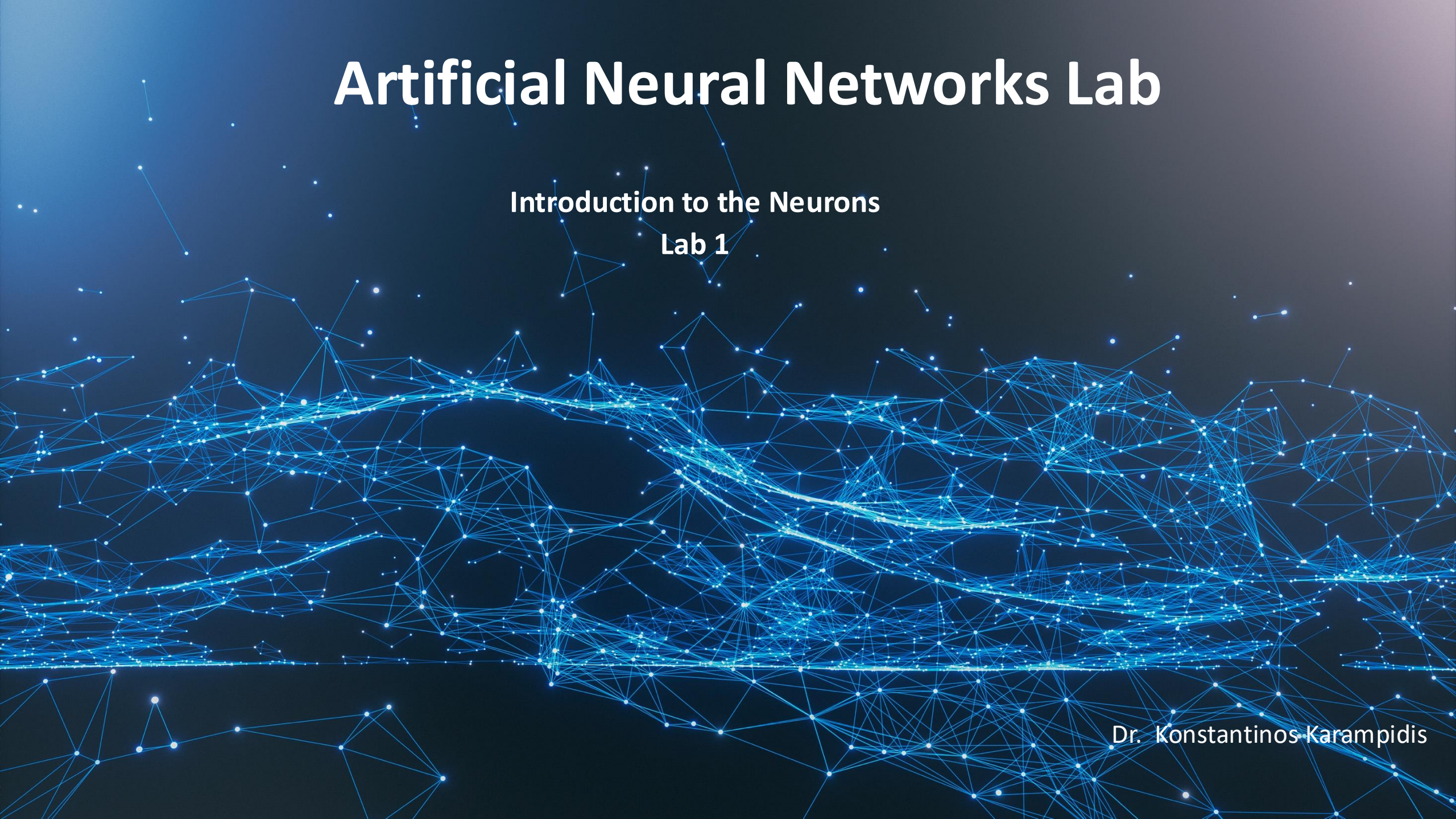
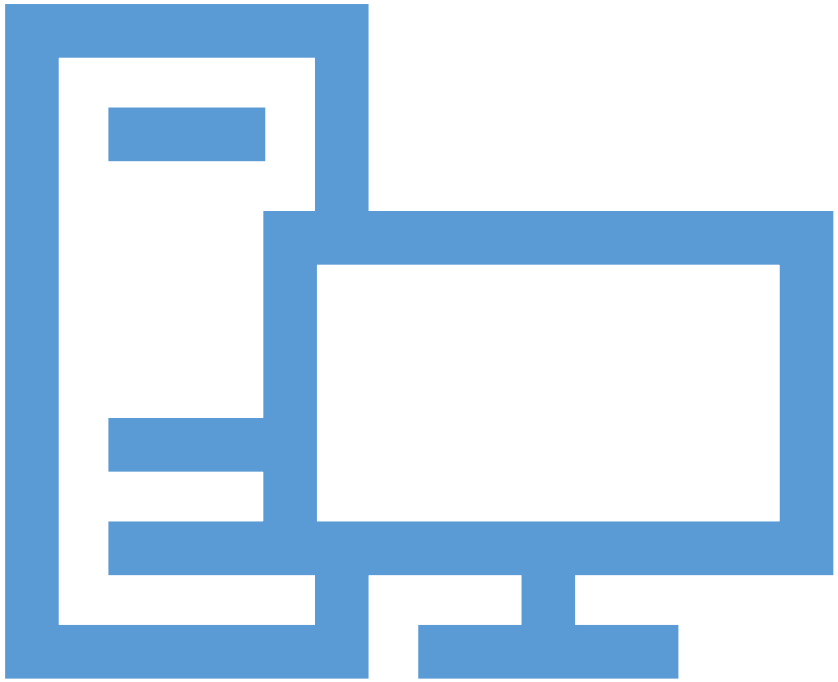
# Artificial Neural Networks Lab

## Introduction to the Neurons

## Lab 1



Dr. Konstantinos Karampidis



## Artificial Neural Networks - Definition

- An attempt to mathematically simulate the function of the human brain.
- It is a computational model that is inspired by biological neural networks.

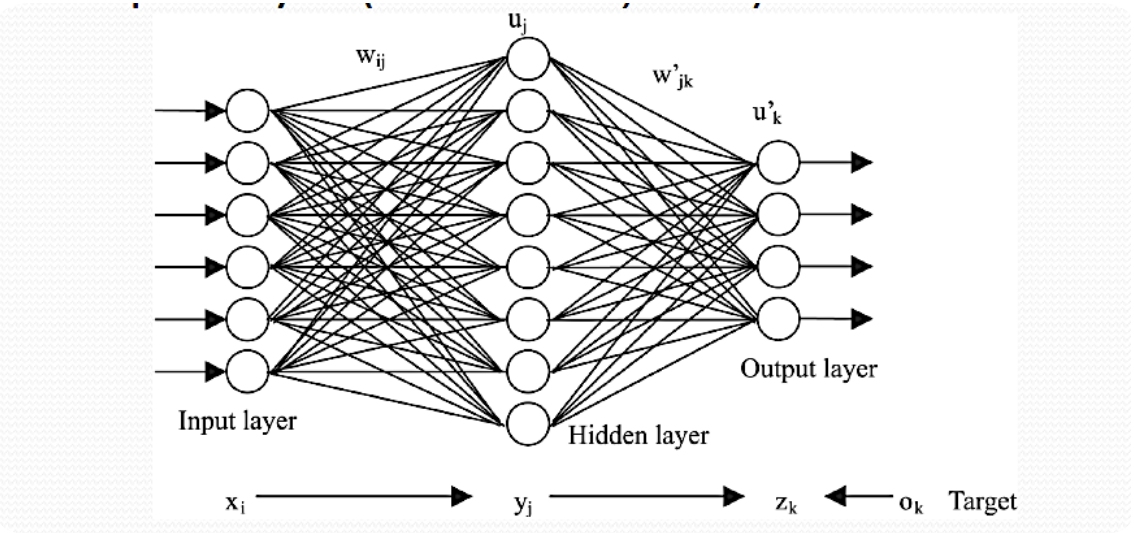
**Use them as:**

- Classification Systems
- Recognition and Identification Systems
- Forecasting Systems

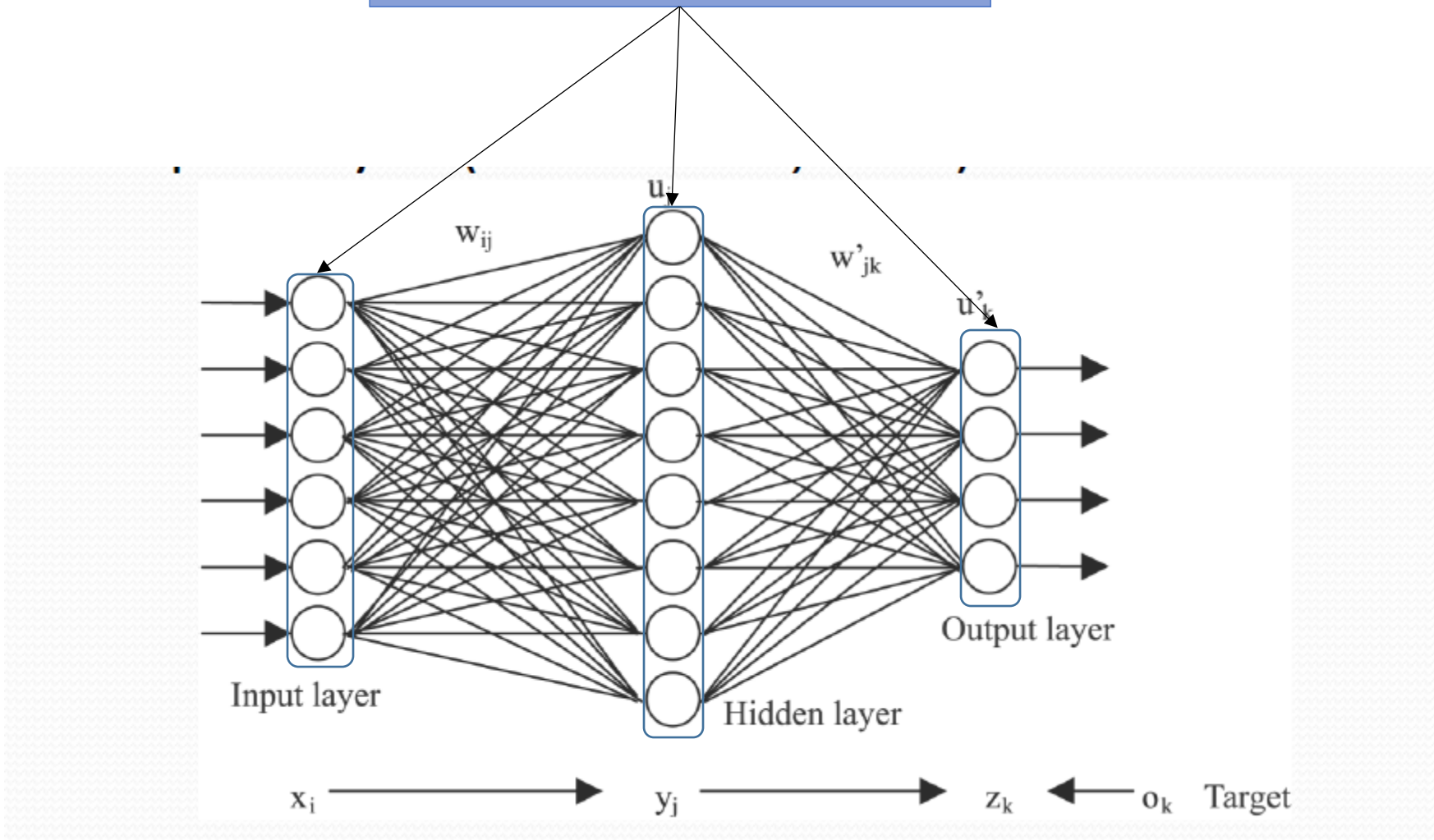
**Basic Architecture**

The basic architecture of an ANN consists of three types of layers:

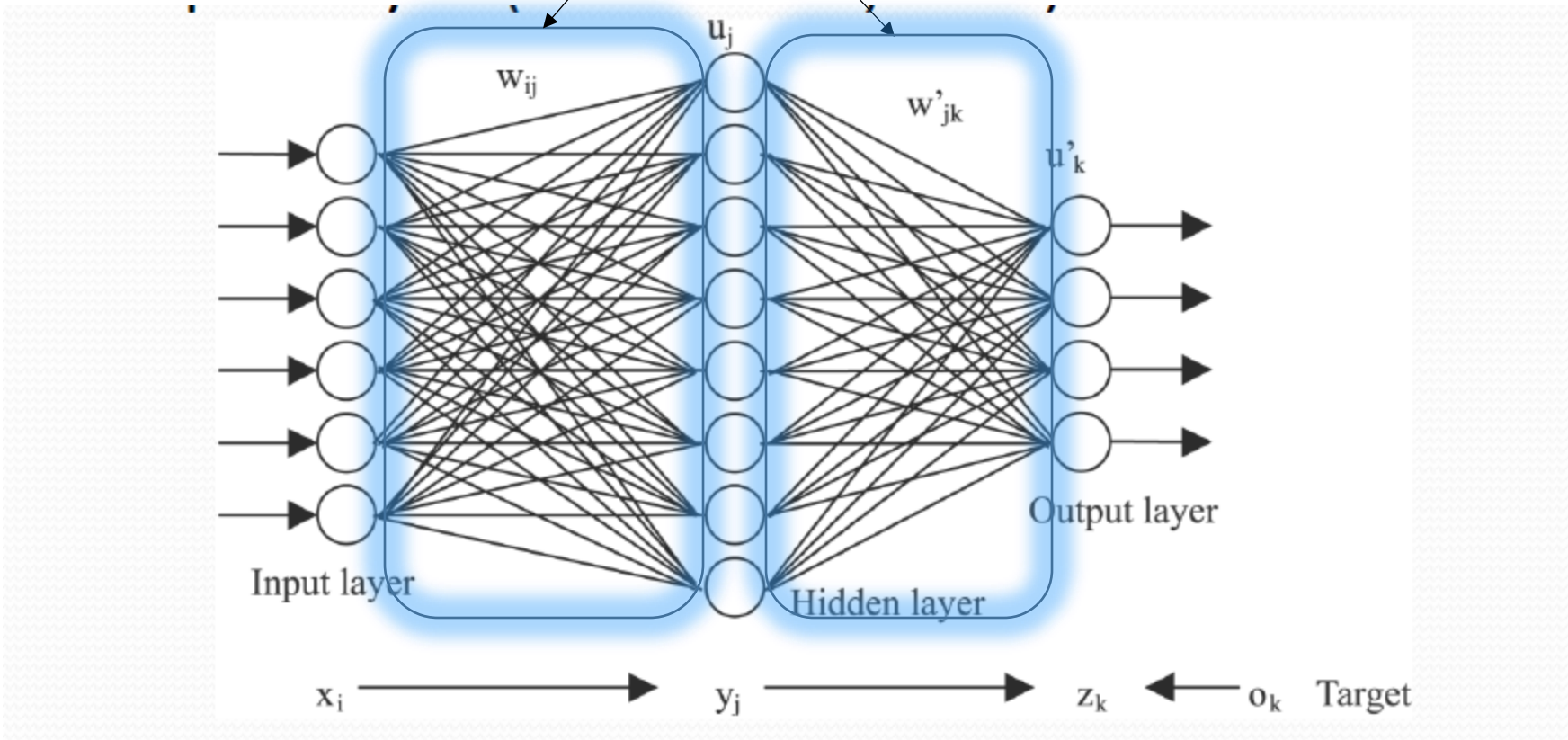
- Input layer
- Hidden layer/layers
- Output layer



Each layer consists of one or more neurons. Neurons are represented as circles in the figure.



Lines between neurons denote the flow of information from one neuron to the other.



**Input Examples**

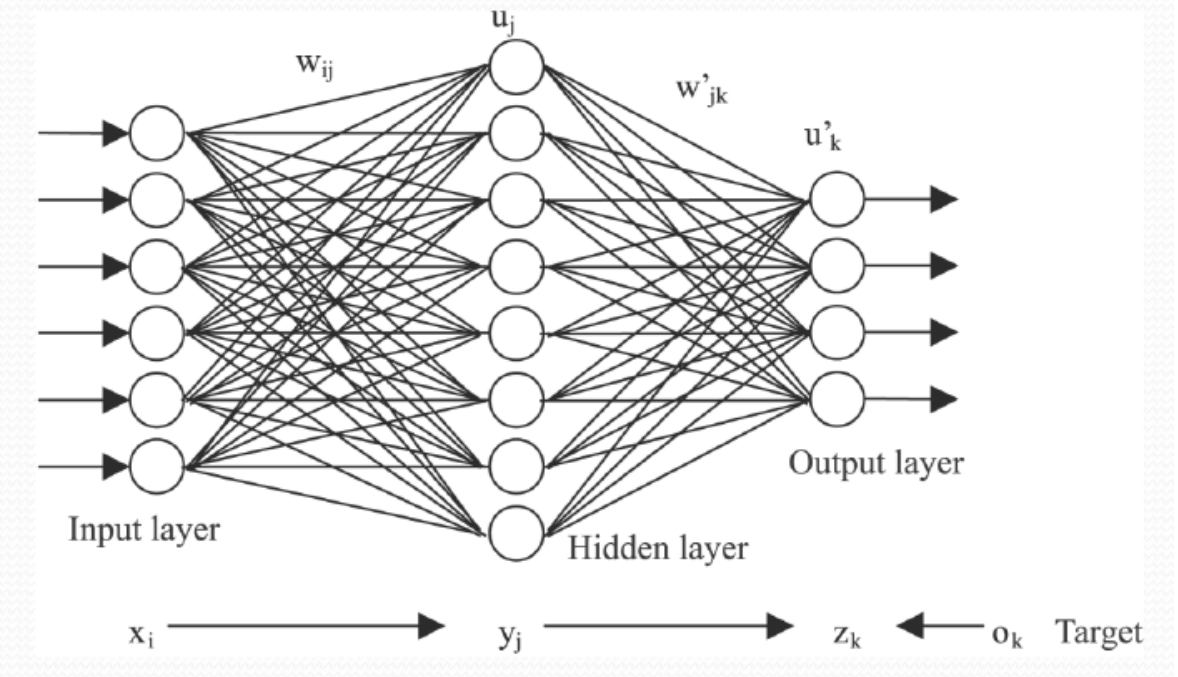
The input to a neural network depends on the goal we want to achieve

Therefore, we can have as input:

- Pixel (pixels) values of an image.
- Samples from an audio signal.
- Continuous stock prices.

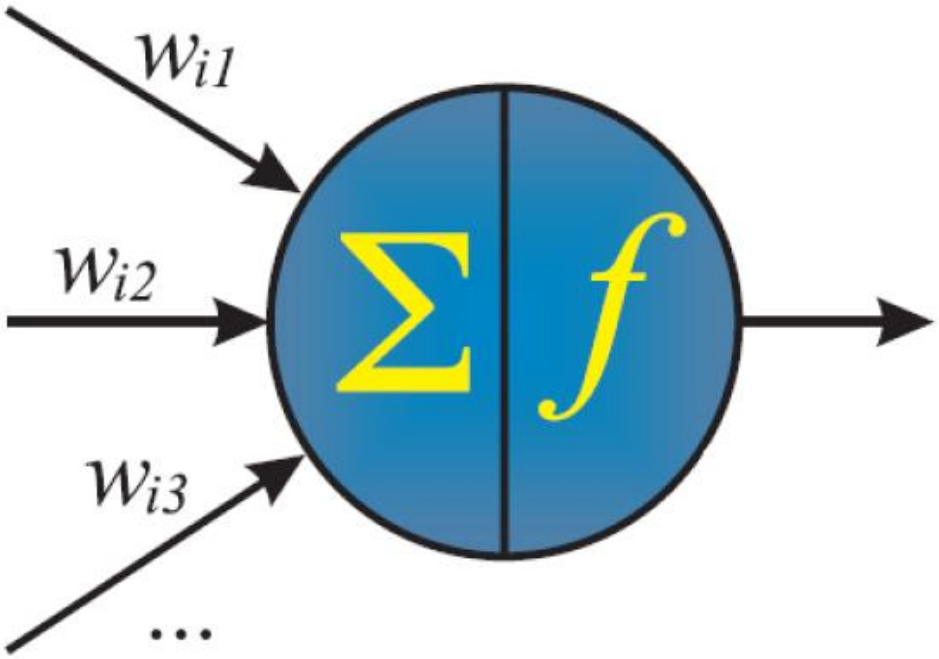
**What do they consist of?**

- The Artificial Neural Networks (ANN) consist of a set of neurons (elements) that are connected to each other.
- As with biological Neural Networks, the entire network function is determined by connections between neurons (elements).



