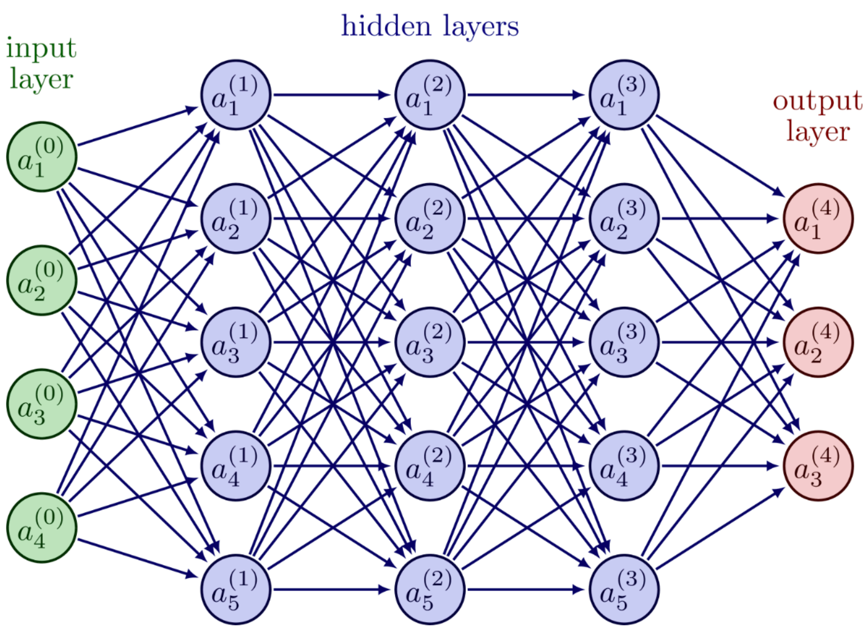
# Introduction to Neural Networks

Iván Moreno

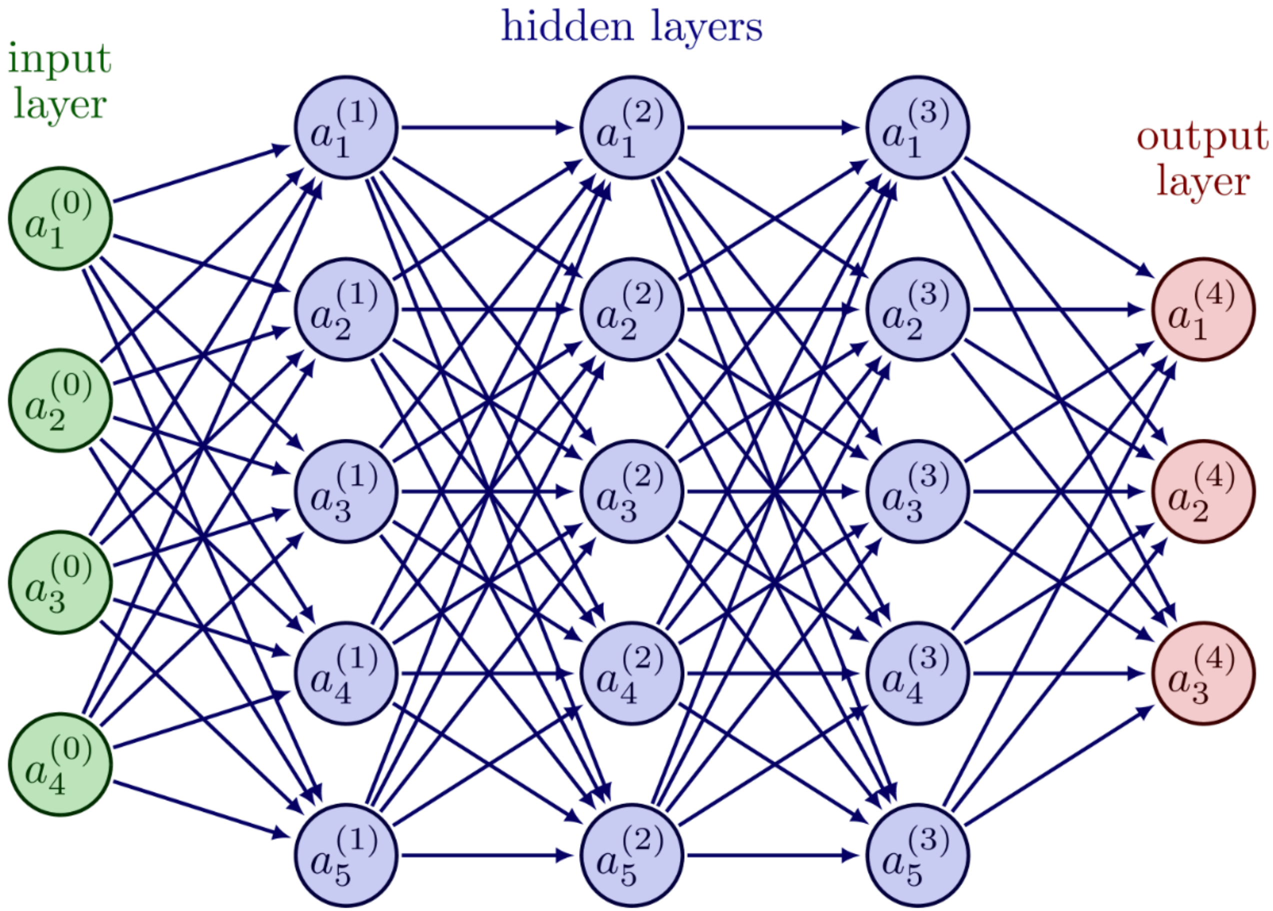
# Overview

1. What is a Neural Network?
2. Perceptrons and Multilayer Perceptrons
3. Forward Propagation
4. Backpropagation and Gradient Descent
5. Activation Functions

