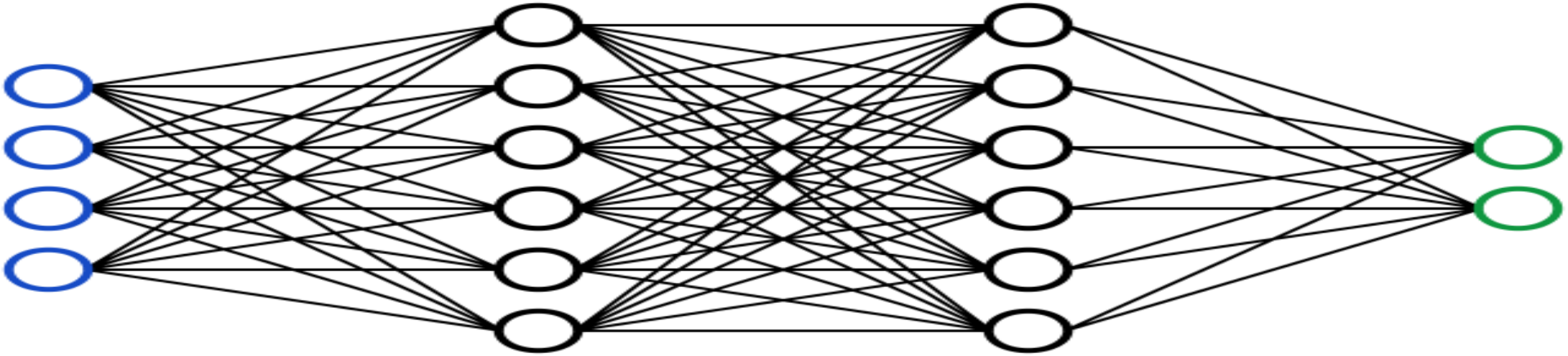


# Artificial Neural Network (ANN)

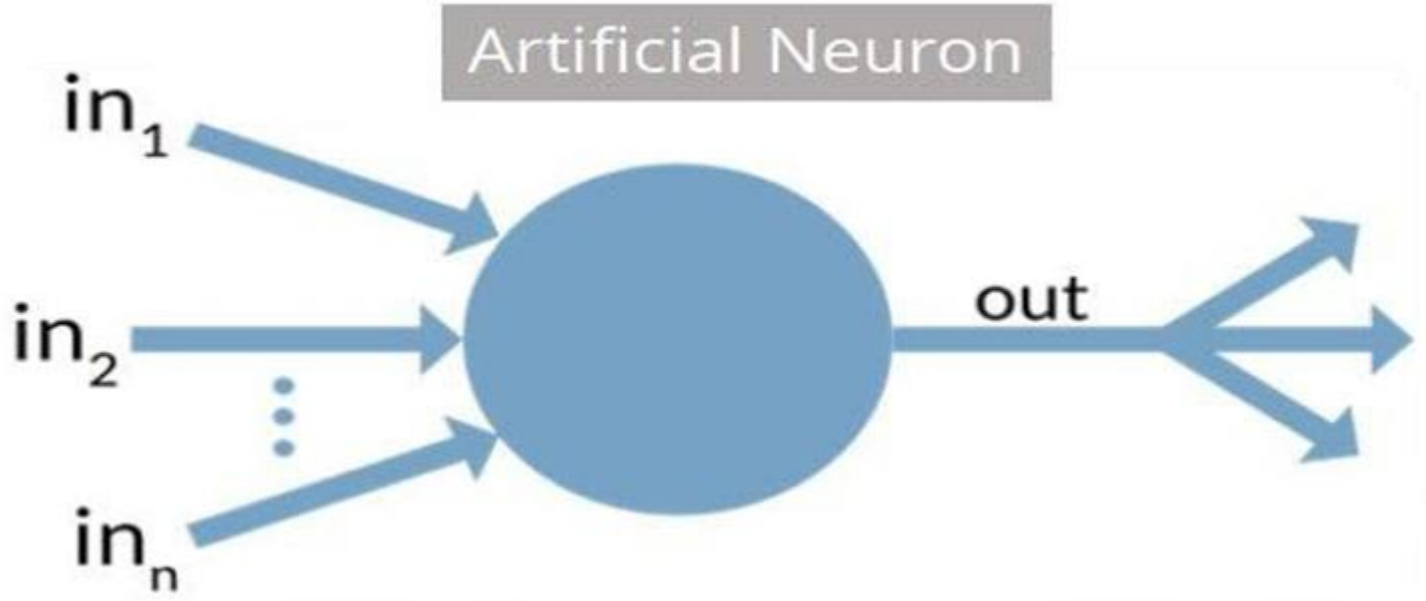
## Lecture3

Dina El-Manakhly, Ph. D.

dina_almnakhly@science.suez.edu.eg

# What is an artificial neuron?

- An artificial neuron is a mathematical function based on a model of biological neurons, where each neuron takes inputs, weighs them separately, sums them up and passes this sum through a nonlinear function to produce output.



# What is a Perceptron?

- A **perceptron** is a specific type of artificial neuron with a simpler structure.
- A **perceptron** specifically performs binary classification.

In machine learning, binary classification is a supervised learning algorithm that categorizes new observations into one of **two** classes.

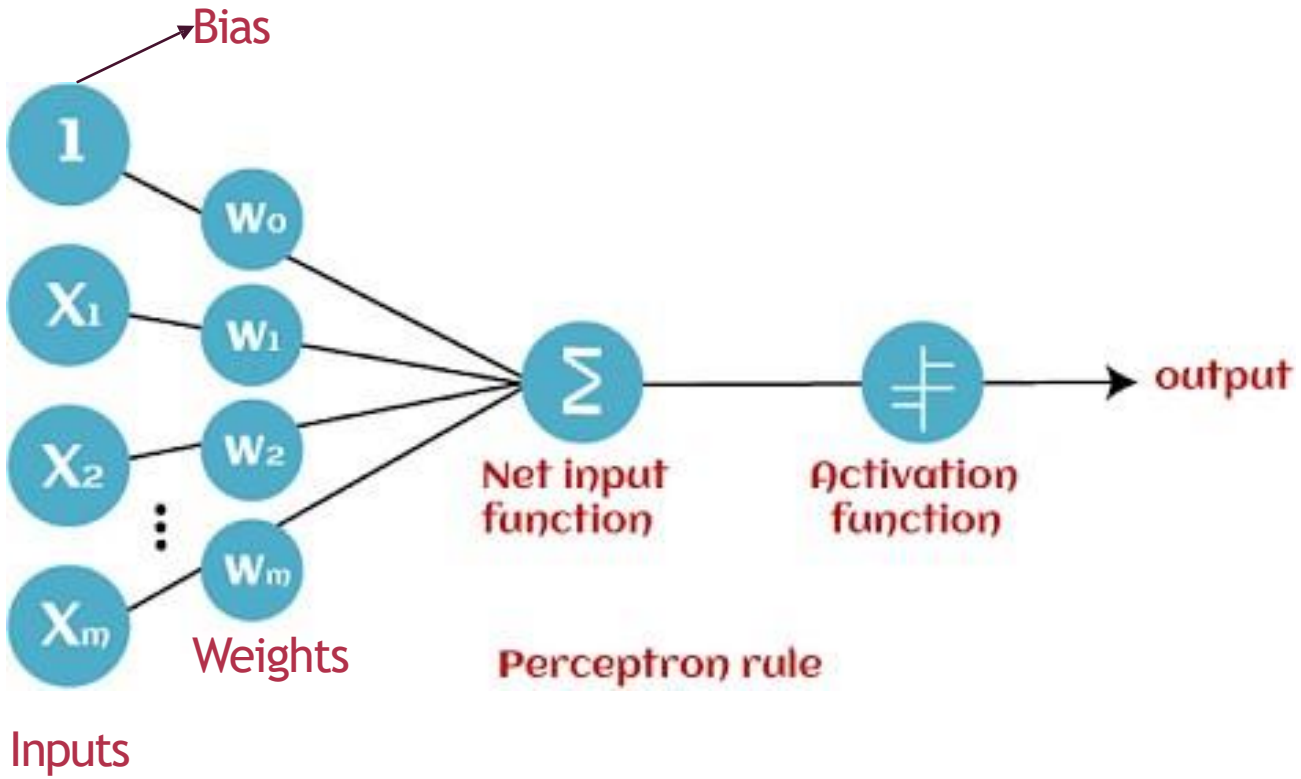
| Application | Observation | 0 | 1 |
|---|---|---|---|
| Medical Diagnosis | Patient | Healthy | Diseased |
| Email Analysis | Email | Not Spam | Spam |
| Financial Data Analysis | Transaction | Not Fraud | Fraud |
| Marketing | Website visitor | Won't Buy | Will Buy |
| Image Classification | Image | Hotdog | Not Hotdog |

- A **perceptron** model begins with multiplying all input values and their weights, then adds these values to create the weighted sum. Further, this weighted sum is applied to the activation function to obtain the desired output (a binary output (0 or 1) based on whether the weighted sum exceeds a certain threshold).
- A **Perceptron** have a specific learning algorithm called the perceptron learning rule, which updates the weights based on misclassified instances.