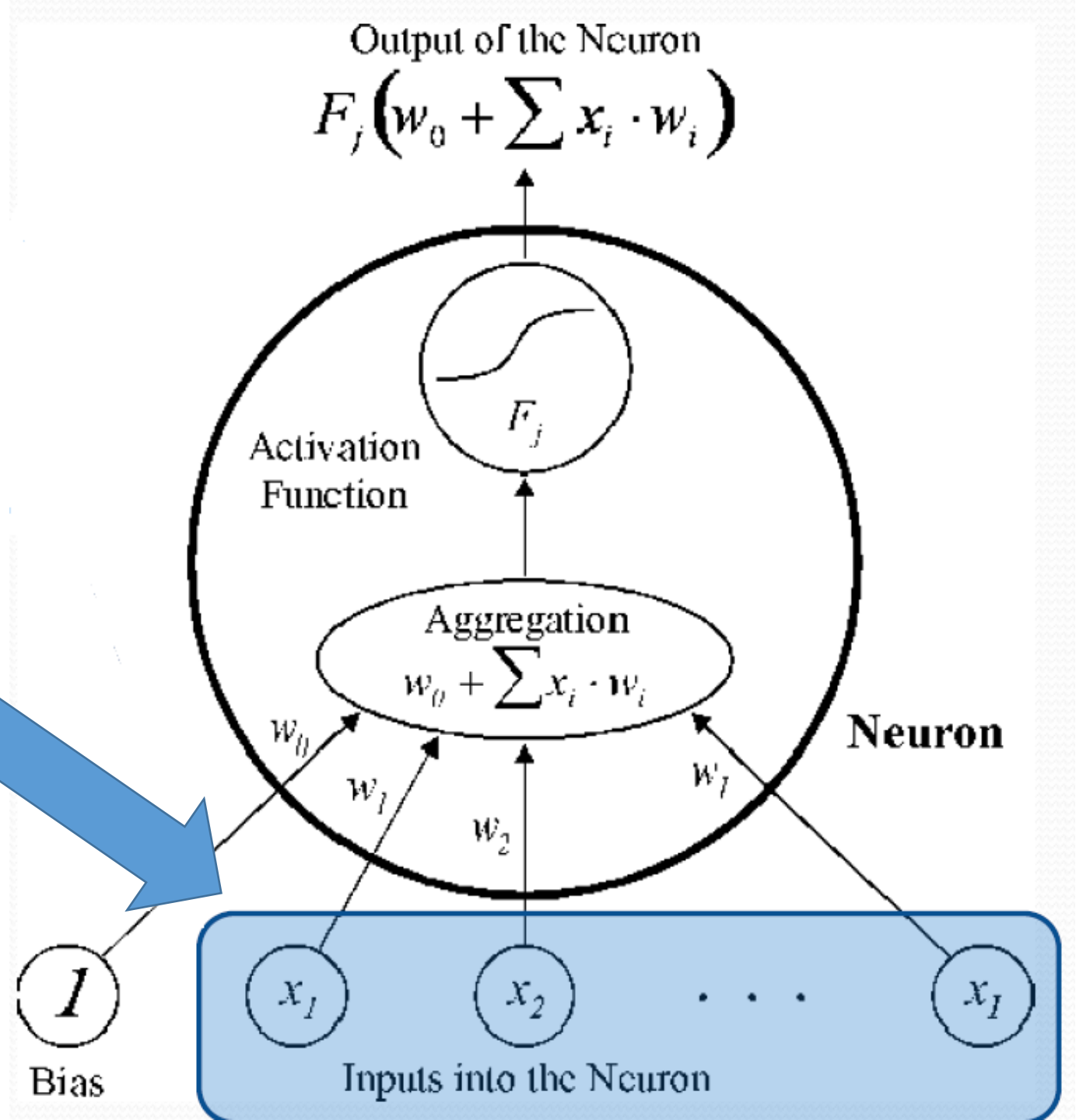
The Neuron

Each neuron implements an activation function **f** by taking an input and producing an output, which at the same time can be an input for one or more other neurons.



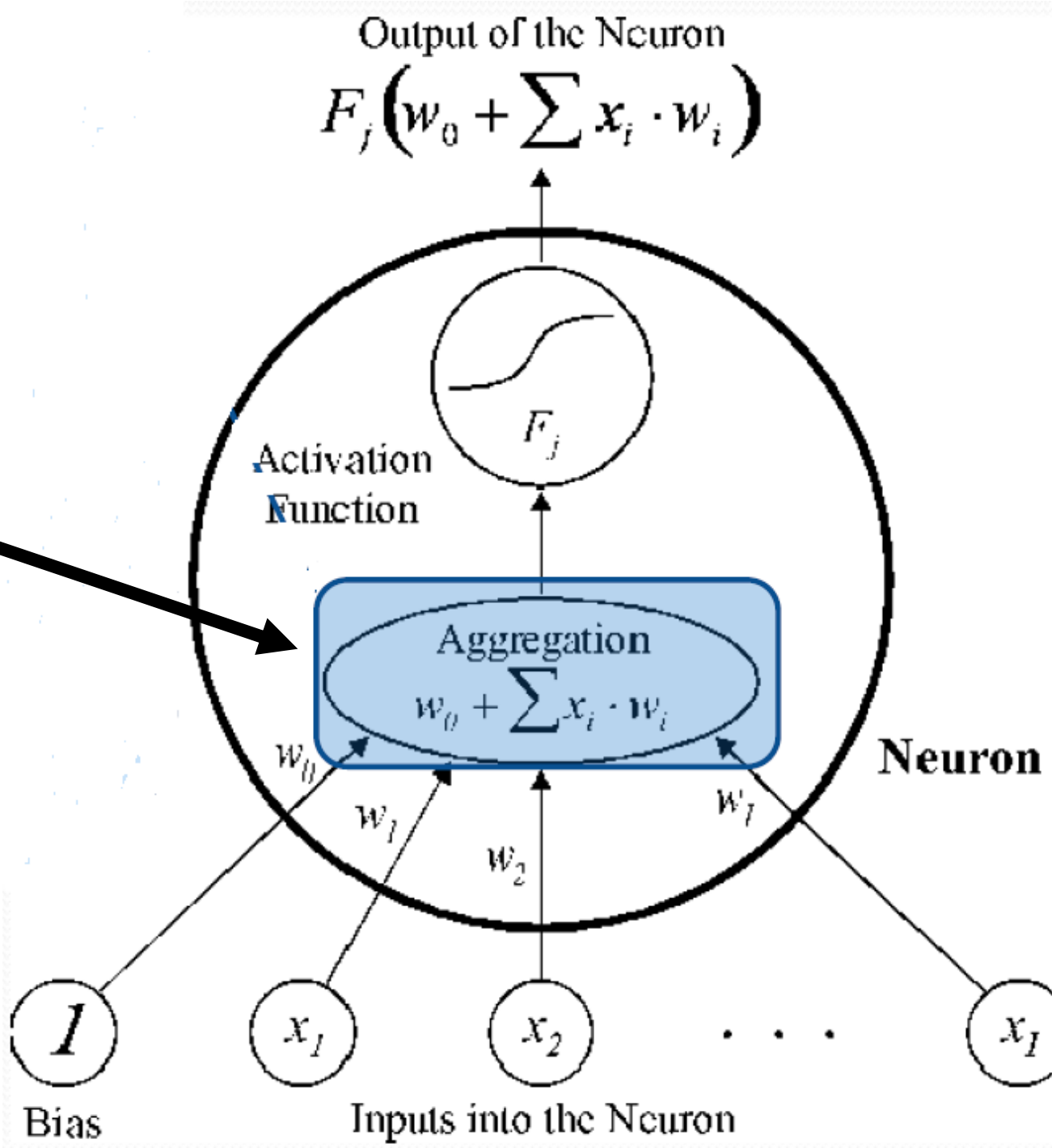
# The Neuron

A neuron receives as input a vector x.



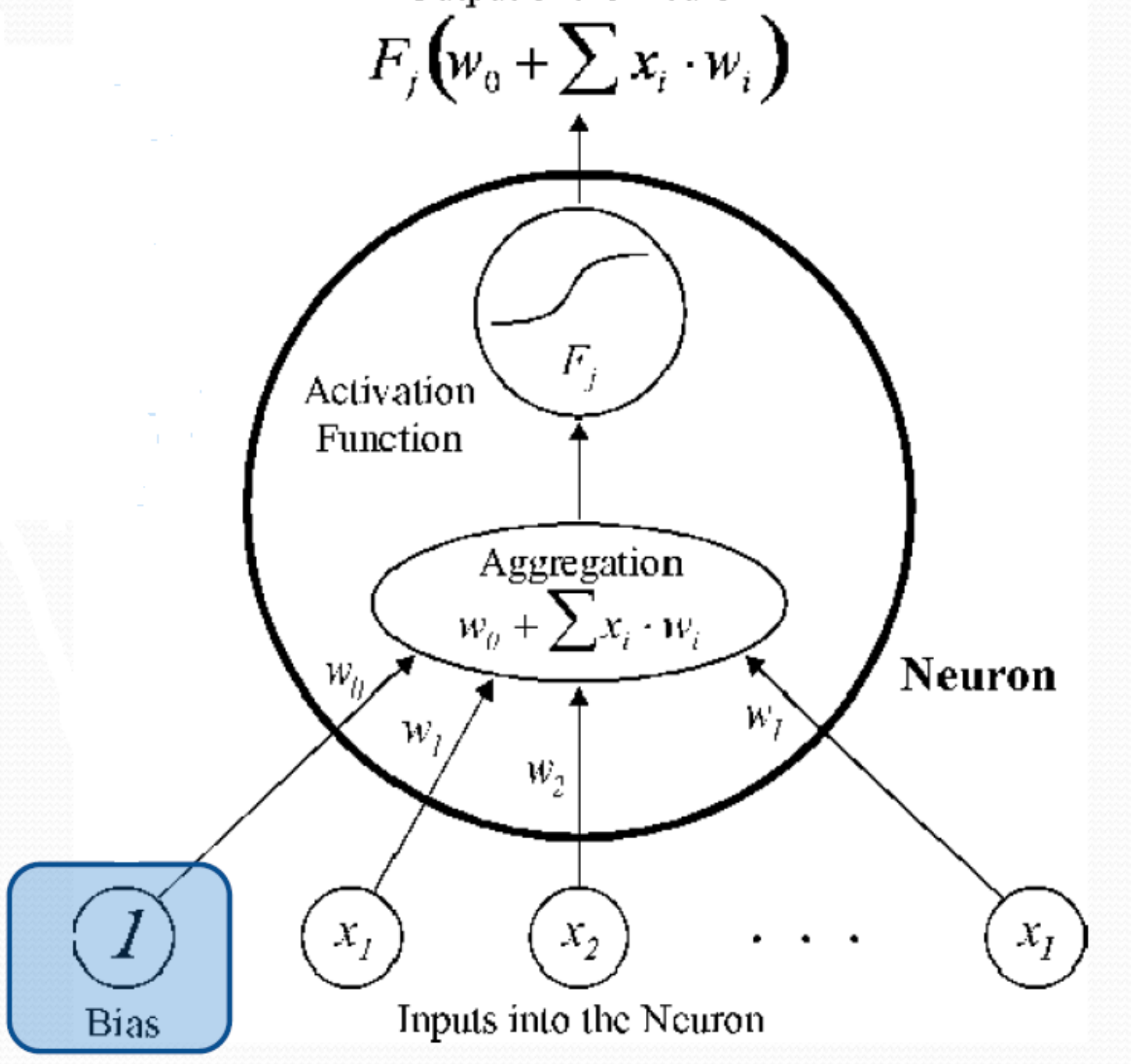
# The Neuron

- Each element of the vector enters weighted
- That is, it is multiplied by a coefficient w that we call weight
- It is then summed up with the other elements of the vector as well as a coefficient called the bias.