# What is a Neural Network?

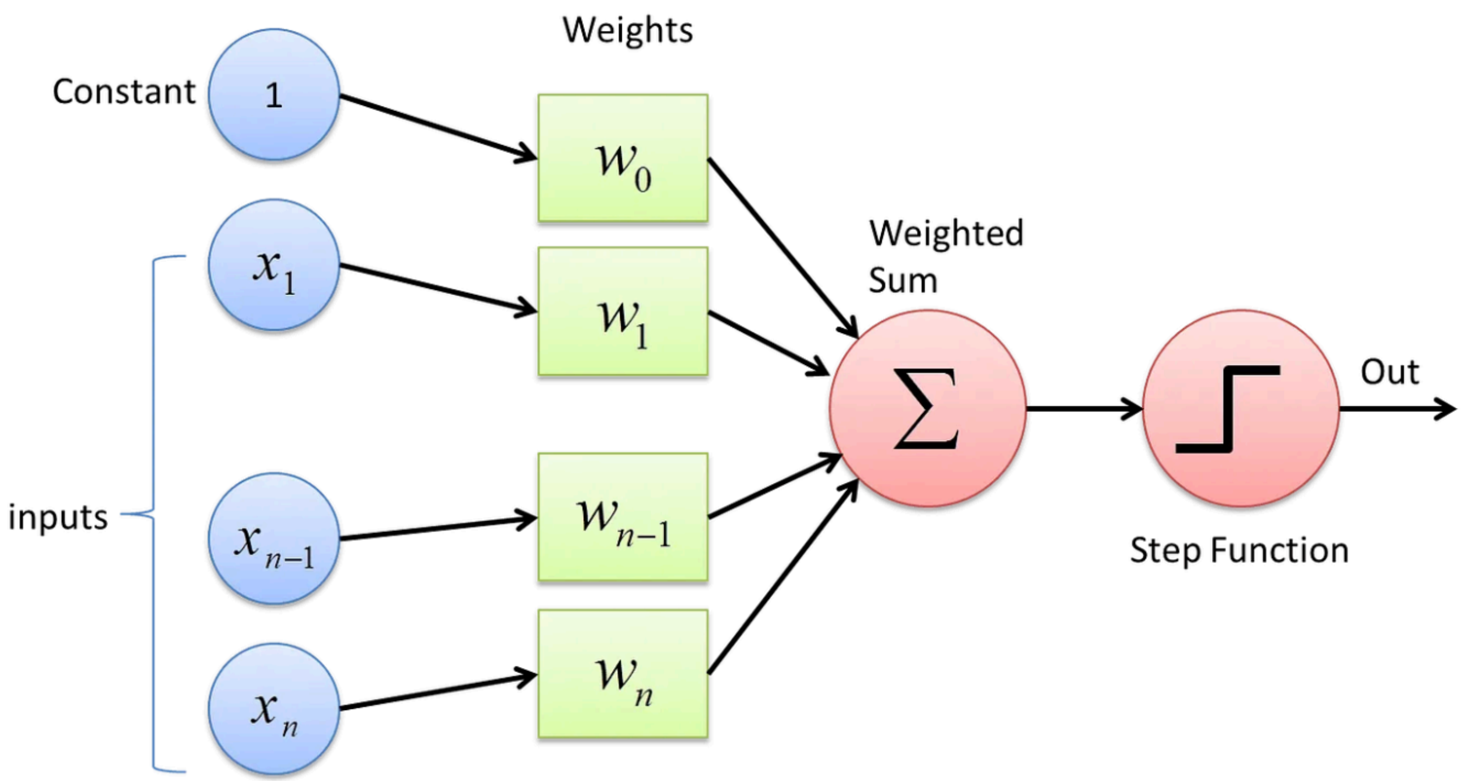



- A neural network is a computational model inspired by the way biological neural networks in the human brain process information.
- Composed of layers of interconnected nodes or neurons.
- Neurons in a neural network are organized into an input layer, hidden layers, and an output layer.




- **Input Layer**: Receives the input data.
- **Hidden Layer**: Processes inputs through weights and biases.
- **Output Layer**: Produces the final output.

# Perceptrons




- The simplest form of a neural network is the perceptron.
- A perceptron models a single neuron with multiple inputs and one output (binary classification).
- It takes a weighted sum of inputs, adds a bias, and applies an activation function.

NICVC

# Perceptron Mathematics

$$\hat{y} = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Where:

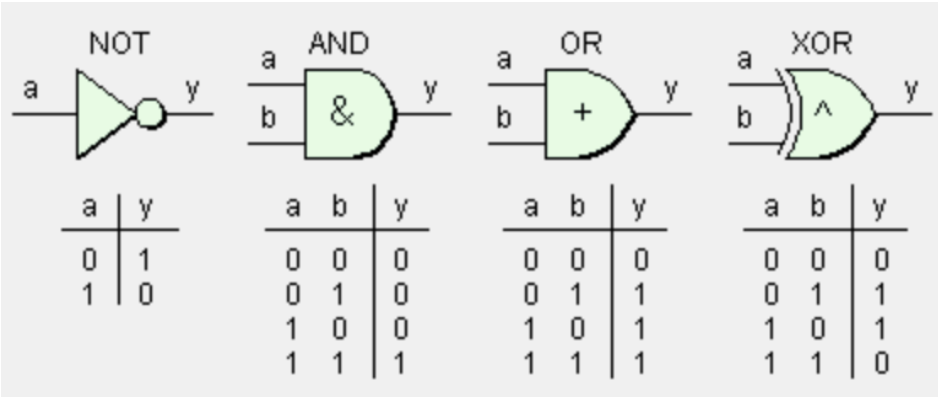
- $\varphi(z)$ is the activation function, and in the case of a perceptron, it is typically the Heaviside step function $H(x)$.