# Basic components of perceptron

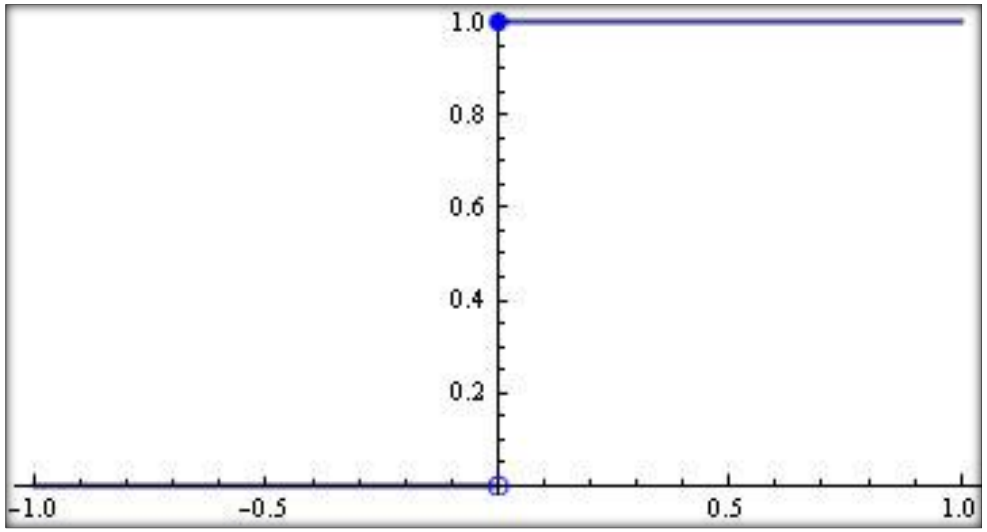
**Main parameters:**

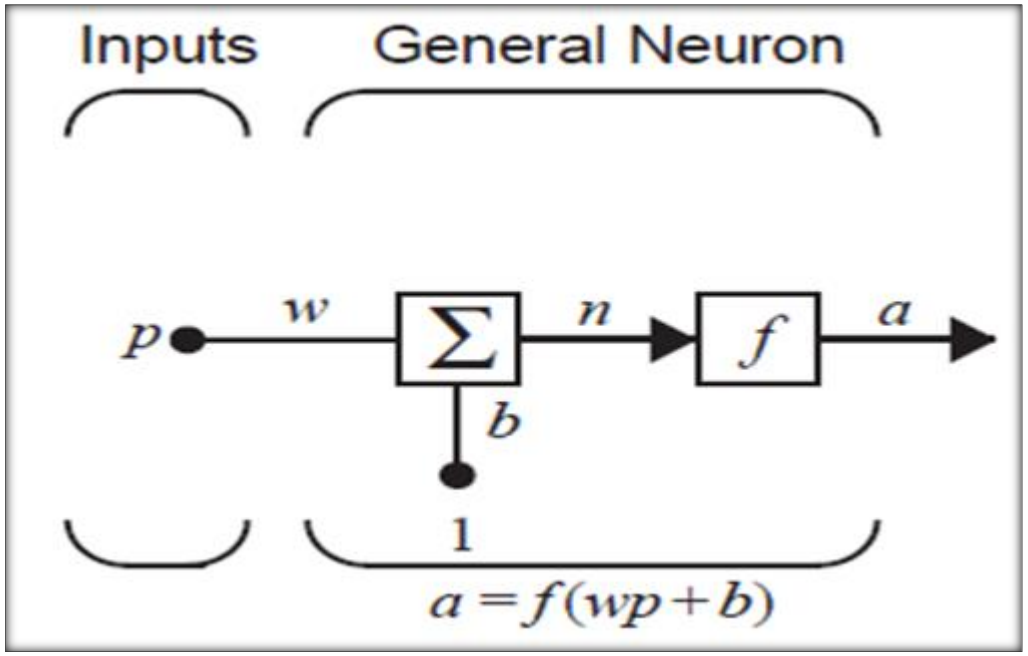
1) Input values.
2) Weights.
3) Bias (1 or -1).
4) Net sum
5) Activation function.



# How does perceptron work?

Step Function




If, for instance, $w = 3$, $p = 2$ and $b = -1.5$, then

$$a = f(3(2) - 1.5) = f(4.5) = 1$$

# Types of perceptron models

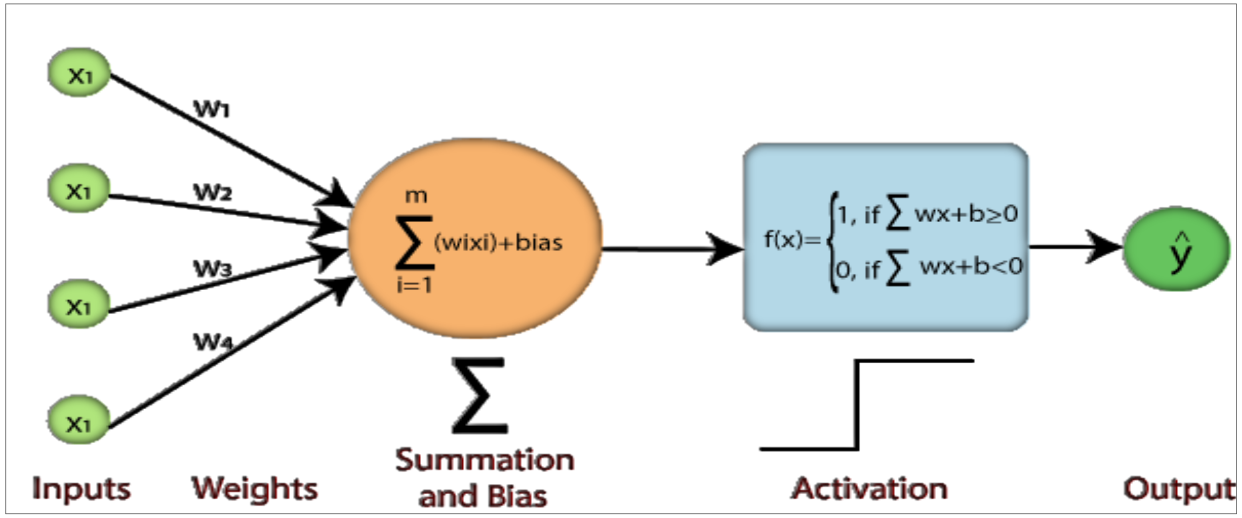
1. Single Layer Perceptron model

2. Multi-Layered Perceptron model

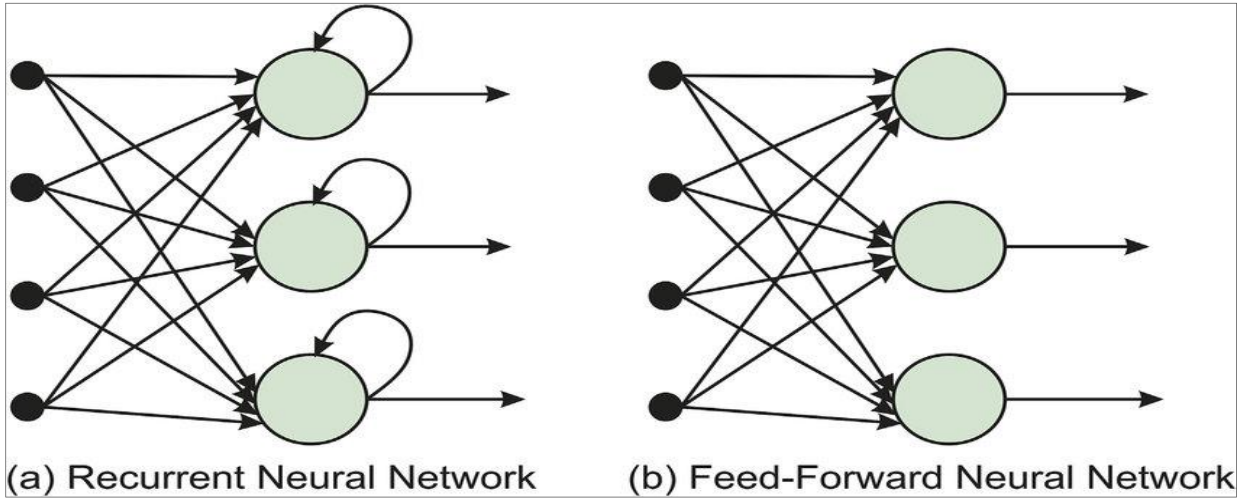
# Types of perceptron models:
# 1) Single layer perceptron (**SLP**)

- A single layer perceptron (**SLP**):

One of the easiest ANN types consists of a feed-forward neural network.




