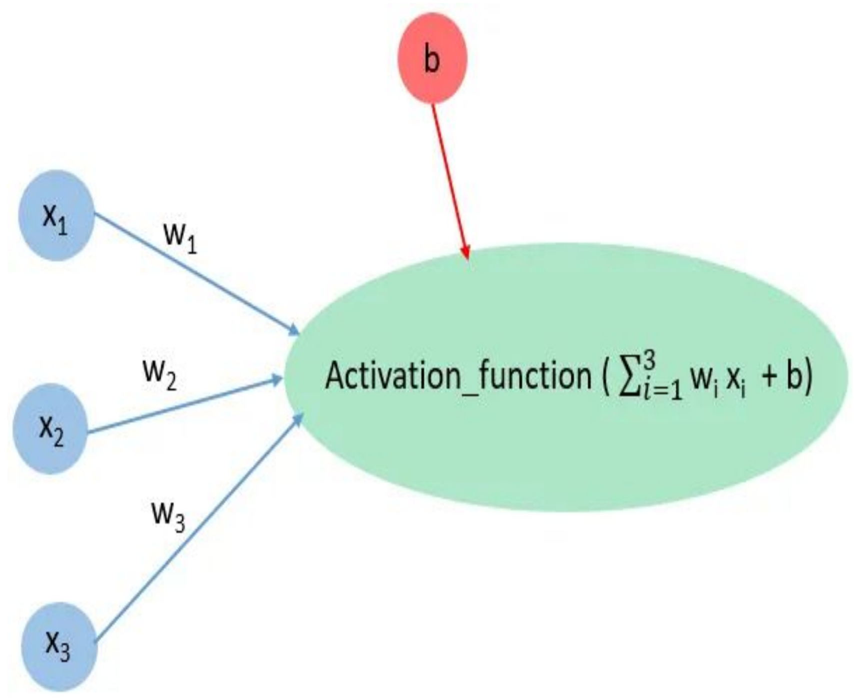
Activation Function

What is an activation function ?

An activation function is a mathematical equation attached to each hidden and output neuron in the network. Its role is to determine whether the neuron should be activated or not.

This is typically based on whether the neuron input is important for the output prediction. Apart from neurons of the input layer, the weighted sum of each neuron inputs, having a bias added to it, is passed through an activation function.

Example of a hidden node (green) : its value is the sum of weighted inputs (blue) and bias (red) passed through an activation function.



Why would we use activation functions in a neural network ?

Referring to the above example, the relation between weights and neuron inputs is always linear if we do not make use of an activation function. Thus, the output is a simple linear transformation of these latters. When only linear relationship between variables are addressed, the predicted model will be unsuitable for complex problems such as computer vision and nature language processing.

Linear and Non linearity

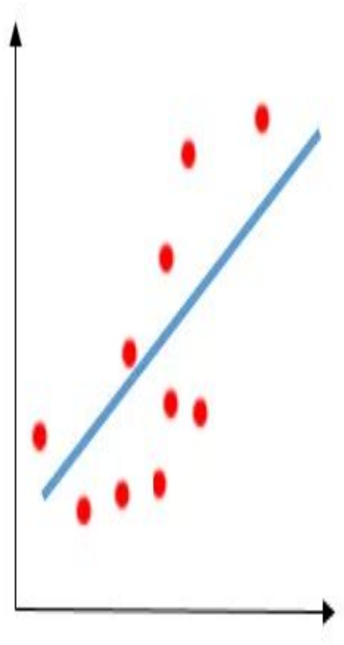
The relation between weights and neuron inputs is always linear if we do not make use of an activation function.

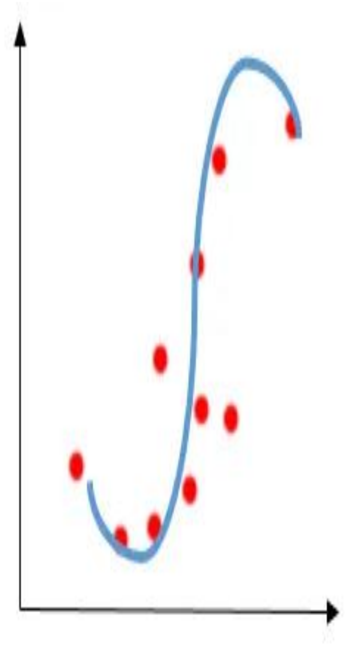
Thus, the output is a simple linear transformation of these latters.

