colab.research.google.com/)

```
import torch
import torch.nn as nn

# Input (batch=1, channels=1, height=3, width=3)
I = torch.tensor([[[[1., 2., 0.],
                    [0., 1., 3.],
                    [2., 1., 0.]]]])

# Define convolution layer
conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=2, bias=False)

# Set kernel manually
conv.weight.data = torch.tensor([[[[1., 0.],
                                   [0., -1.]]]])

# Apply convolution
output = conv(I)

# Apply ReLU
relu = nn.ReLU()
output_relu = relu(output)

print("Convolution Output:\n", output)
print("After ReLU:\n", output_relu)
```