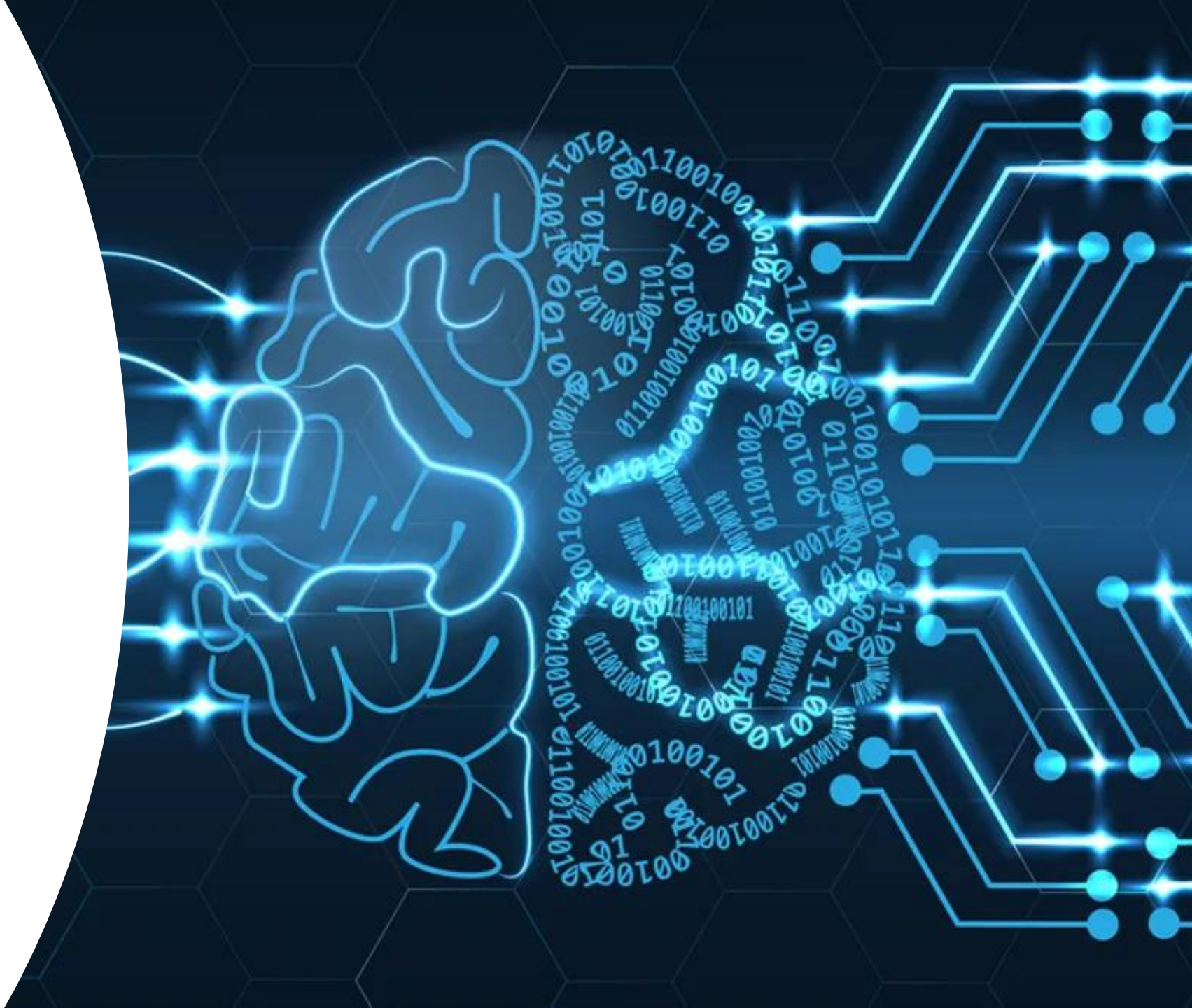
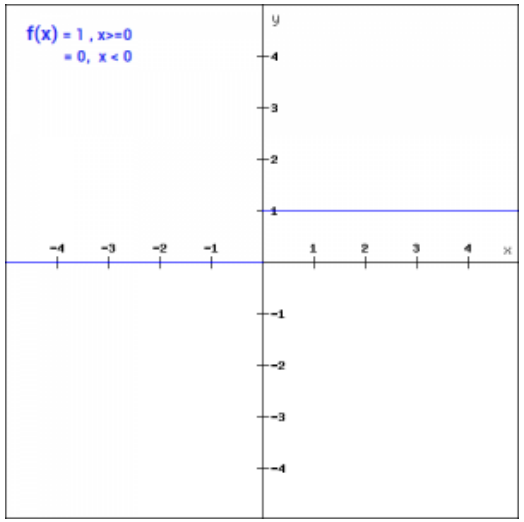
AMITY UNIVERSITY