When only linear relationship between variables are addressed, the predicted model will be unsuitable for complex problems such as computer vision and nature language processing.

Here, the use of an activation function that provides the non-linearity becomes a must.

This enables us to build more complex models that fit the real-world problems.





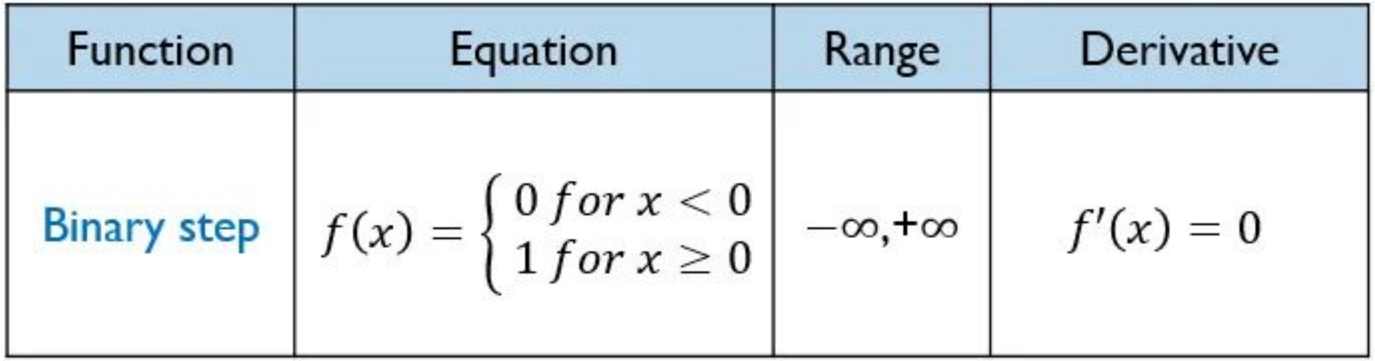
There are 3 main types of activation functions, namely binary step, linear and non-linear. For each of these types, I will present the equation of the function as well as its plot and range. Its derivative is also specified since it is used during the backpropagation process while minimizing the loss function.

1- Binary step function

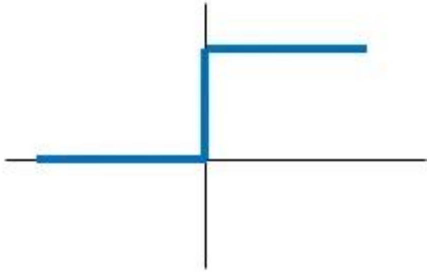
Equation, range, derivative and plot of binary step activation function

It is a threshold-based activation function.

It only supports binary classification. In other terms, it does not allow multi-value outputs.



Function	Equation	Range	Derivative
Binary step	$f(x) = \begin{cases} 0\ for\ x < 0 \\ 1\ for\ x \geq 0 \end{cases}$	$-\infty,+\infty$	$f'(x) = 0$