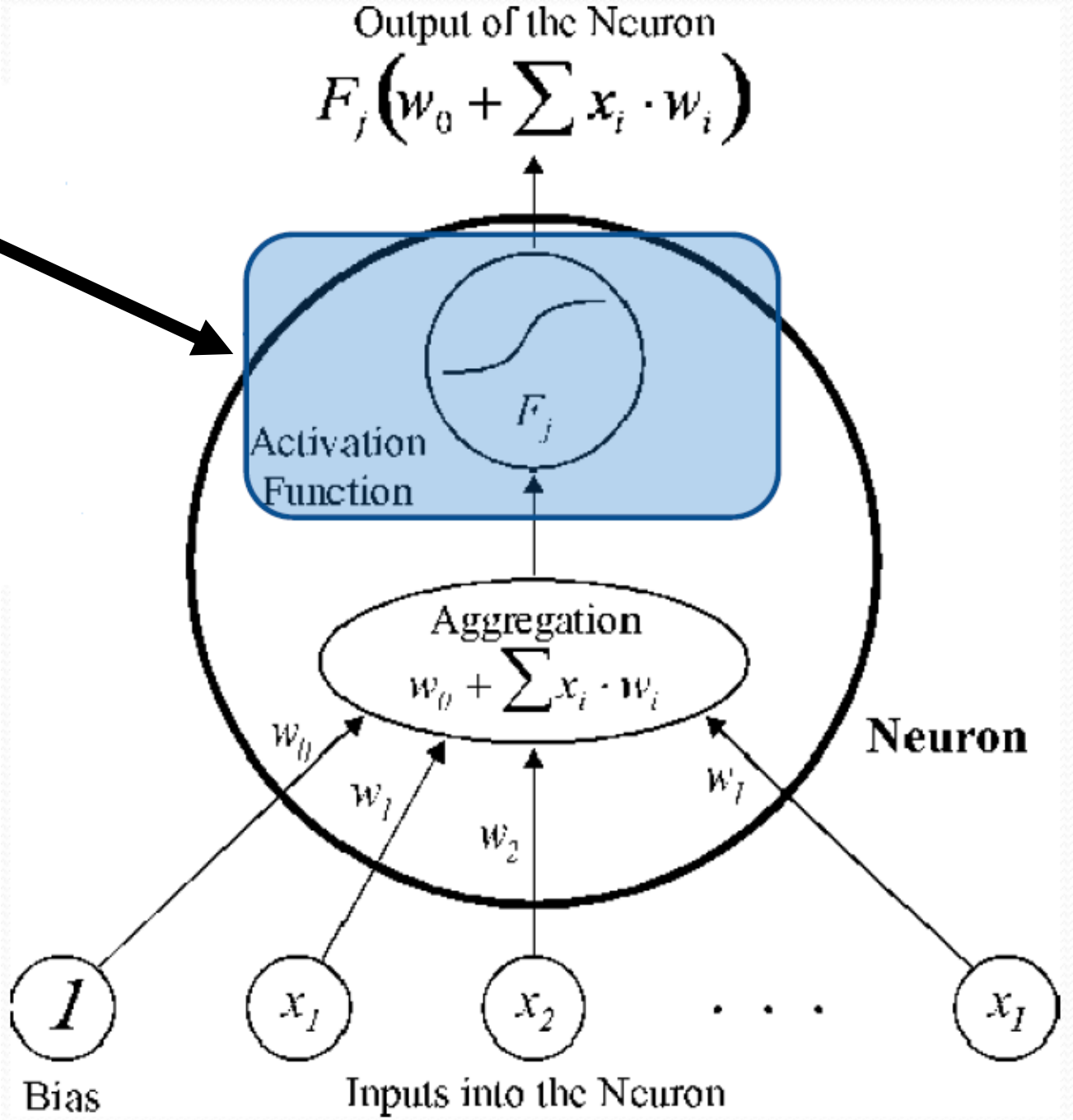
# The Neuron

The bias can be considered as a unit input multiplied by a coefficient $w_0$.



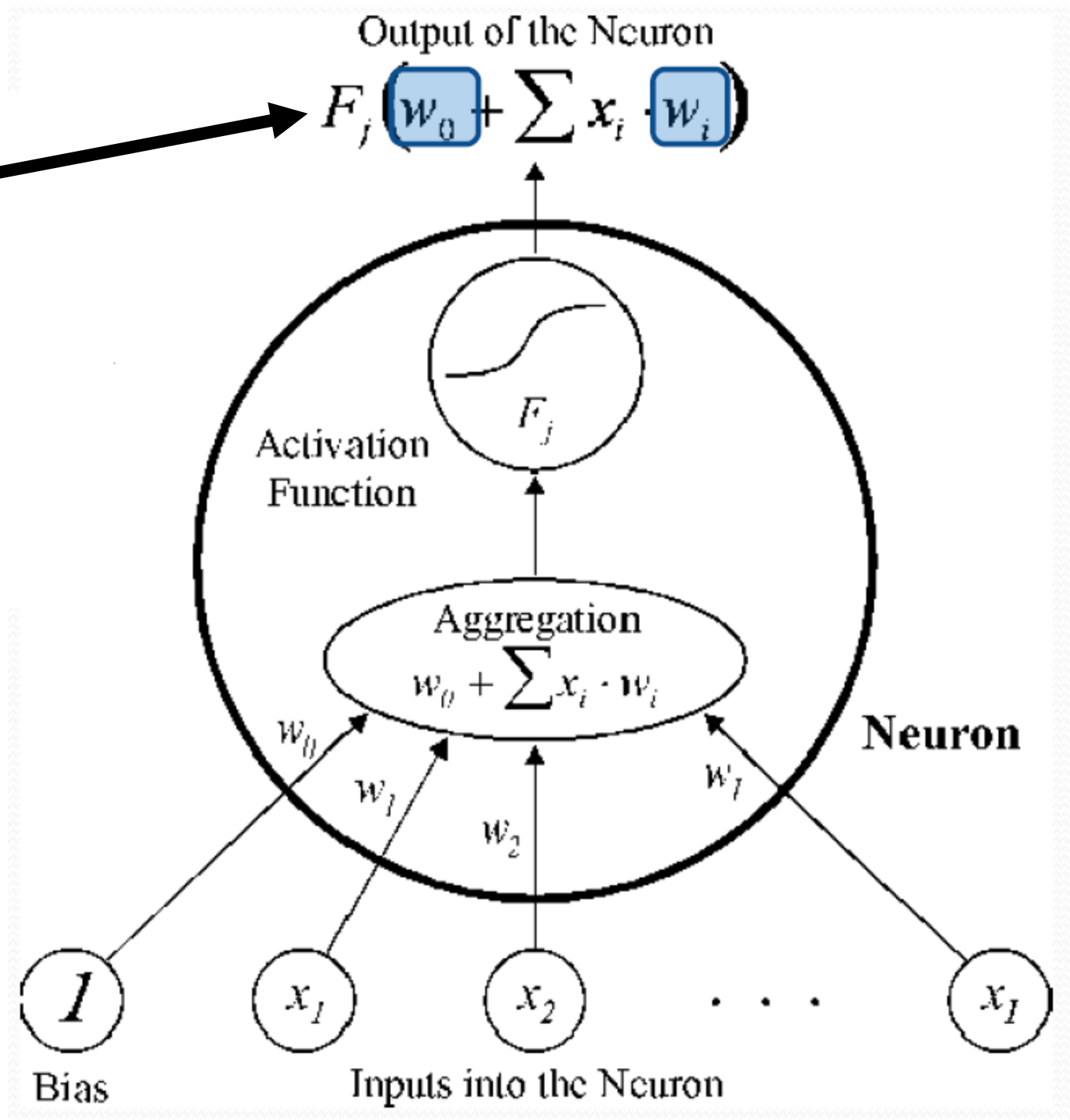
# The Neuron

The activation function f is implemented in the neuron, and we obtain its final output.



# The Neuron

We see that the output of each neuron depends on both the weights and the bias (with the same activation function).

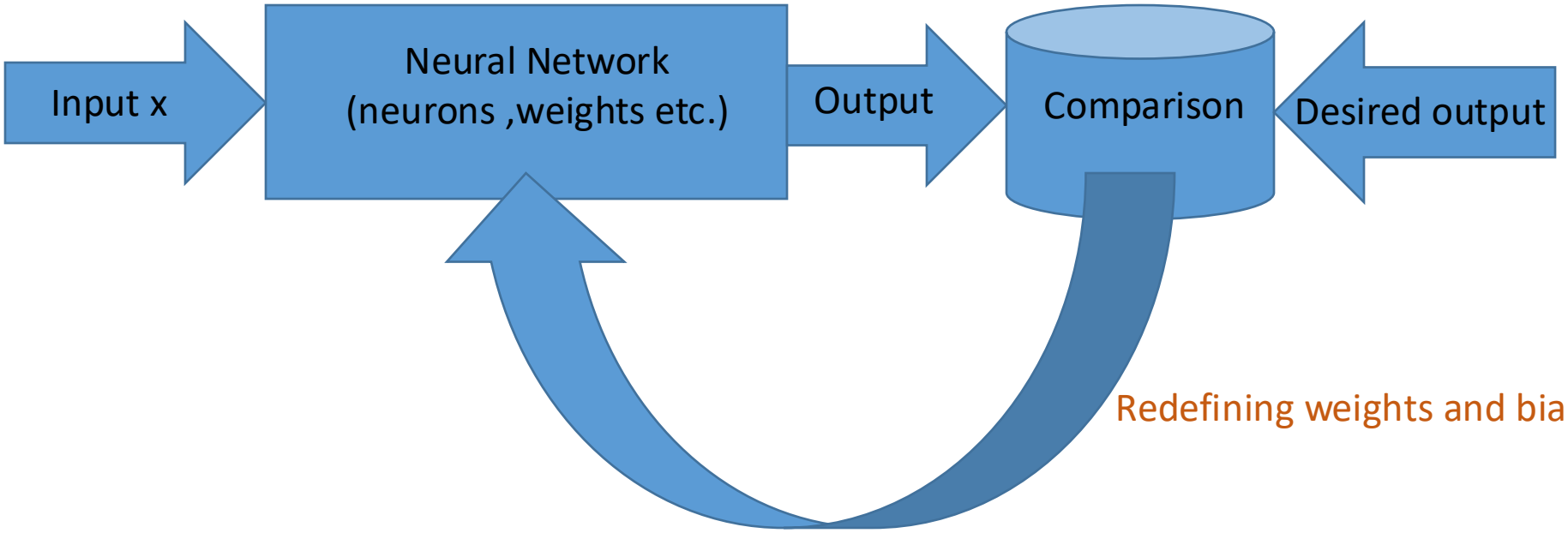


# Training an Artificial Neural Network

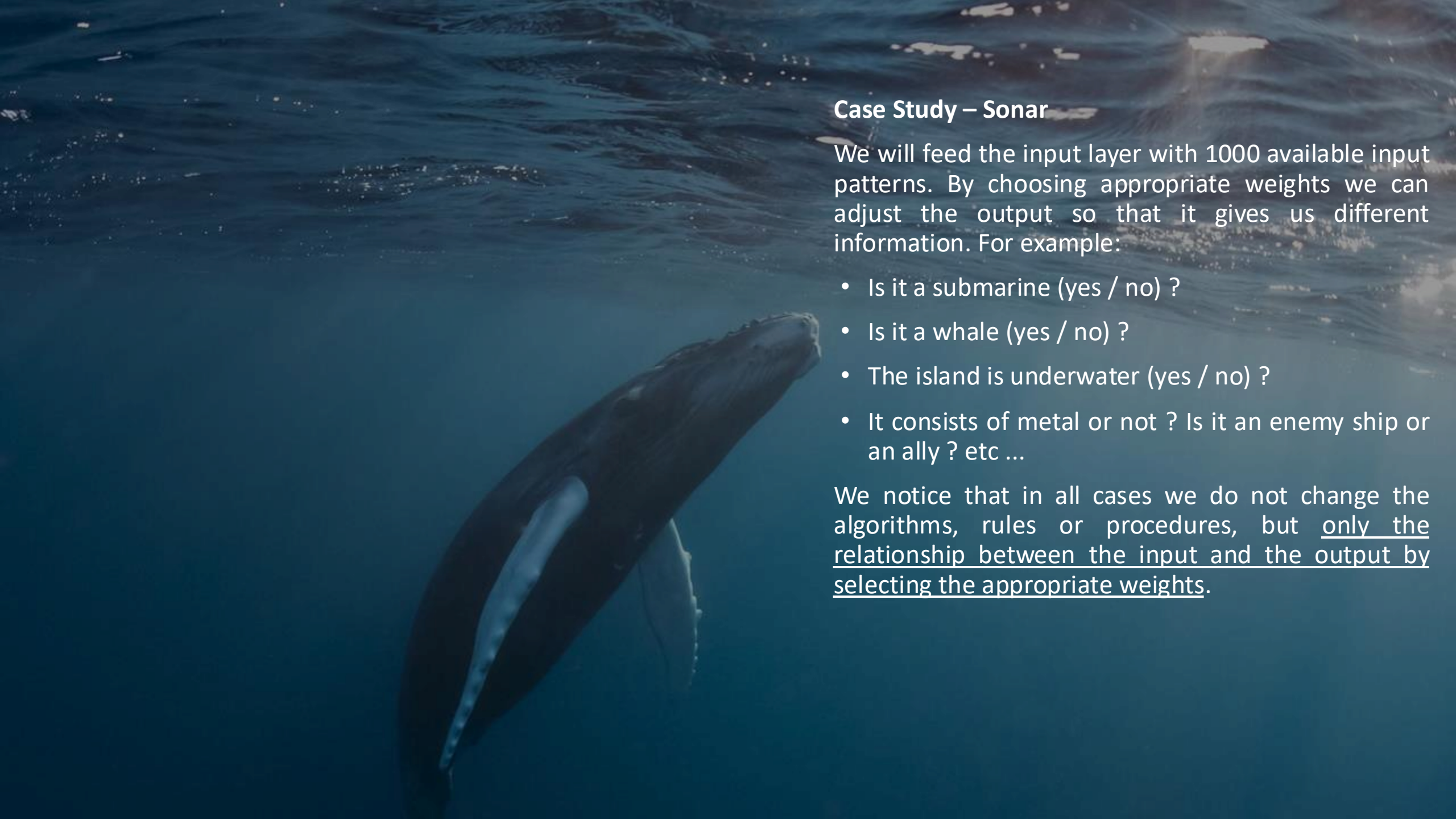
Training a neural network is to adjust properly:

- the network connections (weights)
- the bias

in order to implement a specific function so the network give us (or approach) a desired exit.