"Feedforward (FNN)" refers to a type of neural network architecture where information flows in one direction, from input to output, without any feedback loops or cycles.



# Types of perceptron models:
# 1) Single layer perceptron (**SLP**)

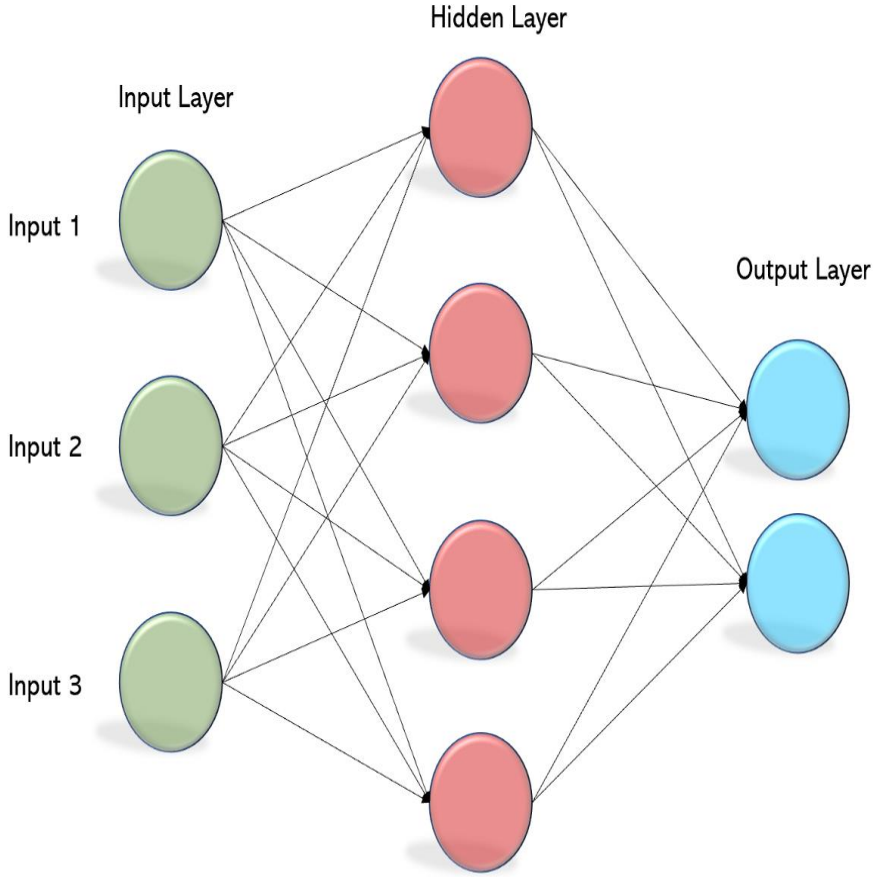
- The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes (1, 0). A Single-layer perceptron can learn only linearly separable patterns.
- The single layer perceptron does not have a priori knowledge, so the initial weights are assigned randomly. SLP sums all the weighted inputs and if the sum is above the threshold (some predetermined value), SLP is said to be activated (output=1).
- The input values are presented to the perceptron, and if the predicted output is the same as the desired output, then the performance is considered satisfactory and no changes to the weights are made. However, if the output does not match the desired output, then the weights need to be changed to reduce the error.

# Types of perceptron models:
# 2) Multi-layer perceptron (MLP)

- Multi layer perceptron (MLP) is a feed forward neural network. At least, it consists of three types of layers: input layer, output layer and hidden layer.

  1) Input layer receives the input signal to be processed.
  2) The required task such as prediction and classification is performed by the output layer.
  3) An arbitrary number of hidden layers that are placed in between the input and output layer are the computational engine of the MLP.

- A multilayer perceptron model has a greater processing power and can process linear and non-linear patterns. Further, it also implements logic gates such as AND, OR, XOR, XNOR, and NOR.



# Multi-layered perceptron advantages

- A multi-layered perceptron model can solve complex non-linear problems.
- It works well with both small and large input data.
- Helps us obtain the same accuracy ratio with big and small data.

# Multi-layered perceptron disadvantages

- In multi-layered perceptron model, computations are time-consuming and complex.
- It is tough to predict how much the dependent variable affects each independent variable.
  - Independent variables in a neural network are the input features.
  - Dependent variables in a neural network are the output predictions.
  - It is a challenge in understanding the precise relationship between the independent variables and the dependent variable
- The model functioning depends on the quality of training.

# Activation functions

- Activation functions are mathematical functions that introduce non-linearity into neural networks. They are applied to the output of each neuron (or node) in the network to introduce non-linear properties and enable the network to learn complex patterns in data. Here are some common activation functions used in neural networks:

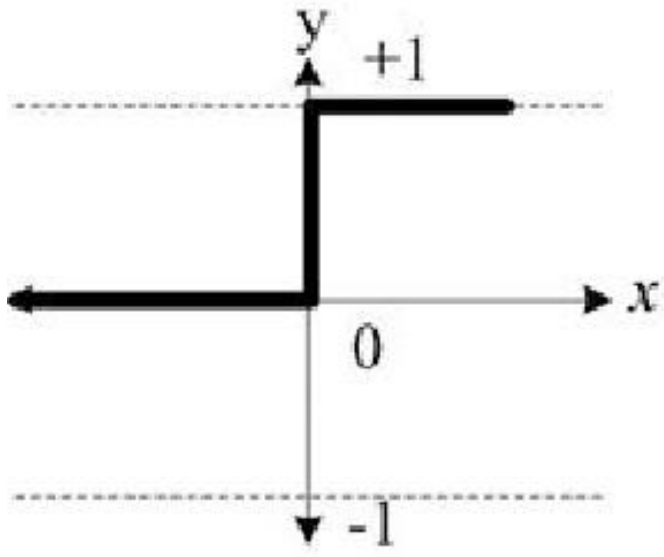
1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

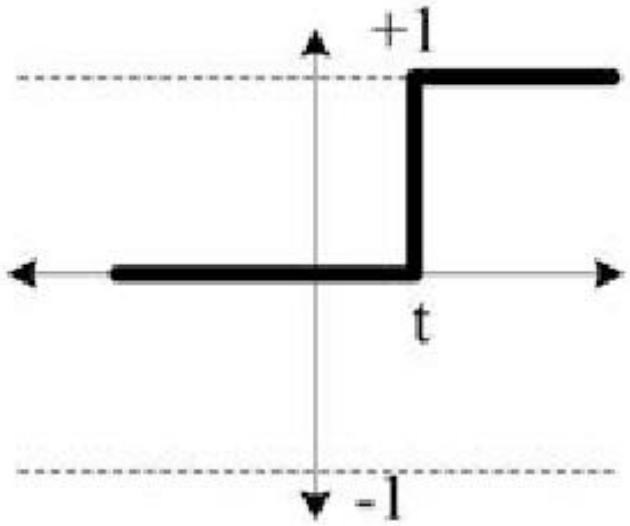
# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

# Activation functions of perceptron
# 1- Step Function

- The Step activation function is a simple threshold-based function that outputs 1 if the input is greater than or equal to a threshold, and 0 otherwise.