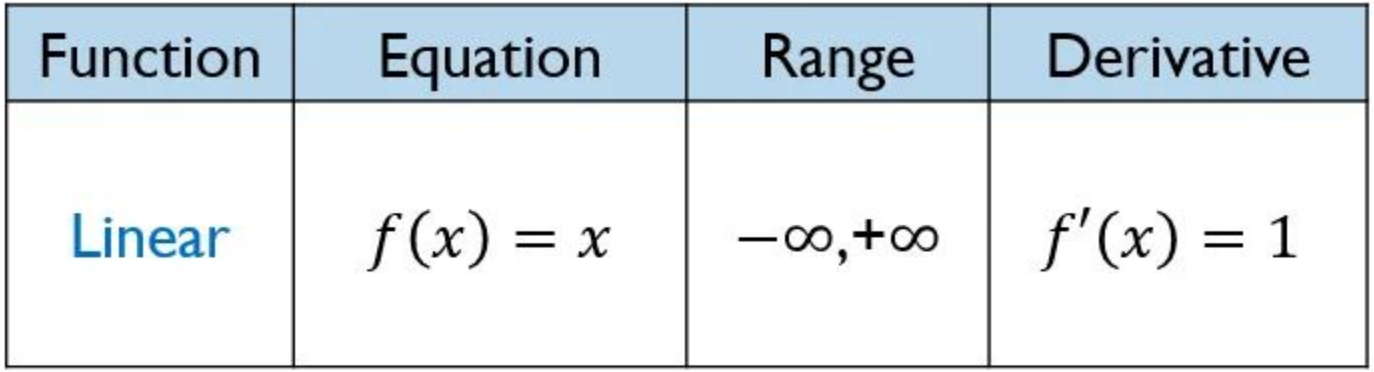


Linear function

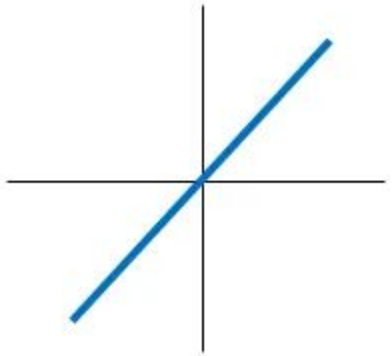
It is used for simple linear regression model.

Since the derivative is a constant, it is not possible to use backpropagation to train the model and understand which weights can provide a better prediction.

It is unable to capture complex patterns no matter the depth of the neural network. Typically, all layers collapse into one as if the network turns into just one layer : the output layer is a linear function of the input one.



Function	Equation	Range	Derivative
Linear	$f(x) = x$	$-\infty,+\infty$	$f'(x) = 1$



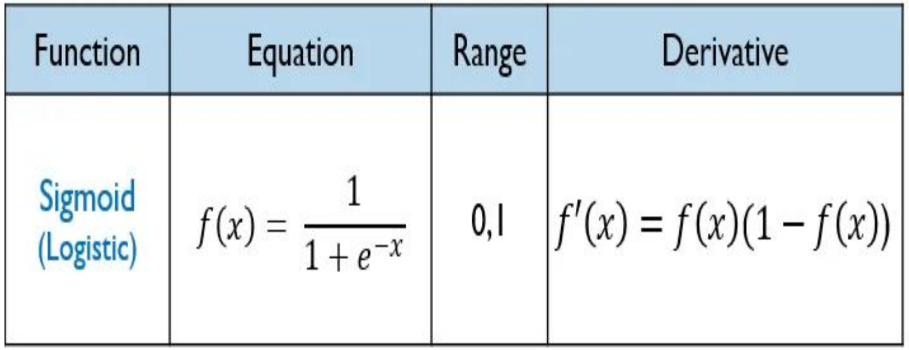
Non-linear function

This type of activation functions is the most commonly used one in neural networks since it introduces, as its name indicates, the non-linearity that we search to better capture the patterns in the data.

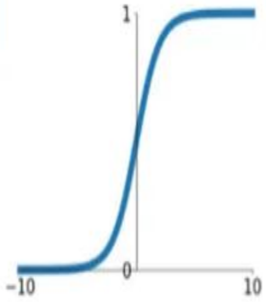
The major non-linear activation functions are as described below :

3.1. Sigmoid/Logistic

- It normalizes the output of the neuron to a range between 0 and 1, giving the probability of the input value. This makes the sigmoid useful for output neurons of classification-aimed neural networks.
- It is highly compute-intensive since it requires computation of an exponent, which makes the convergence of the network slower.
- It suffers from saturation problem. A neuron is considered as saturated if it reaches its maximum or minimum value (Ex. **Sigmoid : f(x) = 0 or 1),** so that its derivative (Ex. Sigmoid : f(x)(1- f(x)) = 0) is equal to 0. In that case, there is no update in weights. The gradient of the loss function with respect to weights consequently vanishes till it goes down to 0. This phenomenon is known as vanishing gradient that causes poor learning for deep networks.
- It is not a zero-centered function. So, the gradient of all the weights connected to the same neuron is either positive or negative. During update process, these weights are only allowed to move in one direction, i.e. positive or negative, at a time. This makes the loss function optimization harder.