- The neurons in the input layer are passive, which means that they do not modify the data.
  - They take a value in the input and "broadcast" it to all the outputs.
  - If needed, we can modify our data before entering the neural network (Pre-Processing step)
- The neurons in the hidden layers and the output layer are *active*.
  - This means that they modify the input values

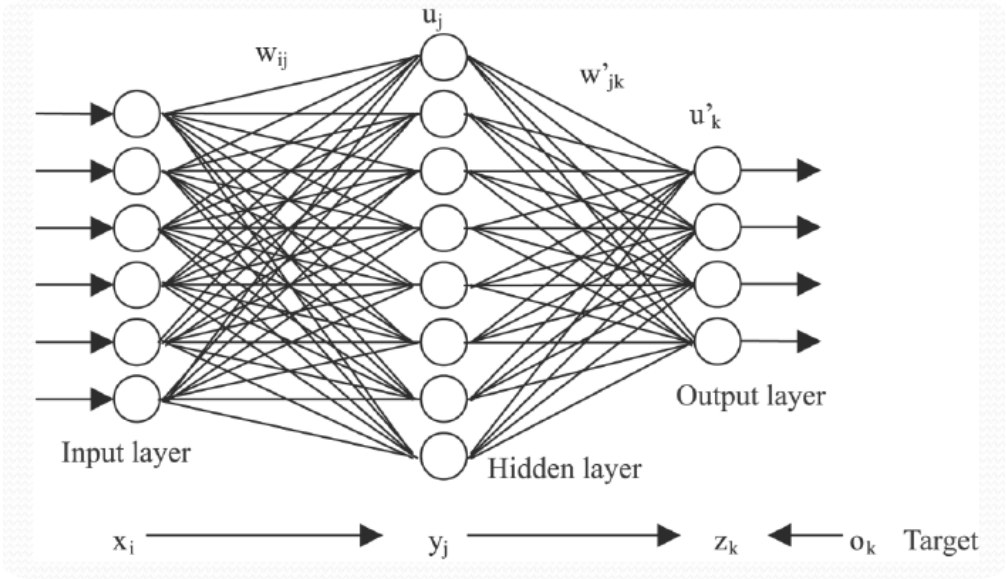


**Case Study – Sonar**

We will feed the input layer with 1000 available input patterns. By choosing appropriate weights we can adjust the output so that it gives us different information. For example:

- Is it a submarine (yes / no) ?
- Is it a whale (yes / no) ?
- The island is underwater (yes / no) ?
- It consists of metal or not ? Is it an enemy ship or an ally ? etc ...

We notice that in all cases we do not change the algorithms, rules or procedures, but only the relationship between the input and the output by selecting the appropriate weights.




**Attention…**

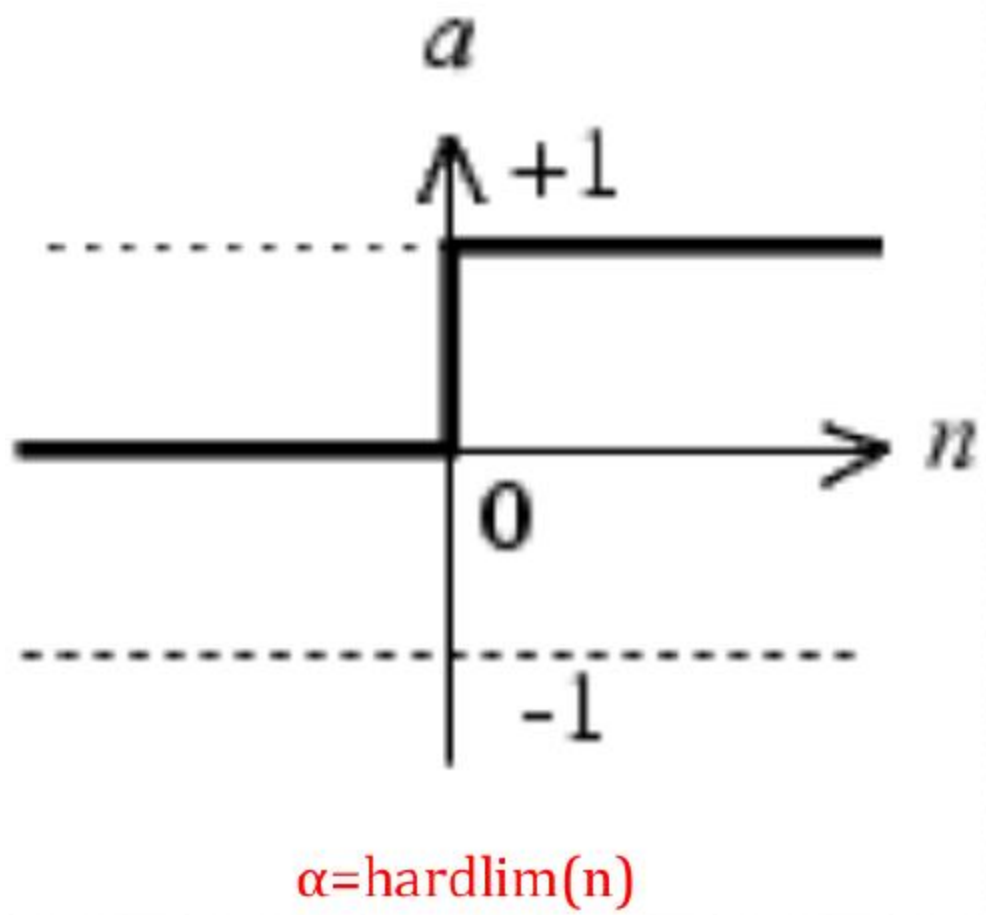
- We should mention that in a neural network it is not madatory that the number of neurons -in each layer- to be the same.
- Also, it is not necessary for all neurons to implement the same activation function.

## Activation Functions

- We can use a number of available activation functions.
- The one we choose depends on the application the neural network is used.
- Some basic activation functions and how they are implemented in Python (neurolab library), are presented in the next slides

# Hard Limit Activation Function

- The Hard - Limit activation function, limits the output of the neuron to 0 if the input to the neuron is negative, or to 1 if the input is greater than or equal to 0.
- This function is mainly used in Perceptrons networks for pattern recognition applications.
- The Python (neurolab) implementation for this function is HardLim () via Neurolab Library.



α=hardlim(n)

**Example**

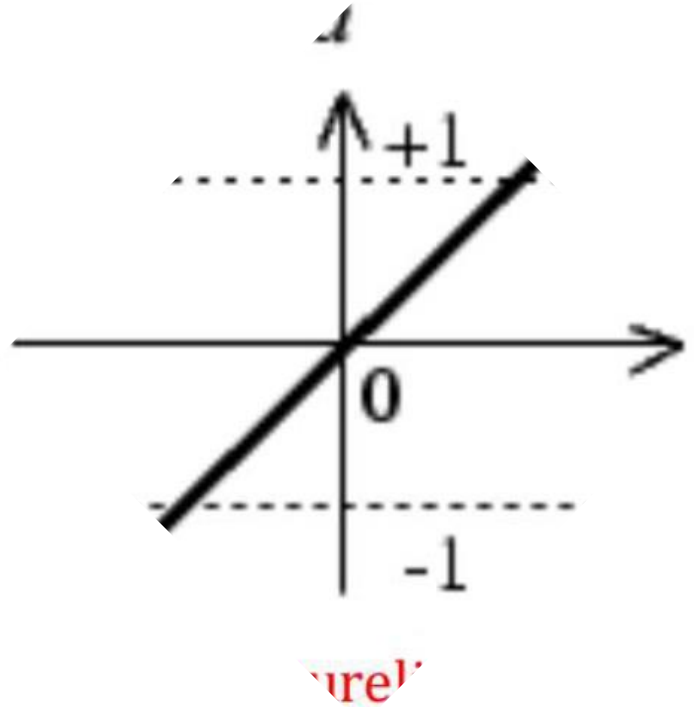
```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.HardLim()
x = np.arange(-5, 6, 0.1) #from -5 to 5 with step 0.1
plt.plot(x,f(x));
```

**Linear Activation Function**

The linear activation function is implemented in neurolab with the PureLin () function.

**Example**

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.PureLin()
x = np.arange(-5, 6, 0.1)
plt.plot(x,f(x));
```



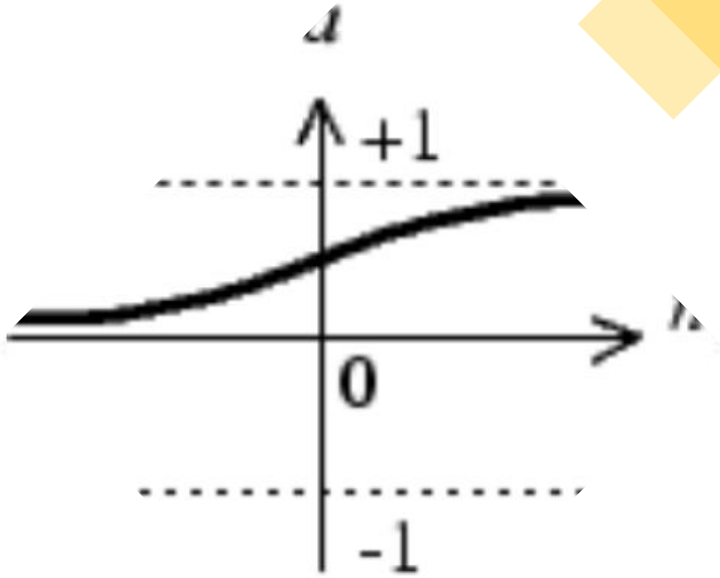
**Logarithmic Sigmoid Activation function**

The sigmoid activation function is implemented in neurolab with the LogSig () function and as a result it limits the output of the neuron into (0, +1).

a = 1 / (1 + exp(-n))

**Example**

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.LogSig()
x = np.arange(-5, 6, 0.1)
plt.plot(x,f(x));
```