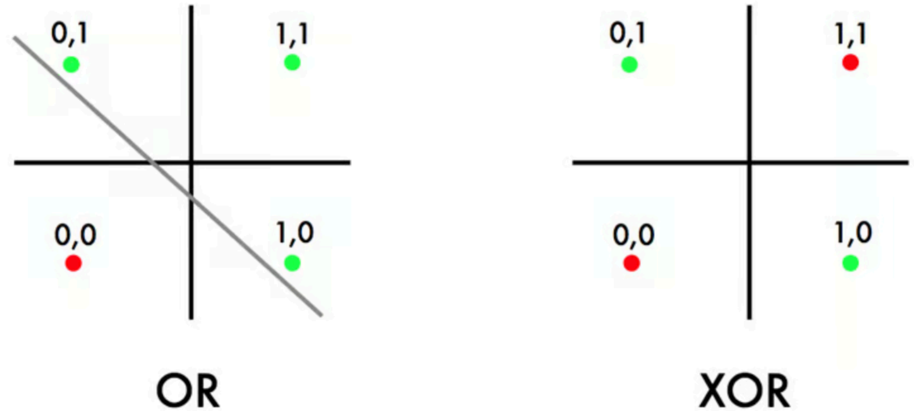
$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- $x_i$ are the input features,
- $w_i$ are the corresponding weights,
- $b$ is the bias,
- $\hat{y}$ is the output of the perceptron.

**Why the bias term?**

The bias term allows the perceptron to learn a decision boundary that does not pass through the origin $(0, 0)$.




### Limitations of Perceptrons

Perceptrons can only model linearly separable functions (AND, OR, NOT), but not XOR.

XOR is not linearly separable, requiring a more complex model.

Multilayer Perceptrons (MLPs) can model complex functions, including XOR.

In a MLP, multiple perceptrons are connected in layers, allowing for non-linear transformations. The hidden layer creates a new feature space where the data becomes linearly separable.

# Implementing a Perceptron in Python

## Write a Python function to implement a basic perceptron:

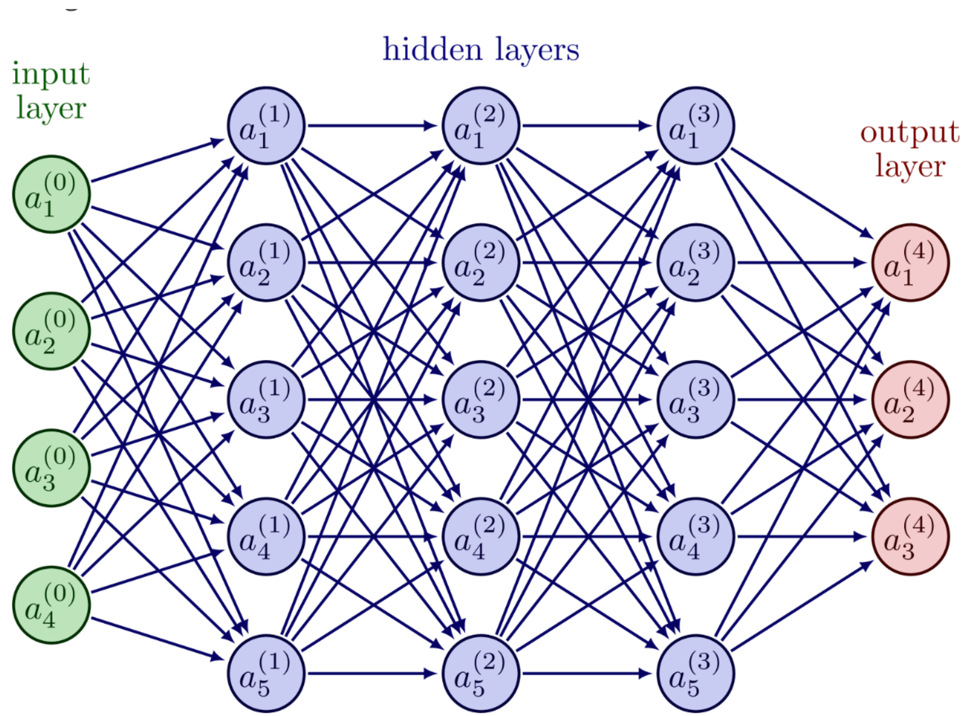
```python
import numpy as np

def perceptron(inputs, weights, bias):
    z = np.dot(inputs, weights) + bias
    return 1 if z > 0 else 0

# Test with sample inputs
inputs = np.array([2, 3])
weights = np.array([0.5, -0.6])
bias = 0.1
output = perceptron(inputs, weights, bias)
output
```