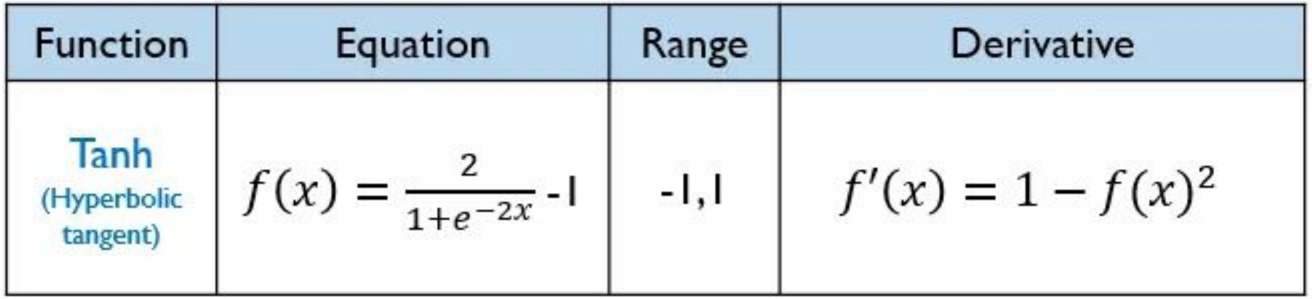


Function	Equation	Range	Derivative
Sigmoid (Logistic)	$f(x) = \frac{1}{1+e^{-x}}$	0,1	$f'(x) = f(x)(1 - f(x))$

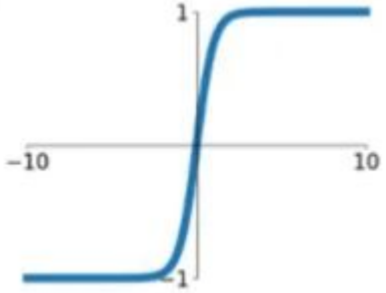


3.2. Tanh (Hyperbolic tangent)

It normalizes the output of the neuron to a range between -1 and 1.
Unlike Sigmoid, it is a zero-centered function so that the optimization of the loss function becomes easier.
As for Sigmoid, Tanh is highly compute-intensive and suffers from saturation problem and thus vanishing gradient.
In fact, when the neuron reaches the minimum or maximum value of its range, that respectively correspond to -1 and 1, its derivative is equal to 0.

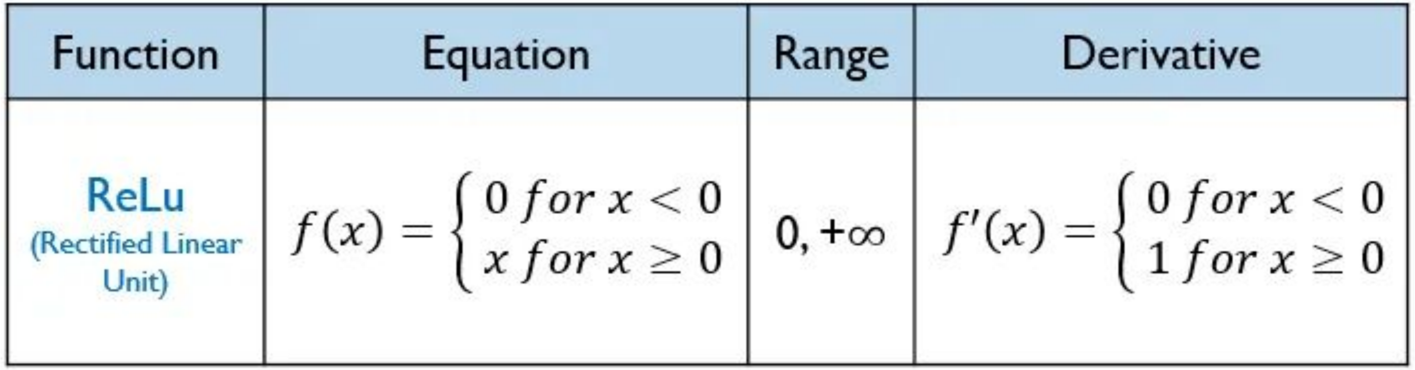


Function	Equation	Range	Derivative
Tanh (Hyperbolic tangent)	$f(x) = \frac{2}{1+e^{-2x}} - 1$	-1,1	$f'(x) = 1 - f(x)^2$

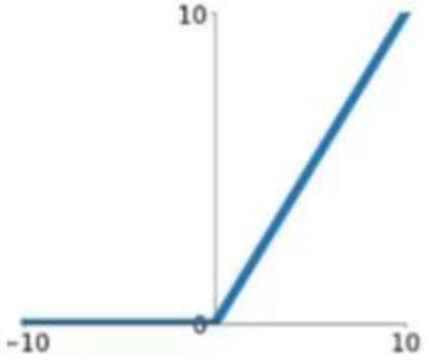


3.3. ReLu (Rectified Linear Unit)

- It is the most used activation function.
- It is easy to compute so that the neural network converges very quickly.
- As its derivative is not 0 for the positive values of the neuron (f′(x)=1 for x ≥ 0), ReLu does not saturate and no dead neurons are reported. Saturation and vanishing gradient only occur for negative values that, given to ReLu, are turned into 0.
- It is not a zero-centered function.