$Step_t(x)$= +1 if x>=0, else 0

$Step_t(x)$= +1 if x>=t, else 0

# Implementing the Step activation function in python

```
def step_function(x, threshold=0):
    if x >= threshold:
        return 1
    else:
        return 0

print(step_function(3, 2))
print(step_function(1, 2))
print(step_function(-1, 0))
print(step_function(0, 0))
```

# Plotting the Step activation function in python

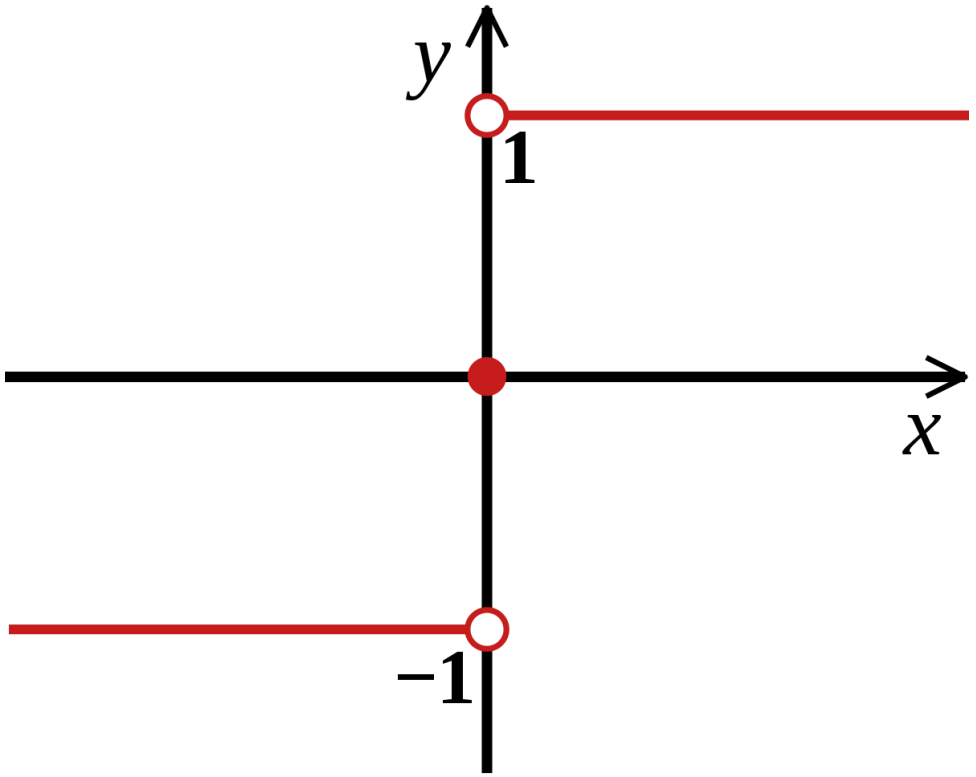
```
import numpy as np
import matplotlib.pyplot as plt
# Generate some input values
x = np.linspace(-5, 5,100)
y=[]
# Calculate the output of the function for each input value
for i in x :
 z=[step_function(i)]
 y.append(z)

# Plot the function
plt.plot(x, y)
plt.title("Step Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.show()
```

# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

# Activation functions of perceptron
# 2- Sign Function




Sign(x) = +1 if x>= 0, else -1

# Implementing the Sign activation function in python

```
def sign_function(x, threshold=0):
    if x >= threshold:
        return 1
    else:
        return -1
```

# Plotting the Sign activation function in python

```
import numpy as np
import matplotlib.pyplot as plt
# Generate some input values
x = np.linspace(-5, 5,100)
y=[]
# Calculate the output of the function for each input value
for i in x :
 z=[sign_function(i)]
 y.append(z)

# Plot the function
plt.plot(x, y)
plt.title("sign Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.show()
```

# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

# Activation functions of perceptron
# 3- Sigmoid Function (2 Stages)

## Stage 1

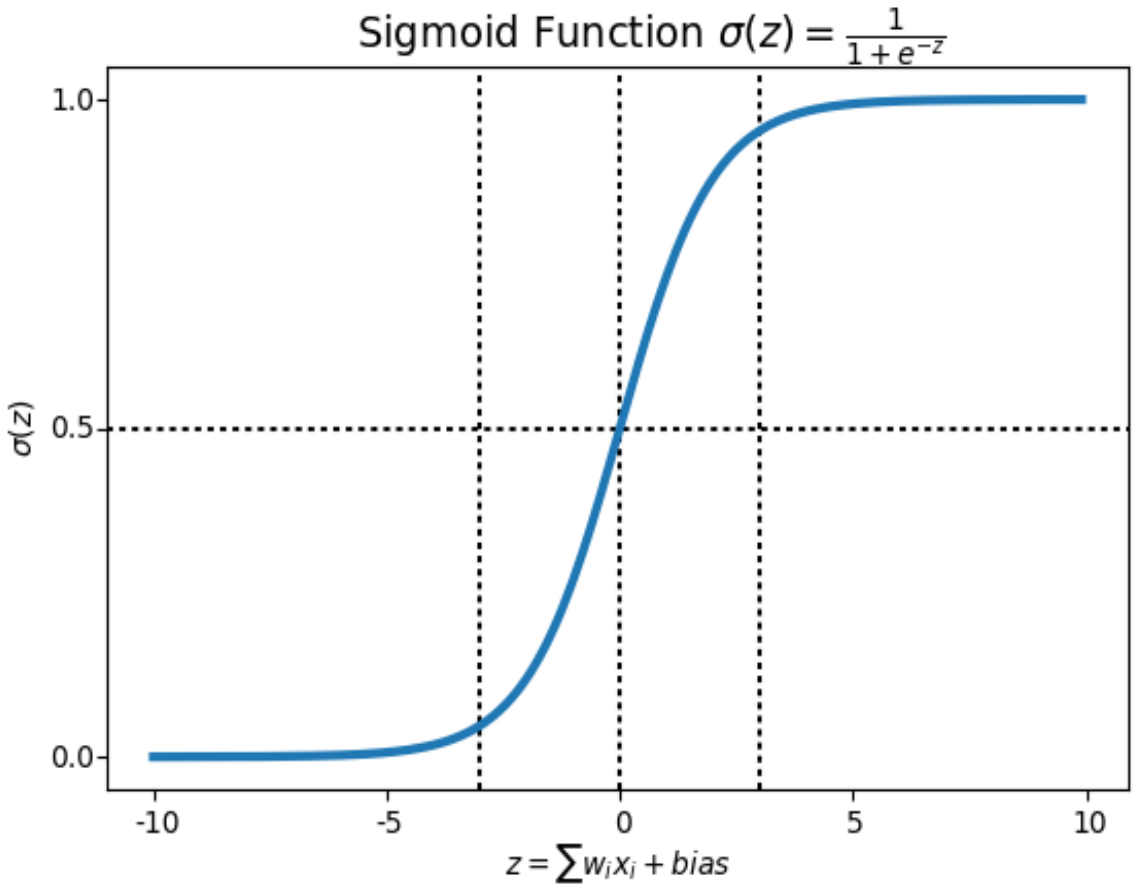
Net Sum

- Sigmoid(z)= 1/(1+$e^{-z}$)
- Sigmoid(z) ranges from 0 to 1

## Stage 2

- 0.5 is taken as a threshold
- if Sigmoid(z) < 0.5 we assume that it's value is 0, if Sigmoid(x) >= 0.5 then it's 1.

It is a function which is plotted as **'S'** shaped graph.