**Why dot product?**

The dot product of inputs and weights is a linear combination that captures the relationship between inputs and weights.

$$\mathrm{w} \cdot \mathrm{x} = \mathrm{w}_1 \mathrm{x}_1 + \mathrm{w}_2 \mathrm{x}_2 + \ldots + \mathrm{w}_n \mathrm{x}_n$$

# Multilayer Perceptrons (MLP)




- **Multilayer Perceptrons (MLPs)** are a type of neural network with one or more hidden layers.
- Each layer is fully connected to the next.
- MLPs can approximate any continuous function given enough neurons and layers (universal approximation theorem), making them powerful function approximators.

# Forward Propagation

- Forward propagation calculates the output of the network by passing data from input to output layers.
- For each layer:
  1. Compute the weighted sum of inputs.
  2. Apply the activation function.

$$a^{(l+1)} = f(W^{(l)}a^{(l)} + b^{(l)})$$

Where:

- $a^{(l)}$ is the activations of the previous layer.
- $W^{(l)}$ and $b^{(l)}$ are the weights and biases of the current layer.
- $f$ is the activation function.

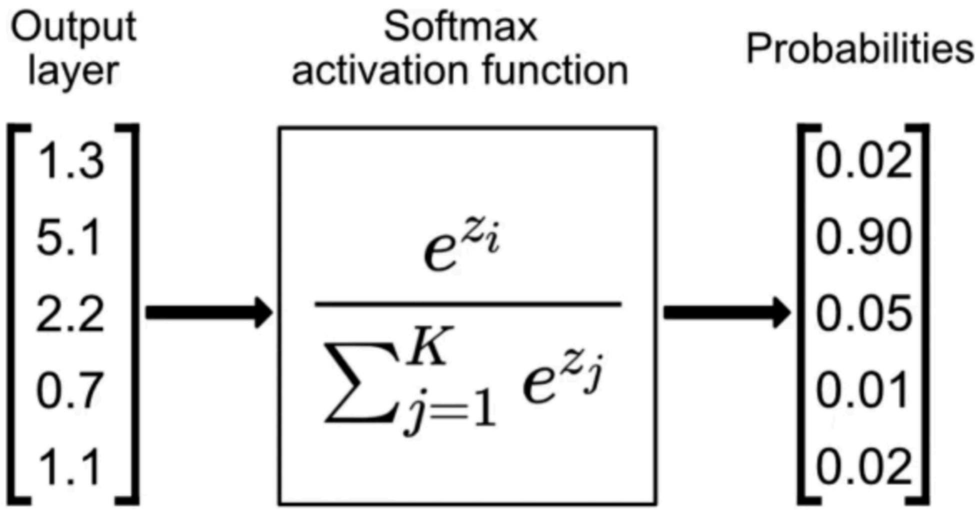
# Implementing Forward Propagation in Python

Implement forward propagation for an n-layer neural network:

```python
import numpy as np

def forward_propagation(X, W, b):
    A = X
    for i in range(len(W)):
        Z = np.dot(W[i], A) + b[i]
        A = sigmoid(Z) if i < len(W) - 1 else softmax(Z)
    return A
```