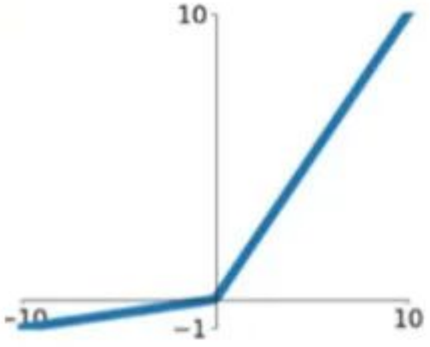
Function	Equation	Range	Derivative
ReLu (Rectified Linear Unit)	$f(x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$	0, +∞	$f'(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$



3.4. Leaky ReLu

- In an attempt to solve the dying ReLu at negative values, Leaky ReLu introduces a small slope. Having the negative values scaled by α enables their corresponding neurons to “stay alive”. The Leaky ReLu is the appellation of the activation function when α = 0.01. It is known as Randomized ReLu if α is equal to any small value other than 0.01.
- It is easy to compute.
- It is close to zero-centered function.

Function	Equation	Range	Derivative
Leaky ReLu	$f(x) = \begin{cases} \alpha x \; for \; x < 0 \\ x \;\; for \; x \geq 0 \end{cases}$	-∞, +∞	$f'(x) = \begin{cases} \alpha \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$



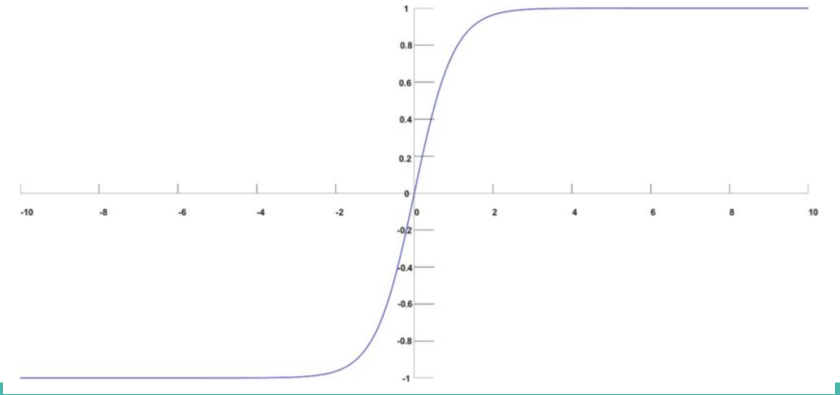
α =0.01

Softmax

The Softmax Function is an activation function used in machine learning and deep learning, particularly in multi-class classification problems.

Its primary role is to transform a vector of arbitrary values into a vector of probabilities.

The sum of these probabilities is one, which makes it handy when the output needs to be a probability distribution.



Softmax

Softmax comes into play in various machine learning tasks, particularly those involving multi-class classification.

It gives the probability distribution of multiple classes, making the decision-making process straightforward and effective.

By converting raw scores to probabilities, it not only provides a value to be worked with but also brings clarity to interpreting results.

Who can utilize the Softmax Function?
The Softmax Function is utilized by data scientists, machine learning engineers, and deep learning practitioners.

It's a fundamental tool in their toolkit, especially when they're working with models that predict the probability of multiple potential outcomes, as in the case of neural network classifiers.

Softmax

When is the Softmax Function Used?
The Softmax Function comes into its own when dealing with multi-class classification tasks in machine learning. In these scenarios, you need your model to predict one out of several possible outcomes.

Softmax is typically applied in the final layer of a neural network during the training phase, converting raw output scores from previous layers into probabilities that sum up to one.