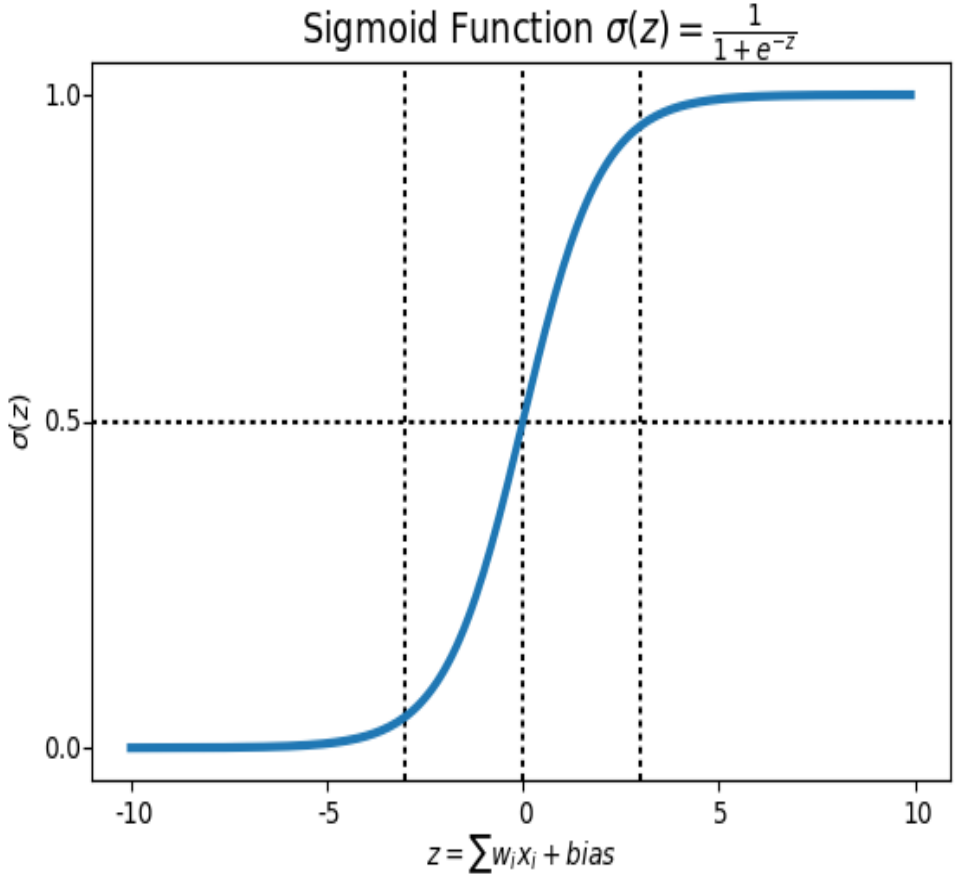


# Implementing the sigmoid activation function in python

## Stage 1

```
import numpy as np
def sig(x):
 return 1/(1 + np.exp(-x))
```

```
Applying Sigmoid Activation on (1.0) gives 0.7
Applying Sigmoid Activation on (-10.0) gives 0.0
Applying Sigmoid Activation on (0.0) gives 0.5
Applying Sigmoid Activation on (15.0) gives 1.0
Applying Sigmoid Activation on (-2.0) gives 0.1
```



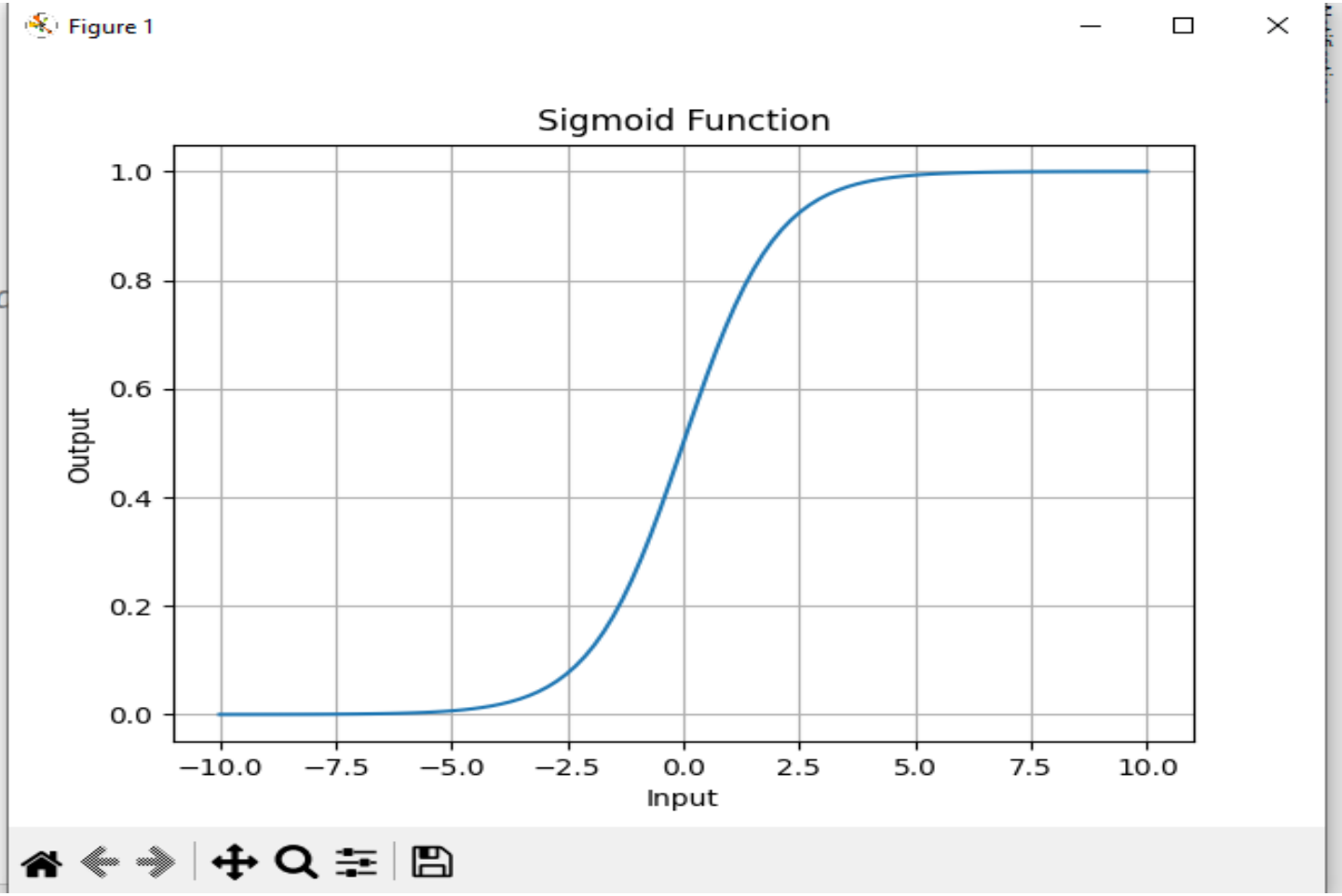
# Plotting the sigmoid activation function in python

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

x = np.linspace(-10, 10, 100)