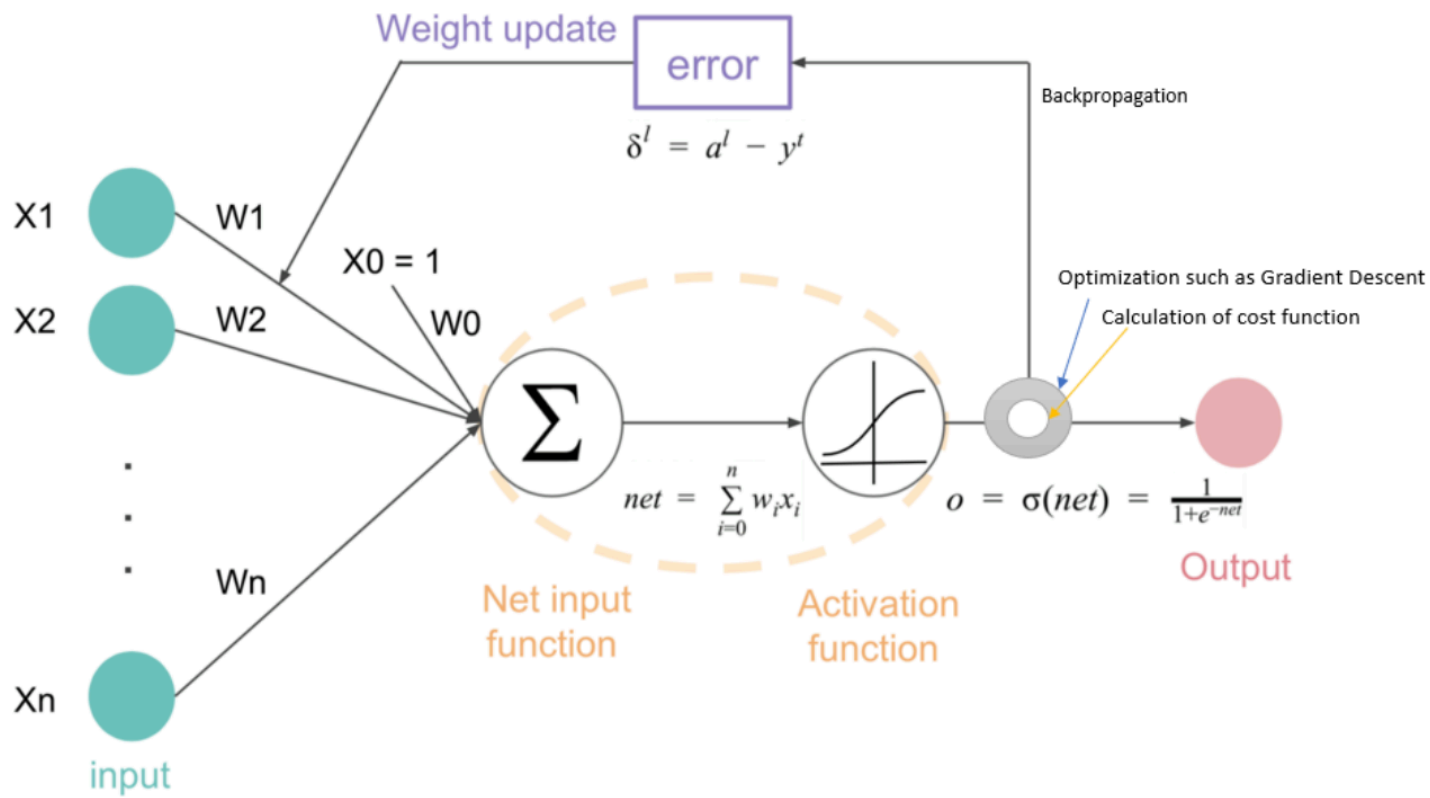
- `X`: Input data.
- `W`: List of weight matrices.
- `b`: List of bias vectors.
- `sigmoid`: Sigmoid activation function.
- `softmax`: Softmax activation function.

**Why softmax?**




The softmax function is used in the output layer of a neural network for multi-class classification tasks. It converts raw scores into probabilities (summing to 1).