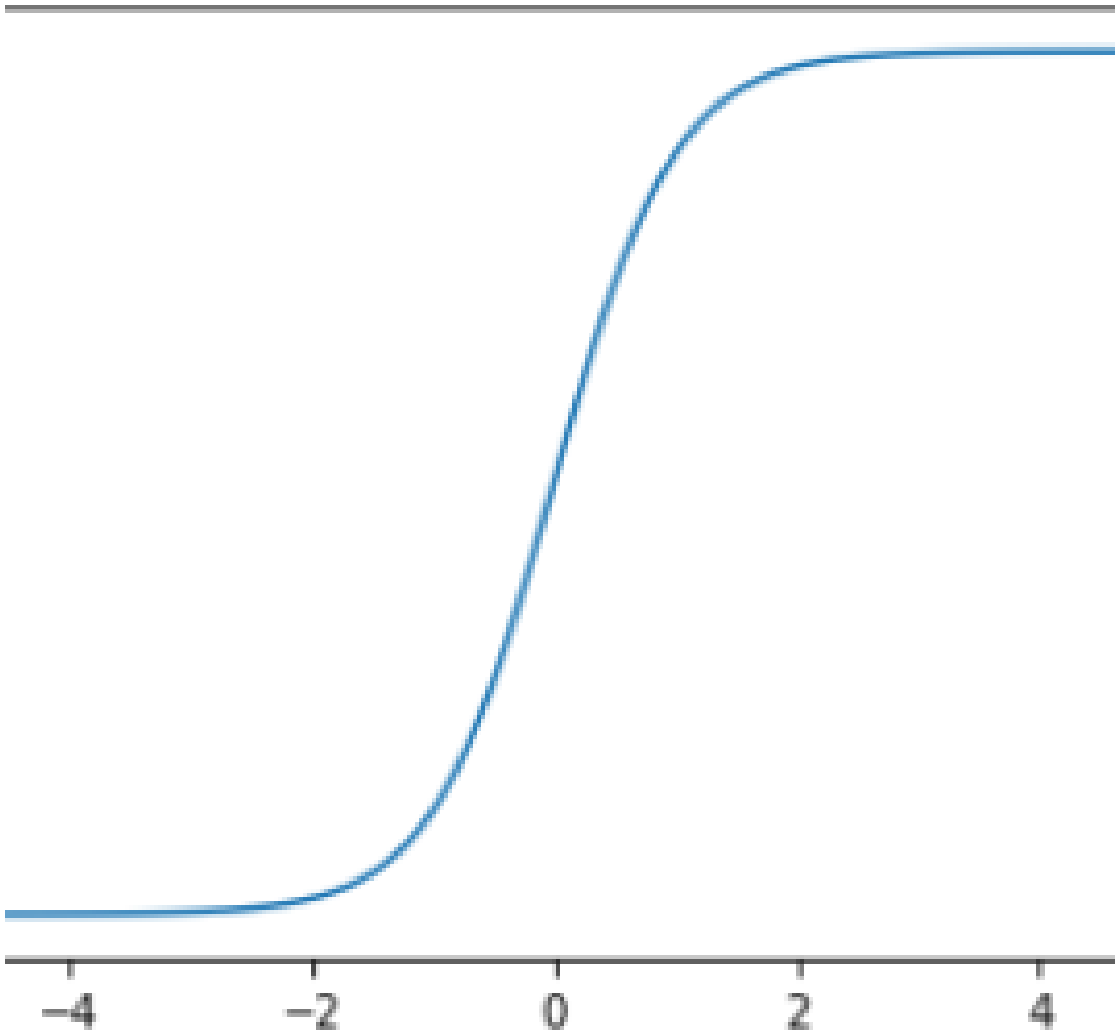
**Hyperbolic tangent sigmoid activation function**

The hyperbolic tangent sigmoid activation function is implemented in neurolab with the TanSig() function and as a result it limits the output of the neuron into (-1, +1).

a=2/(1+exp(-2*n))-1

**Example**

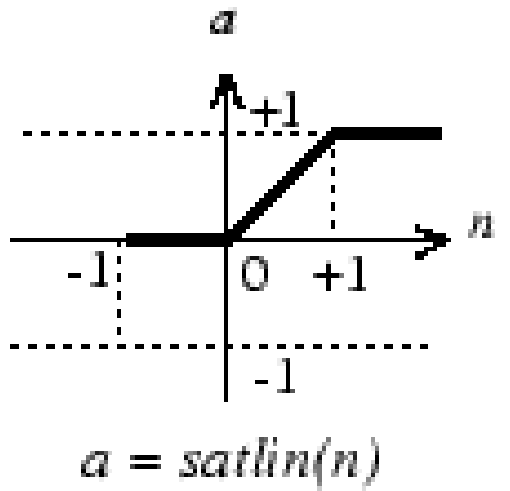
```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.TanSig()
x = np.arange(-5, 6, 0.1)
plt.plot(x,f(x));
```



## Saturating linear activation function

The saturating linear activation function is implemented in neurolab with the SatLin() function and as a result it limits the output of the neuron into (0, +1).

a = satlin(n) = 0, if n <= 0
n, if 0 <= n <= 1
1, if n>=1




**Example**

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.SatLin()
x = np.arange(-5, 6, 0.1)
plt.plot(x,f(x));
```

# Soft max activation function

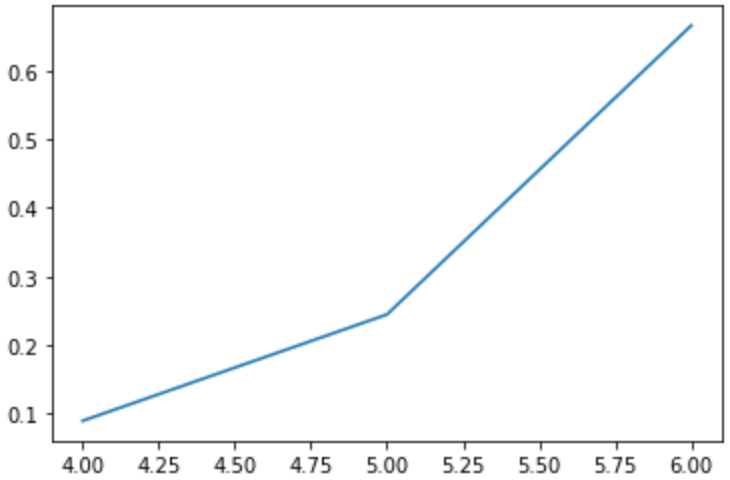
The soft max activation function is implemented in neurolab with the SoftMax() function.

$$softmax(n) = exp(n)/sum(exp(n))$$

The softmax function simply divides the exponent of each input element by the sum of exponents of all the input elements. The sum of the output will be equal to 1.

**Example**

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.SoftMax()
x = np.array([4, 5, 6])
plt.plot(x,f(x));
print(f(x))
```




```
[0.09003057 0.24472847 0.66524096]
```

**Competitive activation function**

The competitive activation function is implemented in neurolab with the Competitive() function.

The competitive activation function simply takes the values: 0 or 1. 1 if it is a minimum element of x, else 0.

**Example**

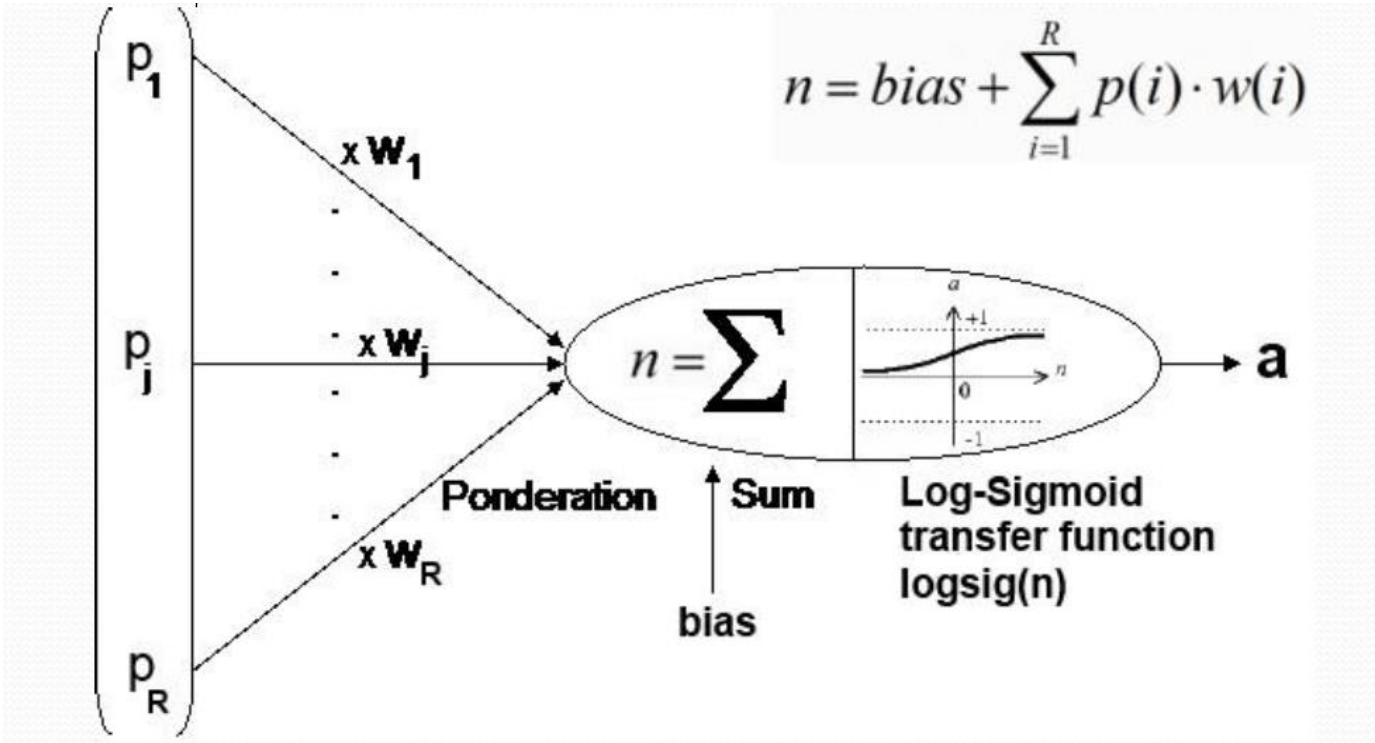
```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

f = nl.trans.Competitive()
f([-5, -0.1, 0, 0.1, 100])
array([ 1.,  0.,  0.,  0.,  0.])
```

## A vector as an input to the neuron

- Now let's present an input vector to a neuron that implements the sigmoid activation function.
- As we know, a vector is a set of numbers that are represented in the form of a one-dimensional array.
- Each one of these numbers is called a characteristic value (feature) and each input vector is called a training pattern.

**A vector as an input to the neuron**

Let us assume that we have a vector p (R dimensional) and which is given to the neuron as an input.