# Compute sigmoid for each input va
y = sigmoid(x)

# Plot the sigmoid function
plt.plot(x, y)
plt.title('Sigmoid Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid()
plt.show()
```



# Implementing the Transfer_sigmoid activation function in python

## Stage 2

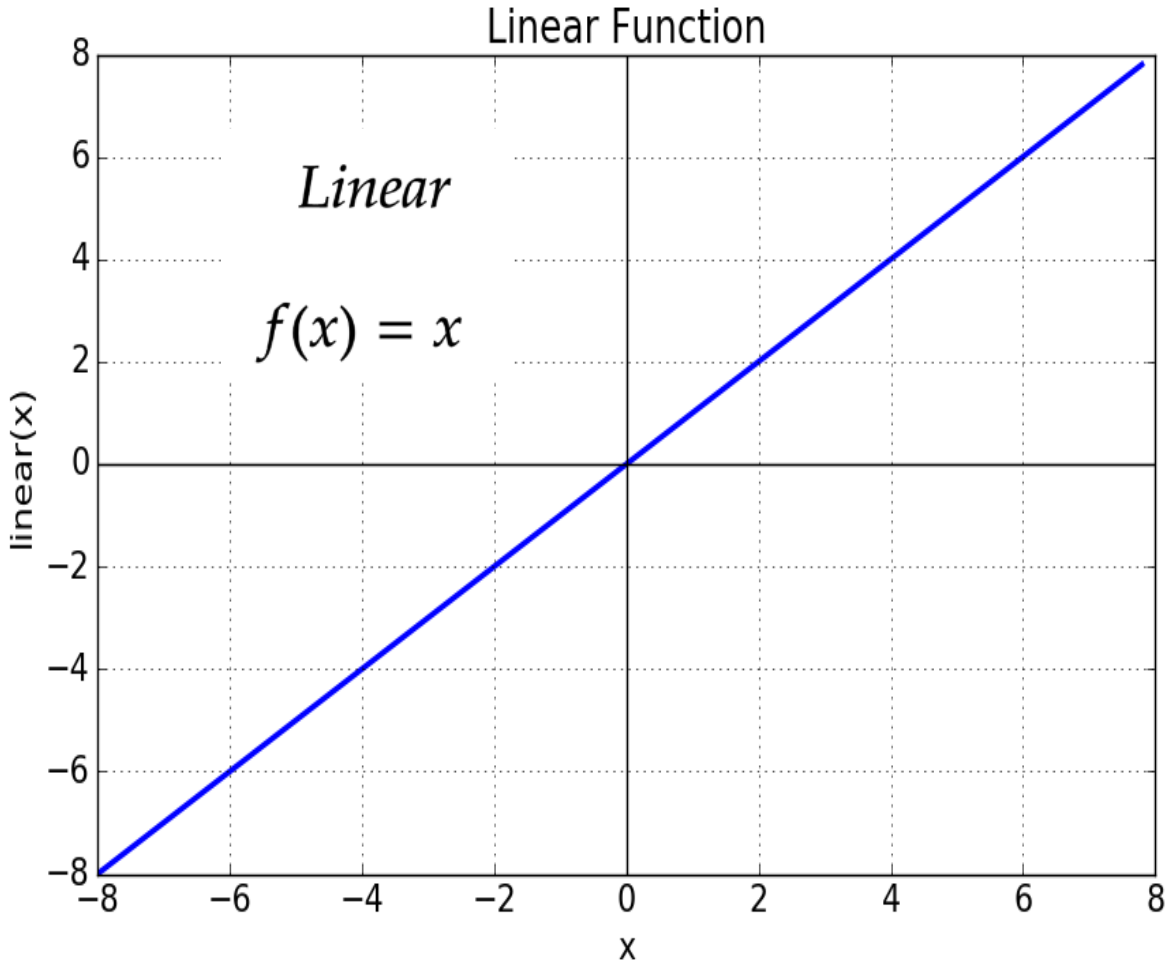
```
import numpy as np
#=============================================
def sigmoid(z):
    return Transfer_sigmoid_function(1 / (1 + np.exp(-z)))
#=============================================
def Transfer_sigmoid_function(x, threshold=0.5):
    if x >= threshold:
        return 1
    else:
        return 0
#=============================================
print(sigmoid(1))#0.7
print(sigmoid(-2))
print(sigmoid(-10))
```

# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

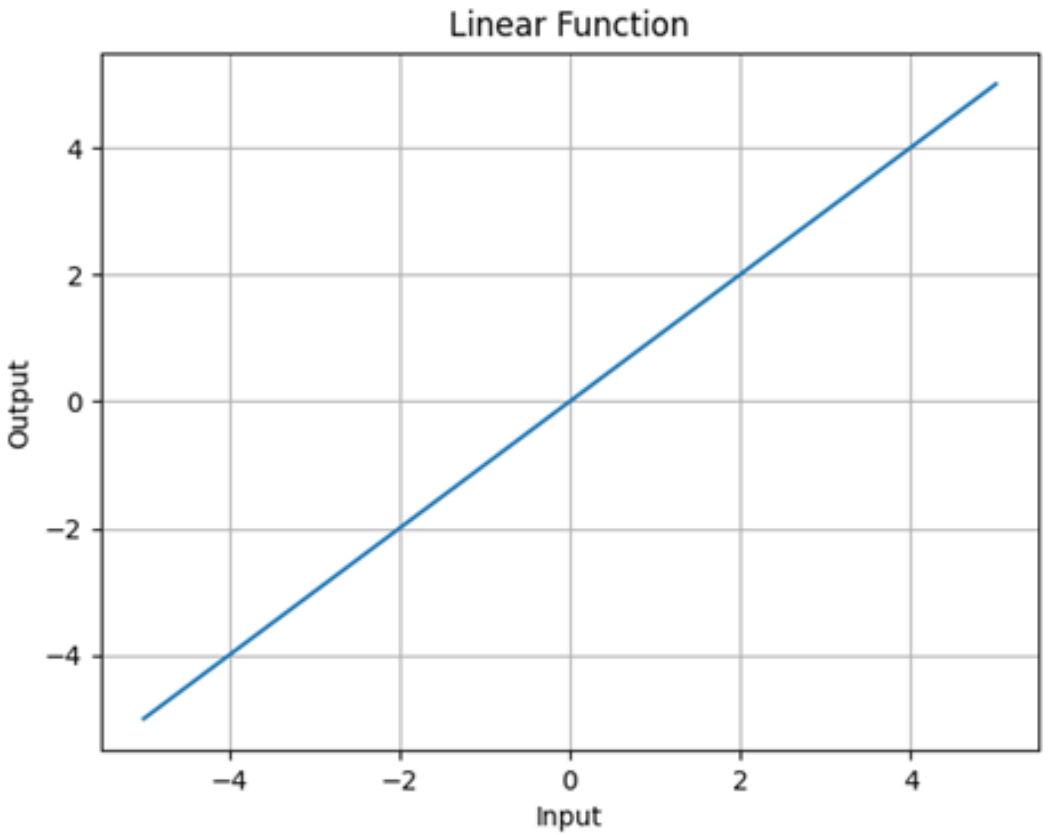
# Activation functions of perceptron
# 4- Linear Function

- The linear activation function, also known as "no activation," or "identity function", is where the activation is proportional to the input.
- The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.



# Implementing and plotting the linear activation function in python

```
def linear(x):
    return x
print(linear(7))
#=================================
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5, 5, 100)
y = linear(x)
plt.plot(x, y)
plt.title('Linear Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid()
plt.show()
```

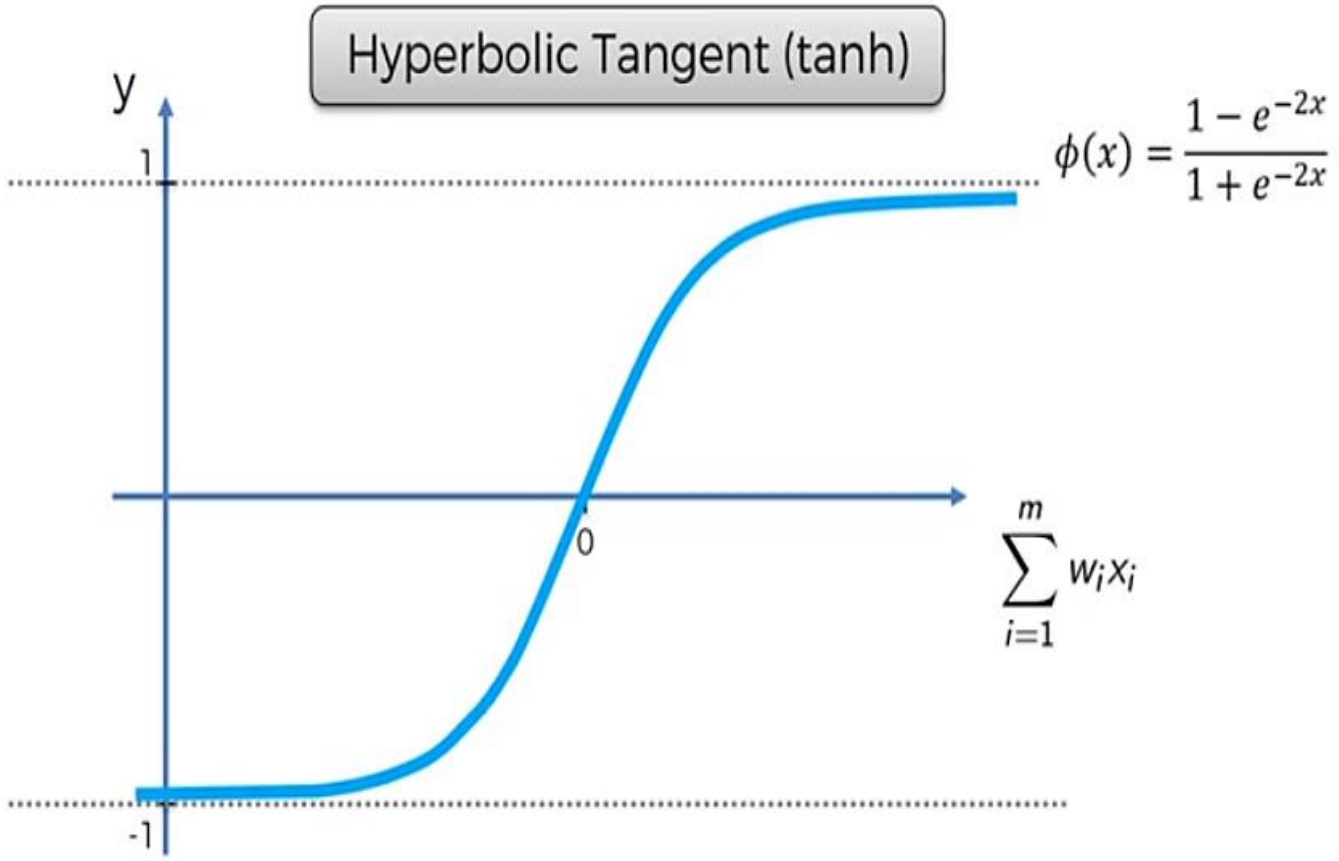


# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

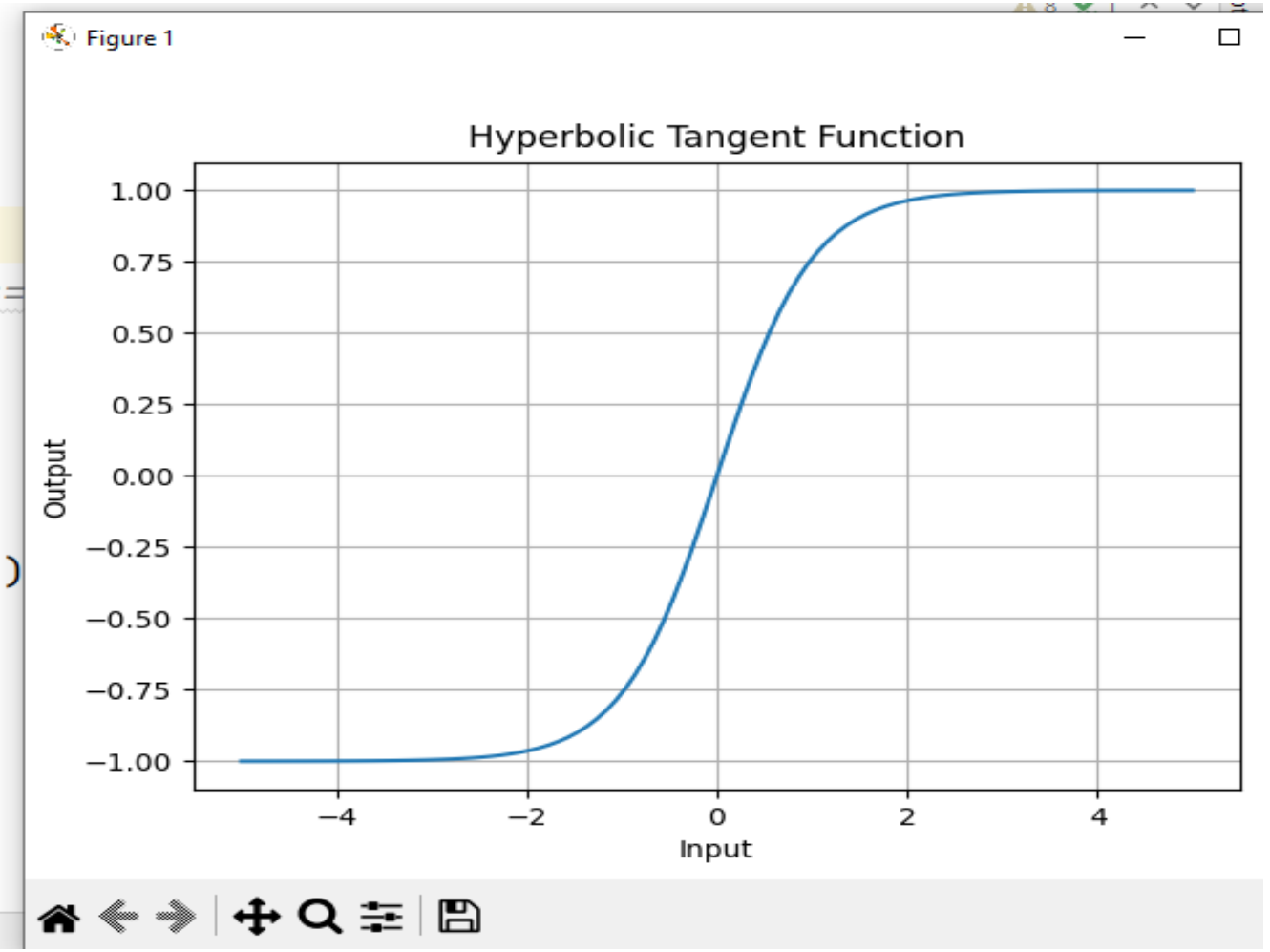
# Activation functions of perceptron
# 5- Tanh Function

- Tanh function (**Hyperbolic Tangent**) is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1.



# Implementing and plotting the Tanh activation function in python