# Backpropagation




- **Backpropagation** is the algorithm used to train neural networks.
- It calculates the gradient of the loss function with respect to each weight by the chain rule.
- **Gradient Descent** updates weights to minimize the loss function.

# Backpropagation Mathematics

1. Compute the error at the output layer:

$$\delta^{(L)} = \nabla_a L \odot f'(z^{(L)})$$

2. Propagate the error backward through the network:

$$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot f'(z^{(l)})$$

3. Update weights:

$$W^{(l)} = W^{(l)} - \eta \cdot \delta^{(l)} \cdot (a^{(l-1)})^T$$

Where:

- $L$ is the loss function.
- $\eta$ is the learning rate.
- $\delta^{(l)}$ is the error at layer $l$.
- $a^{(l)}$ is the activation of layer $l$.

# Implementing Backpropagation in Python

Extend the forward propagation function to include backpropagation (with learning rate $\eta$):

```
def forward_propagation(X, W, b):
    A = X
    activations = [A]
    for i in range(len(W)):
        Z = np.dot(W[i], A) + b[i]
        A = sigmoid(Z) if i < len(W) - 1 else softmax(Z)
        activations.append(A)
    return A, activations

def backpropagation(X, Y, W, b, eta):
    A, activations = forward_propagation(X, W, b)
    deltas = [A - Y]
    for i in range(len(W) - 1, 0, -1):
        delta = np.dot(W[i].T, deltas[0]) * sigmoid_derivative(activations[i])
        deltas.insert(0, delta)
    for i in range(len(W)):
        W[i] -= eta * np.dot(deltas[i], activations[i].T)
        b[i] -= eta * deltas[i]
```

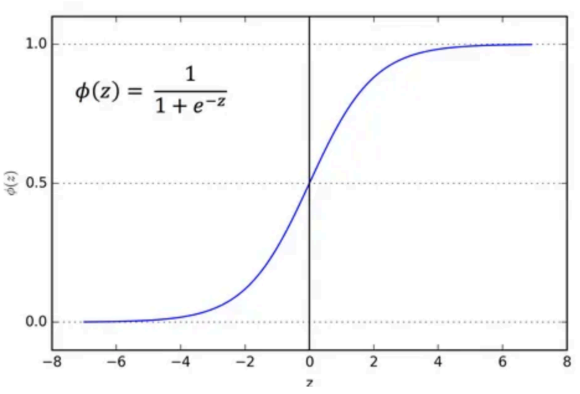
## Intuition:

# Activation Functions

- Activation functions introduce **non-linearity** to the network.
- Why non-linearity?
  - Allows the network to model complex relationships.
  - Without activation functions, the network would collapse to a linear model.
- Common activation functions:
  - Sigmoid (Logistic)
  - $\tanh$ (Hyperbolic Tangent)
  - ReLU (Rectified Linear Unit)

# Activation Function Details

## Sigmoid



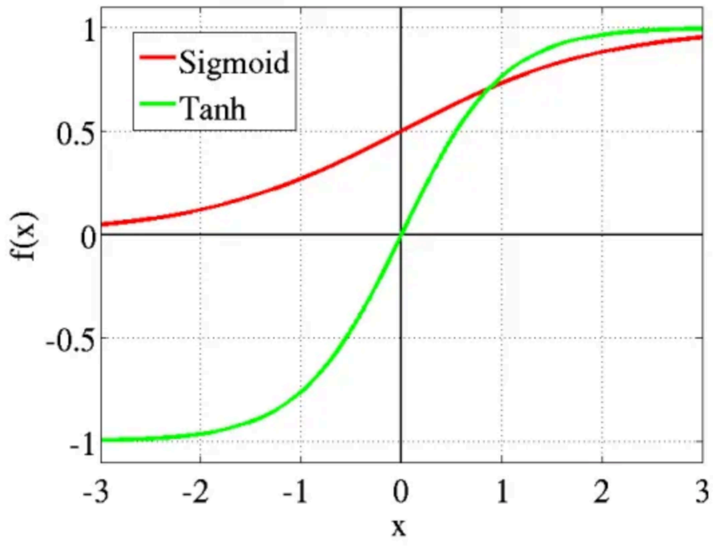

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Sigmoid function squashes the output between 0 and 1.
- Used in the output layer for binary classification.
- Prone to vanishing gradient problem.