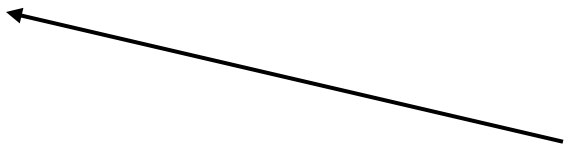


# Python implementation (1st approach)

```
import numpy as np
import neurolab as nl

p = np.array([2,4,6,8]) # a random input vector
w = np.array([0.1,0.2,0.3,0.4]) # a random weight vector
b = 0.5 # the bias
s = 0

f = np.empty([4])

for i in range(4):
    f[i] = p[i]*w[i]
    s = s+f[i]

n = s+b # the output after summation
f = nl.trans.LogSig()
a = f(n) # the final output
```

# Python implementation (2nd approach)

```
import numpy as np
import neurolab as nl

p = np.array([2,4,6,8]) # a random input vector
w = np.array([0.1,0.2,0.3,0.4]) # a random weight vector
b = 0.5 # the bias

n = np.sum(np.multiply(p,w))+b
f = nl.trans.LogSig()
a = f(n)
```

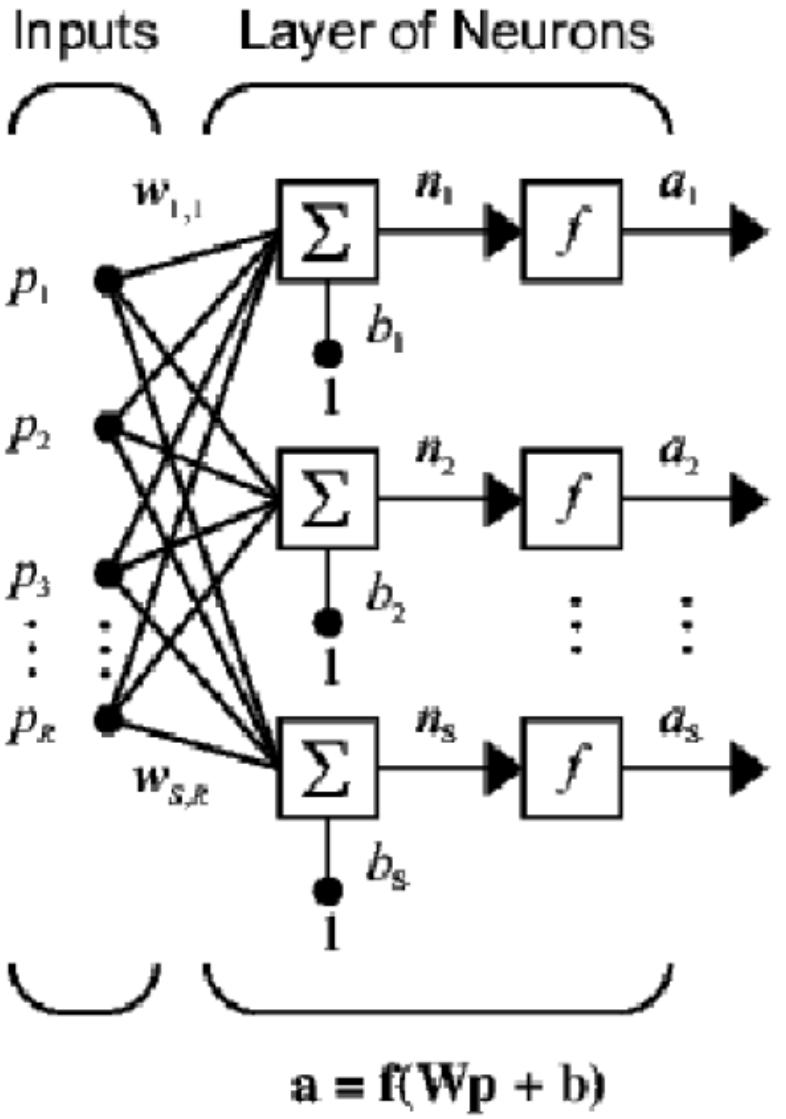


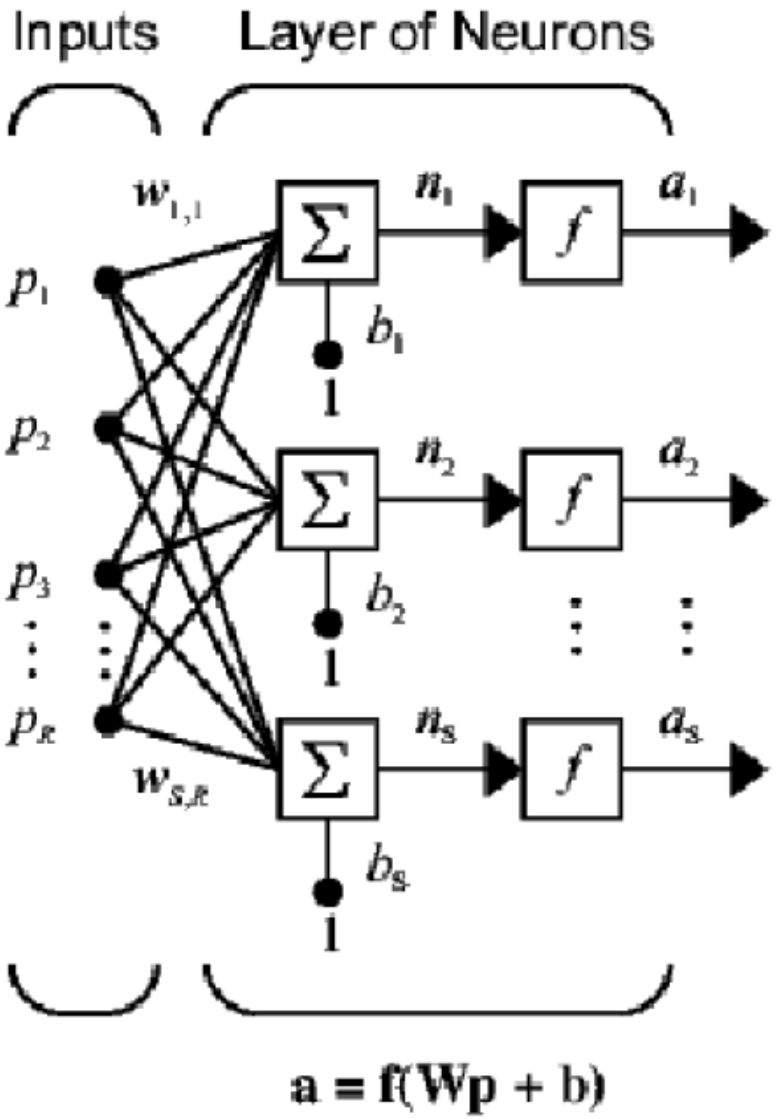
*The output of the neuron is a number*

If we have R neurons, then the output will be R-dimensional.

## Layers of active neurons

- Typically, all characteristic values (features) of the input are fully associated with the layer of neurons (fully interconnected).
- For each neuron in the layer, the characteristic values are pondered by the weight coefficient (w).
- As an overall output from the neuron level, we have a vector that has S values, as much as the layer neurons.





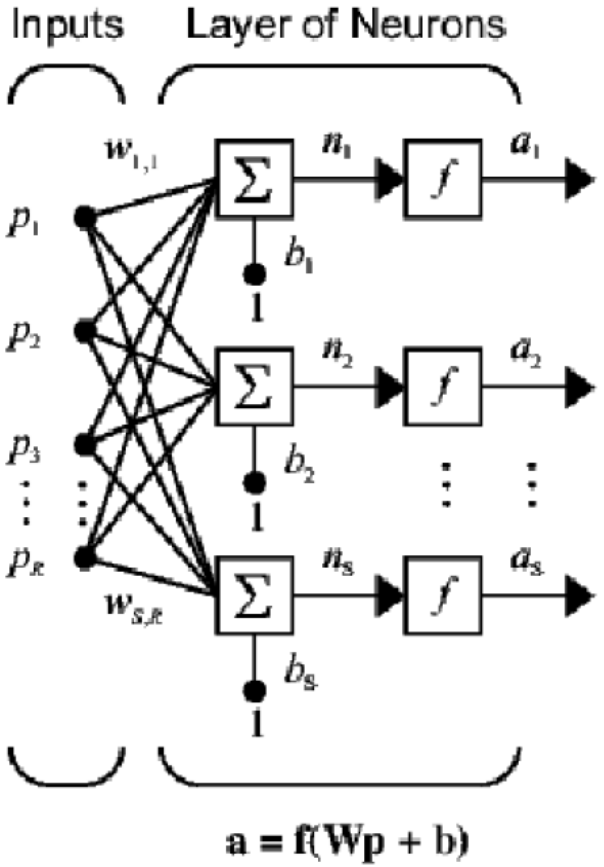

**We notice that…**

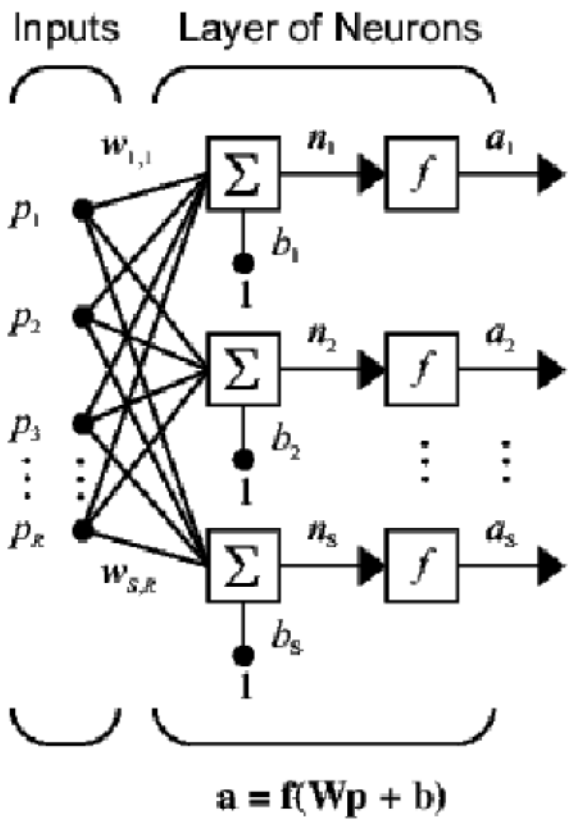
- It is not necessary that the number of neurons coincides with the number of features of the input vector.
- It is not necessary that all neurons implement the same transfer function.
- If for example, the neurons of a layer implements two activation functions (e.g., f1 and f2) we create parallel networks.

**We notice that…**

- The weights will be denoted by the term $\mathbf{w_{ij}}$
- All connections between the input vector and the layers' neurons are indicated by the matrix of weights which is:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$






$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

**We notice that…**

- Each line of this matrix contains the weights of the connections leading to a particular neuron
- Each column contains the weights of the connections starting from a particular characteristic input value.