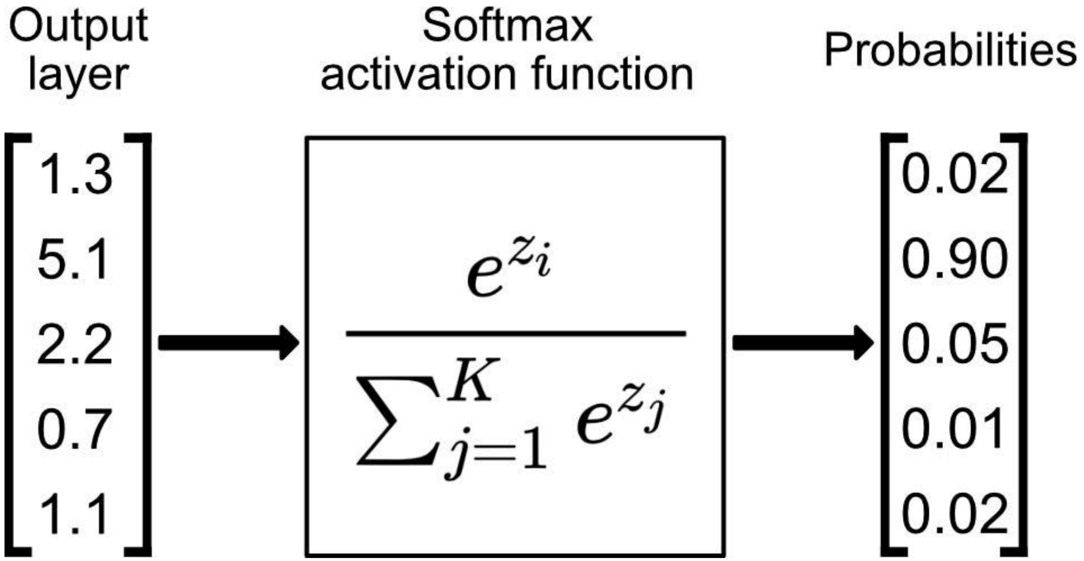
Where is the Softmax Function Implemented?
In practice, the Softmax Function finds its application in various fields where machine or deep learning models are used for prediction.

This could be anything from identifying objects in an image, predicting sentiment in a block of text, or even predicting market trends in finance.

Any field requiring a definitive class prediction based on multiple potential outcomes could benefit from the Softmax Function.

How does the Softmax Function work?

The Softmax function is a wonderful tool used predominantly in the field of machine learning and deep learning for converting a vector of numbers into a vector of probabilities.



$$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} \longrightarrow \boxed{\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}} \longrightarrow \begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

Advantages of the Softmax Function

Probabilistic Interpretation – Converts input values into a probability distribution, ensuring outputs sum to 1 for clear interpretation.

Ideal for Multi-Class Classification – Assigns probabilities to each class, helping identify the most likely class.

Works Well with Gradient Descent – Differentiable and supports gradient-based optimization for deep learning.

Numerical Stability – Techniques like log-softmax improve stability in large-scale computations.

Enhances Generalization – Boosts model confidence in correct predictions while reducing overconfidence in incorrect ones.

Supports Real-World Decision Making – Outputs class probabilities, providing useful uncertainty estimates for decision-making applications.

Activation Function	Pros	Cons	Example (Python)
Sigmoid	Outputs between (0,1) → good for probability-based tasks. Smooth and differentiable.	Vanishing gradient → slows down deep networks. Not zero-centered → inefficient weight updates.	python def sigmoid(x): return 1 / (1 + np.exp(-x)) print(sigmoid(np.array([-2, 0, 2])))
Tanh (Hyperbolic Tangent)	Zero-centered → better gradient updates. Stronger gradients than sigmoid.	Vanishing gradient for large or small values.	python def tanh(x): return np.tanh(x) print(tanh(np.array([-2, 0, 2])))
ReLU (Rectified Linear Unit)	Computationally efficient. Helps solve vanishing gradient issue.	Dying ReLU problem (neurons output 0 for negative values).	python def relu(x): return np.maximum(0, x) print(relu(np.array([-2, 0, 2])))
Leaky ReLU	Prevents dying ReLU problem. Small gradient for negative values.	Slightly more computationally expensive than ReLU.	python def leaky_relu(x, alpha=0.01): return np.where(x > 0, x, alpha * x) print(leaky_relu(np.array([-2, 0, 2])))
Softmax	Converts values into probabilities (sums to 1). Useful for multi-class classification.	Sensitive to large input values → numerical instability.	python def softmax(x): exps = np.exp(x - np.max(x)) return exps / exps.sum() print(softmax(np.array([1, 2, 3])))