**What's the vanishing gradient problem?**

In deep networks, gradients can become very small during backpropagation, leading to slow learning or convergence issues.

# Tanh




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

- Similar to the sigmoid function but centered at 0.
- Output ranges from -1 to 1.
- Helps with zero-centered data.

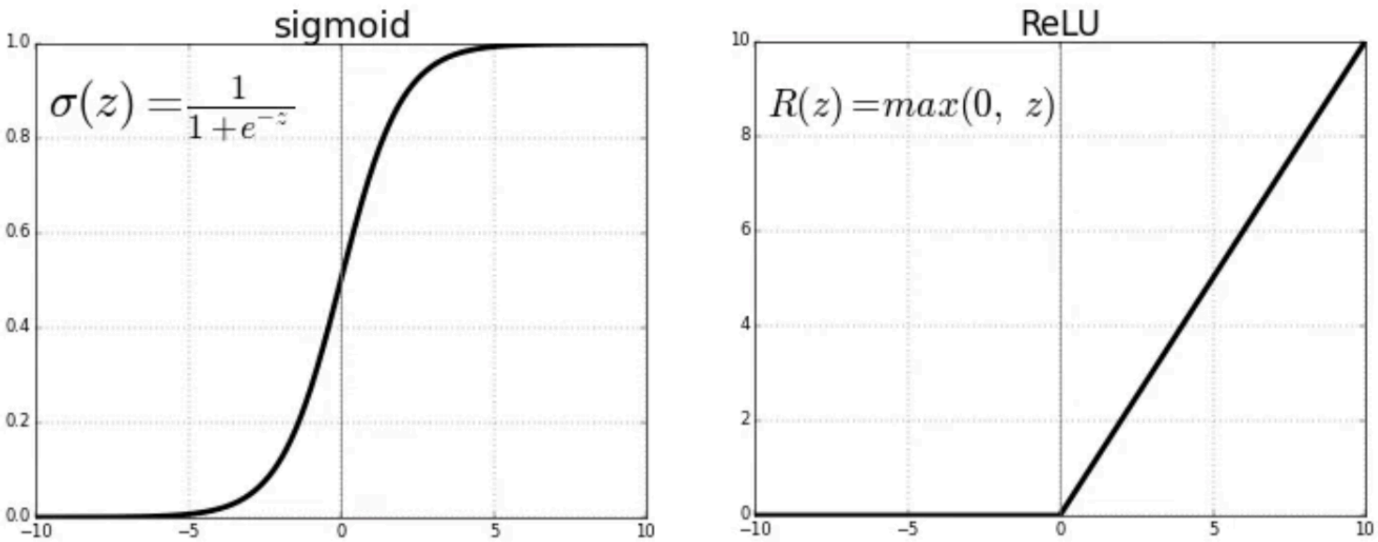
NICVC

**What does it mean for data to be zero-centered?**

Zero-centered data has a mean of 0, which can help with convergence during training, especially when using gradient-based optimization methods.

In non-zero-centered data, the gradients can be biased in a particular direction, leading to slower convergence. Think of it as if every step you take is always leaning towards one side.

# ReLU




- Rectified Linear Unit (ReLU) is widely used in deep learning.
- It is simple and computationally efficient.
- When $z > 0$, the derivative is 1, avoiding the vanishing gradient problem (unlike sigmoid and tanh).

$$\mathrm{ReLU}(z) = \max(0, z)$$

# Conclusion

- Neural networks are powerful tools for modeling complex relationships.
- Key components include layers, activation functions, forward propagation, and backpropagation.
- Understanding the math behind these processes is crucial for designing and debugging neural networks.