```
import numpy as np
def tanh(x):
 return np.tanh(x)
print(tanh(-2))#-0.94
#=======================================
import matplotlib.pyplot as plt
x = np.linspace(-5, 5, 100)
y = tanh(x)
plt.plot(x, y)
plt.title('Hyperbolic Tangent Function')
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid()
plt.show()
```

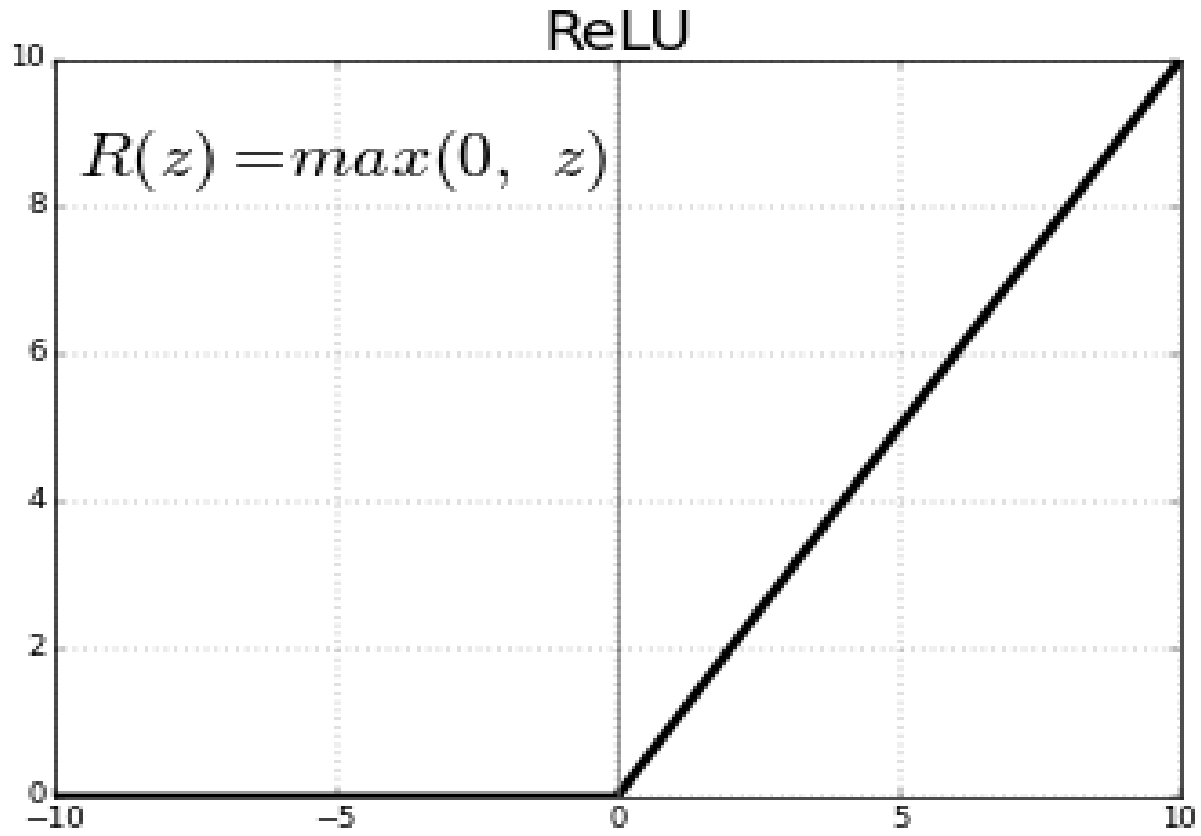


# Activation functions

1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

# Activation functions of perceptron
# 6- Relu Function

- The ReLu activation function returns 0 if the input is negative otherwise return the input as it is.

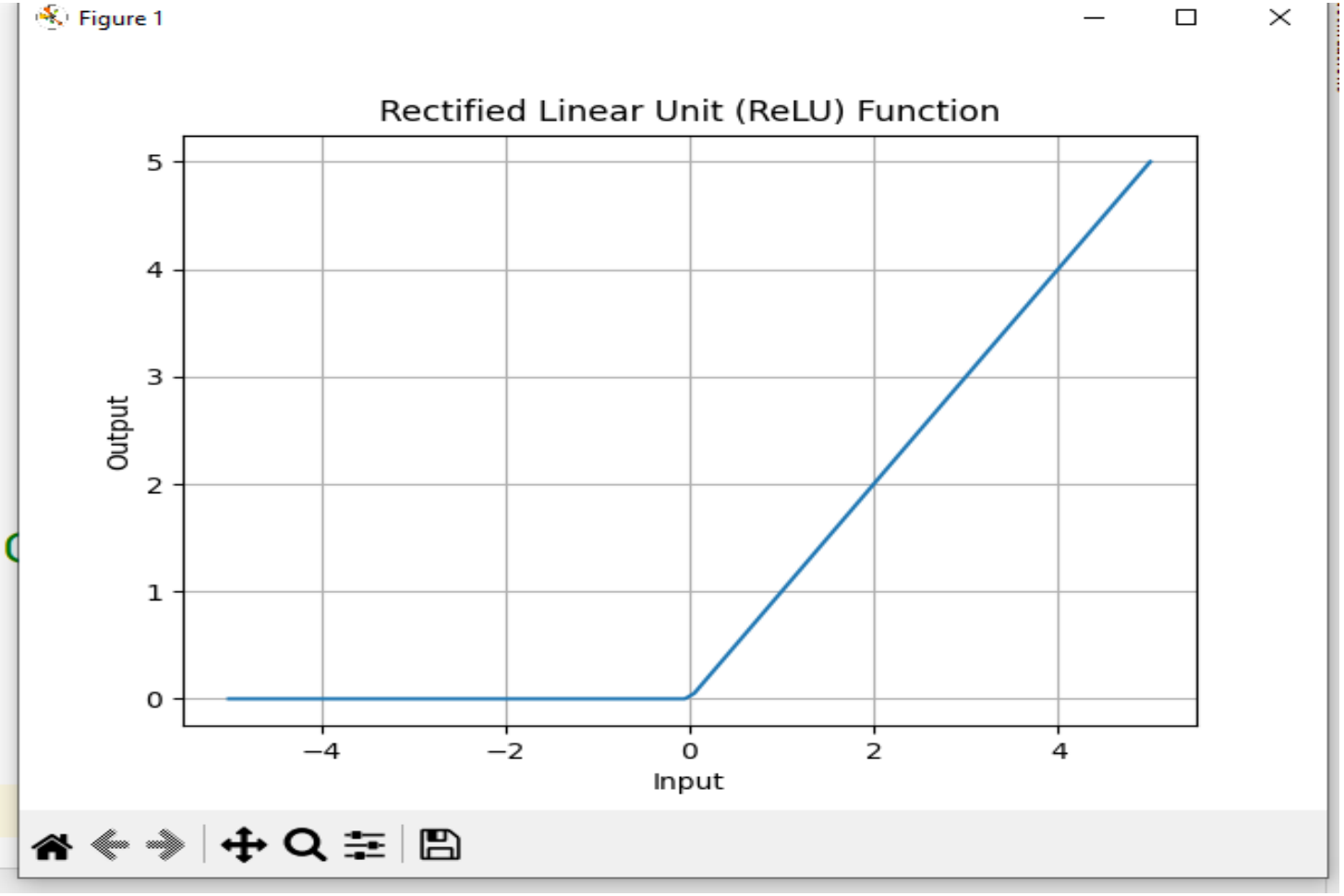


# Implementing and plotting the Relu activation function in python

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

def relu(x):
    return np.maximum(0, x)

x = np.linspace(-5, 5, 100)
y = relu(x)
# Plot the ReLU function
plt.plot(x, y)
plt.title('Rectified Linear Unit (
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid()
plt.show()
```