# Implementation

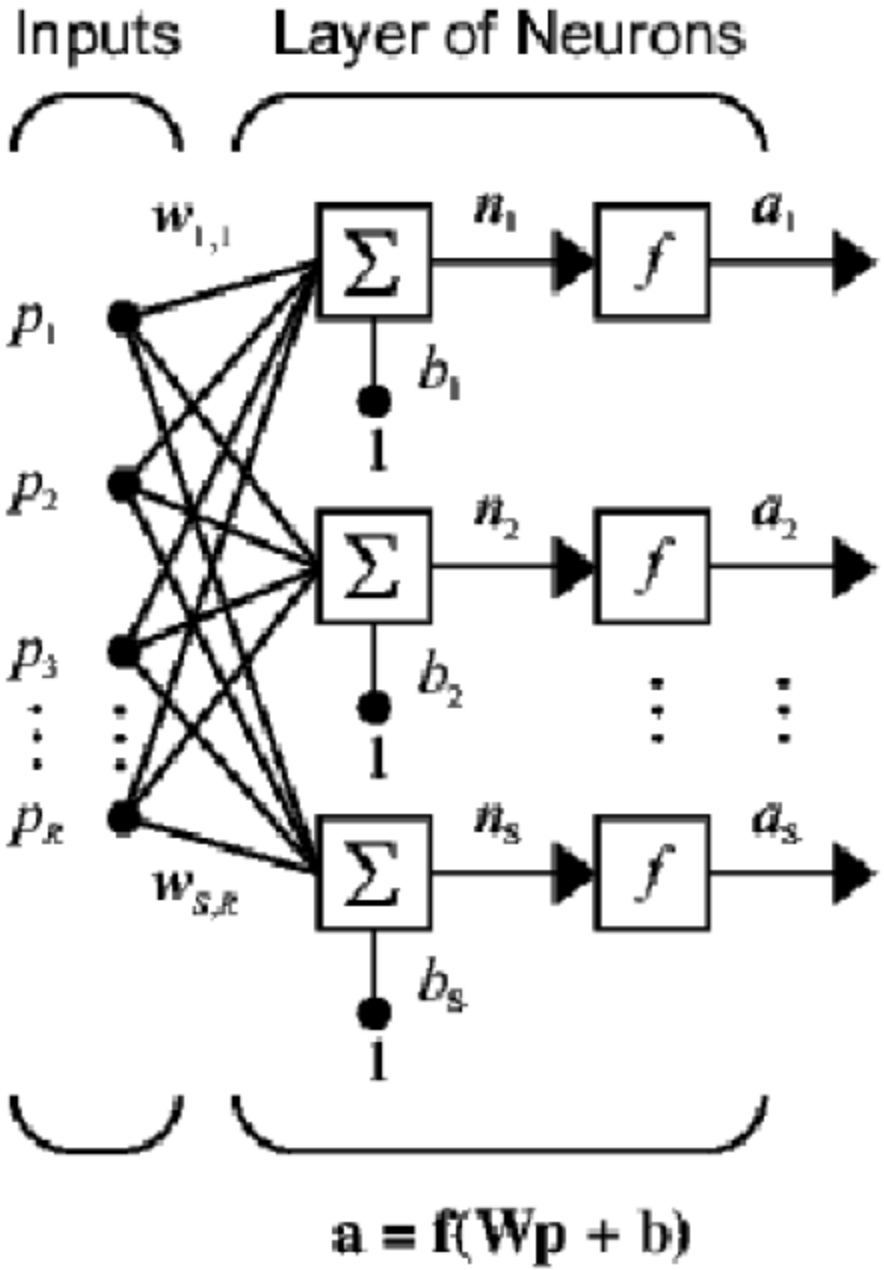
- The sum of each neuron $n_i$ is calculated by the formula:

$$n_i = p_1 W_{i1} + p_2 W_{i2} + \cdots + p_R W_{iR} + b_i$$

- $n_i$ value enters the transfer function f and the final output of the neuron is:

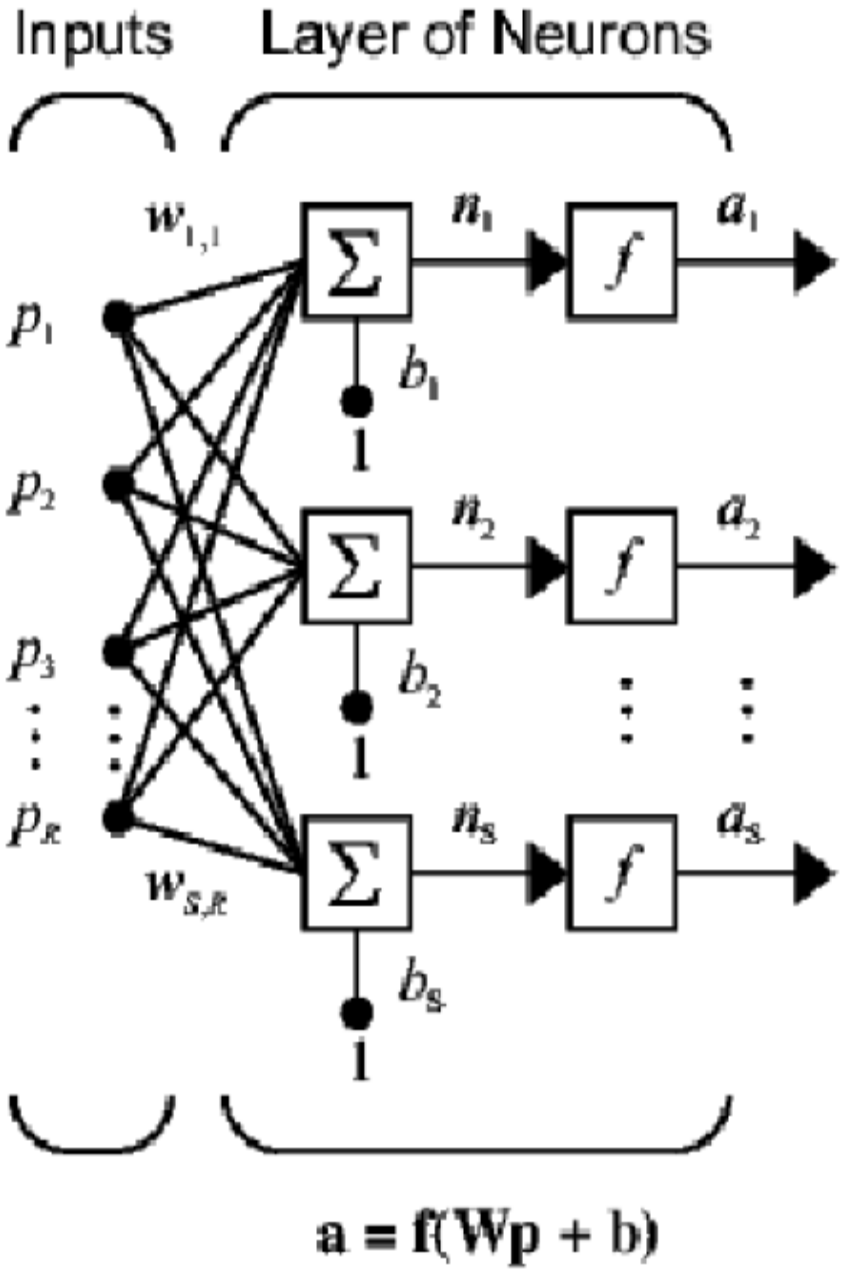
$$a_i = f(n_i)$$

- The total output of the layer is the a vector.



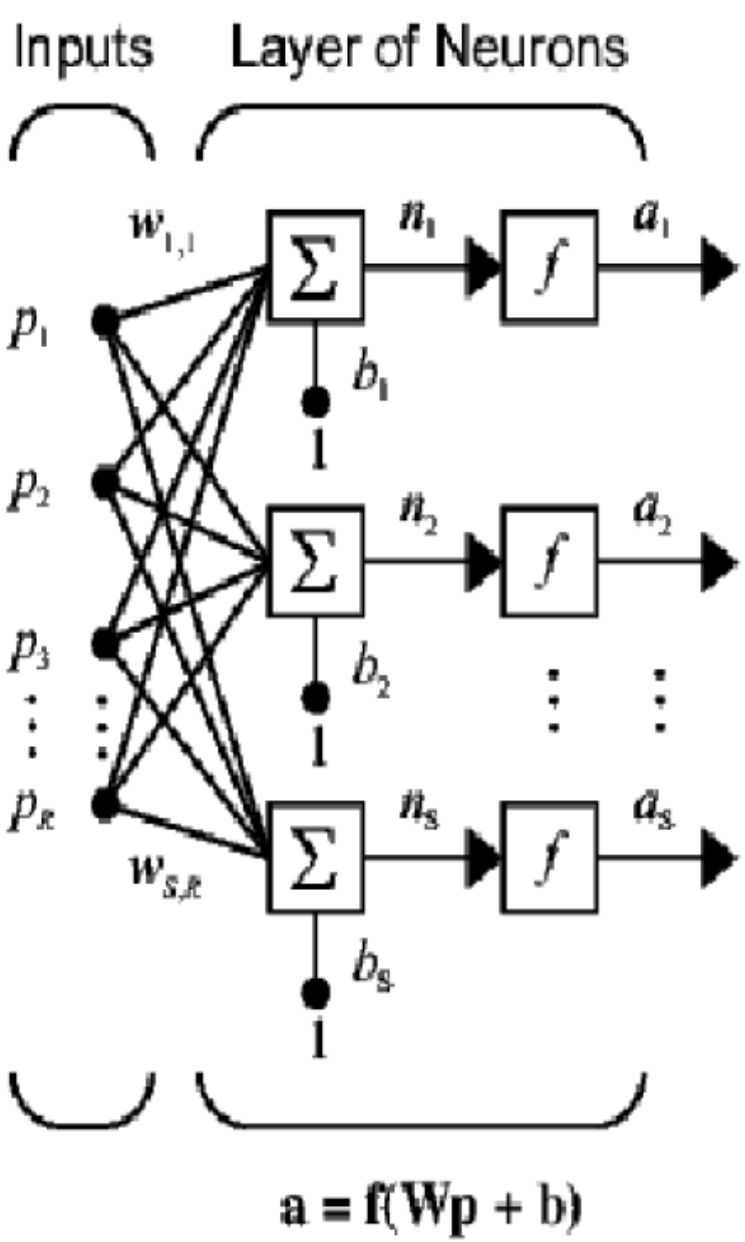
# Python Implementation

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

p = np.array([2,4,6,8]) # the input vector

w = np.array([[0.1, 0.2, 0.3, 0.4],
              [0.5, 0.6, 0.7, 0.8],
              [0.9, 1.0, 1.1, 1.2],
              [1.3, 1.4, 1.5, 1.6]]) # the weight vector

bias = 0.5

a = np.empty([4])
n = np.empty([4,4])

for i in range(4): #from neuron 1 to neuron 4

   a[i] = 0

   for j in range(4): #for each value of weight and input vector

      n[i,j] = w[i,j]*p[j]
      a[i]   = a[i]+n[i,j]

   a[i] = a[i]+bias;
   f    = nl.trans.LogSig()
   a[i] = f(a[i])
```



# Python Implementation -2nd approach

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

p = np.array([2,4,6,8]) # the input vector
w = np.array([[0.1, 0.2, 0.3, 0.4],
              [0.5, 0.6, 0.7, 0.8],
              [0.9, 1.0, 1.1, 1.2],
              [1.3, 1.4, 1.5, 1.6]]) # the weight vector
bias = 0.5
R = 4 # number of characteristic values
S = 4 # number of neurons

a = np.empty([1,S])
n = np.empty([R,S])

for i in range(R):

    for j in range(S):
        n[i,j] = w[i,j]*p[j]
        a[0,i] = a[0,i]+n[i,j]

    a[0,i] = a[0,i]+bias
    f      = nl.trans.LogSig()
    a[0,i] = f(a[0,i])
```



# Exercise 1

- Create a random* input vector. It will contain three (3) patterns, with four (4) features for each pattern.
- Create a random* matrix of weights and random* bias as well.
- Calculate the output values of a neuron in which the activation function is sigmoid.

*Tip: for every pattern you have one output.*

* use rand function from numpy

# Exercise 2

- Modify the code of the the previous exercise in order to give us results when we have the ‘data’ from Iris Dataset as an input.
- Install scikit-learn via **Anaconda-Navigator**
- Load Iris with this code:

```
from sklearn.datasets import load_iris
iris = load_iris()
input_vector = iris.data
```

- The result will be a matrix , in which there will be the outputs for each training pattern (150), for each neuron (4).
- Information for this data set can be found in here