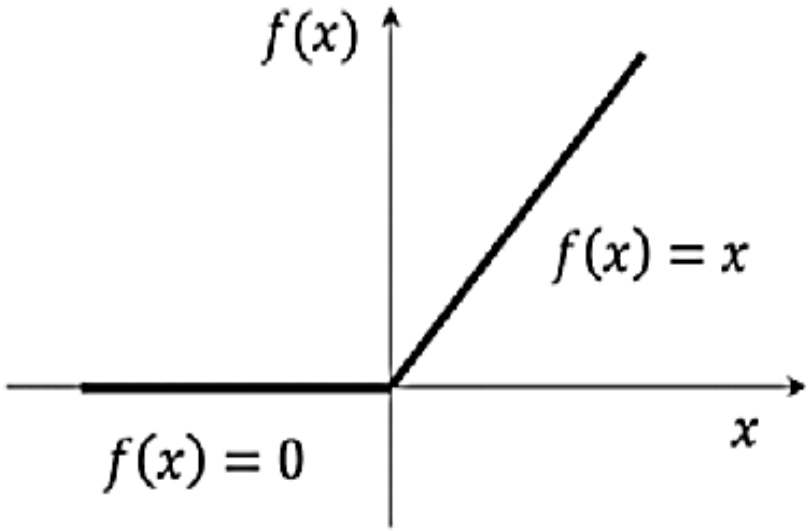


# Activation functions

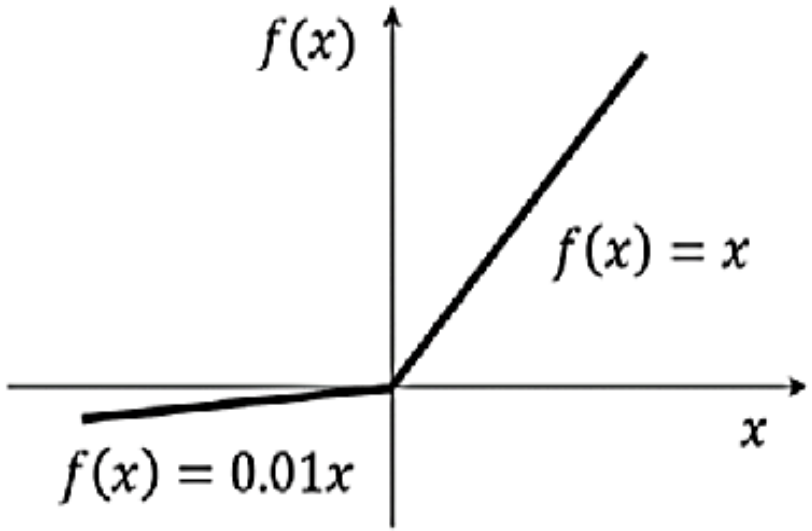
1. Step function
2. Sign Function
3. Sigmoid Function
4. Linear Function
5. Tanh Function
6. Relu Function
7. Leaky Relu Function

# Activation functions of perceptron
# 7- Leaky Relu Function

- The leaky ReLu gives an extremely small linear component of x to negative inputs.




*ReLU activation function*

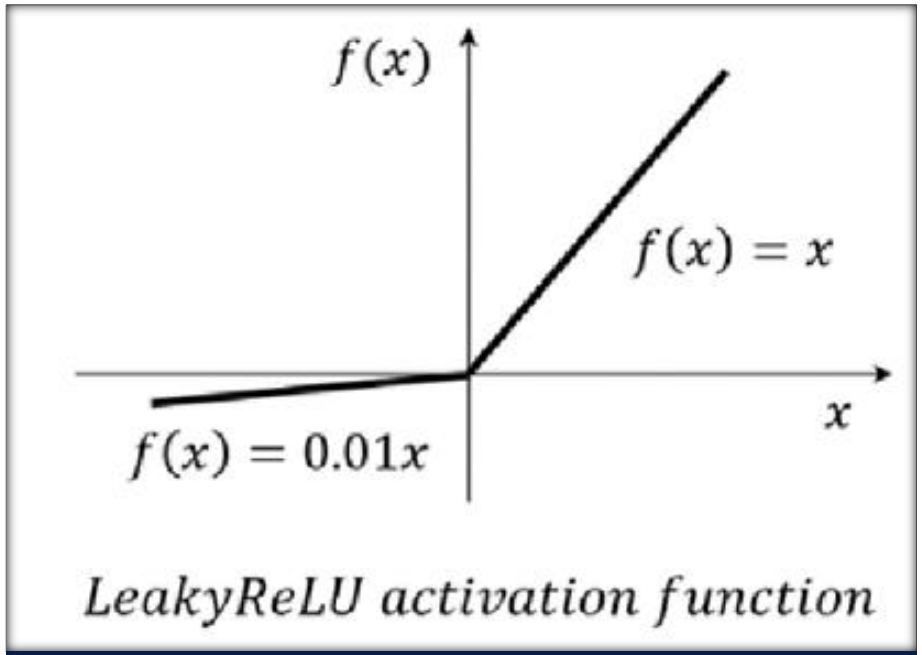



*LeakyReLU activation function*

# Implementing the Leaky Relu activation function in python

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


def leaky_relu(x, alpha=0.01):
    if x>=0:
        return x
    else:
        return alpha*x
```




*LeakyReLU activation function*

```
Applying Leaky Relu on (1.0) gives 1.0
Applying Leaky Relu on (-10.0) gives -0.1
Applying Leaky Relu on (0.0) gives 0.0
Applying Leaky Relu on (15.0) gives 15.0
Applying Leaky Relu on (-20.0) gives -0.2
```

# Plotting the Leaky Relu activation function in python

```
# Generate some input values
x = np.linspace(-5, 5, 100)
y=[]
for i in x:
 z=leaky_relu(i)
 y.append(z)

# Plot the Leaky ReLU function
plt.plot(x, y)
plt.title('Leaky Rectified Linear
plt.xlabel('Input')
plt.ylabel('Output')
plt.grid()
plt.show()
```