Deep Neural Networks



Binary Step

f(x) = 1, x>=0
= 0, x<0



Python Code:

```
def
binary_step(x):
    if x<0:
        return 0
    else:
        return 1
binary_step(5),
binary_step(-1)
```

Output:

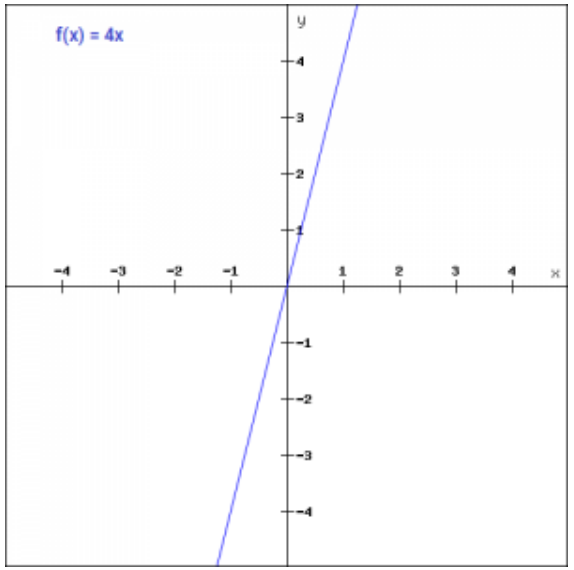
```
(5,0)
```

- Threshold based classifier
- If the input to the activation function is > threshold, then neuron is activated, else deactivated, i.e. its output is not considered for next hidden layer
- Useful for binary class only, not multi class
- The gradient of the step function is '0' which causes a hindrance in the back propagation process.

$$f'(x) = 0, \text{ for all } x$$

Linear Function

f(x)=ax



Python Code:

```
def linear_function(x):
    return 4*x
linear_function(4), linear_function(-2)
```

Output:

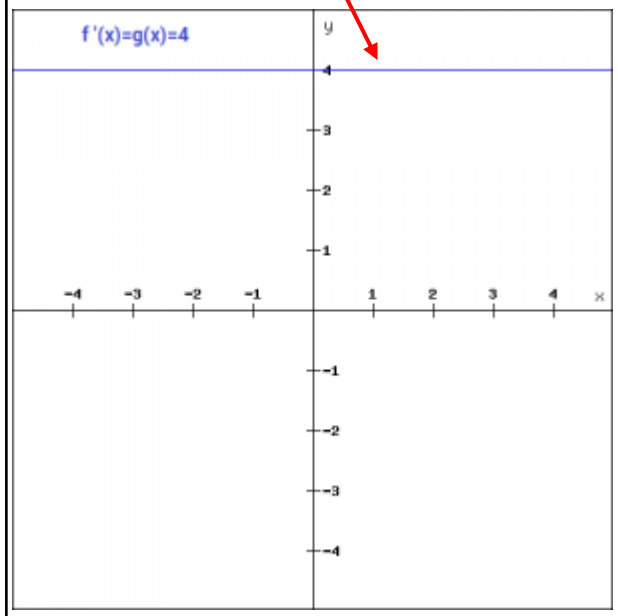
```
(16, -8)
```

- The activation is proportional to the input. The variable ‘a’ in this case can be any constant value
- Differentiate the function with respect to x, the result is the coefficient of x, which is a constant.
- Although the gradient here does not become zero, but it is a constant

$$f'(x) = a$$

Linear Function (Gradient)

f'(x) = a



Python Code:

```
def linear_function(x):
    return 4*x
linear_function(4), linear_function(-2)
```

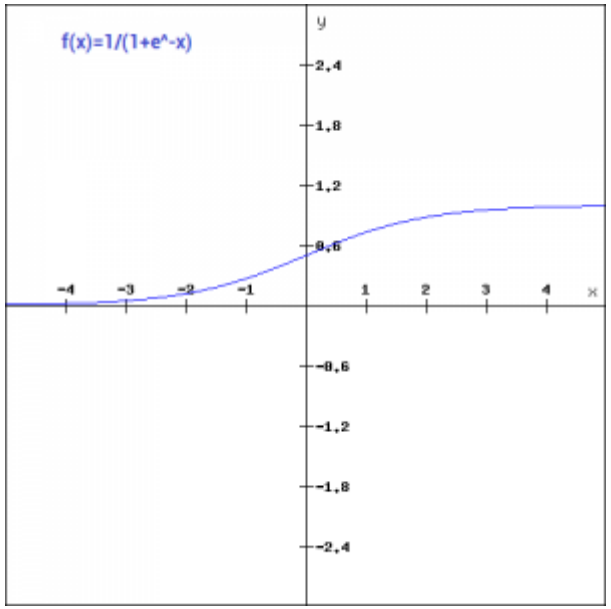
Output:

```
(16, -8)
```

- Implies during the backpropagation process weights & biases get updated with same updating factor.
- Neural network not improve the error as gradient is same for every iteration.
- Not suitable to capture the complex patterns from the data.

Sigmoid

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



Python Code:

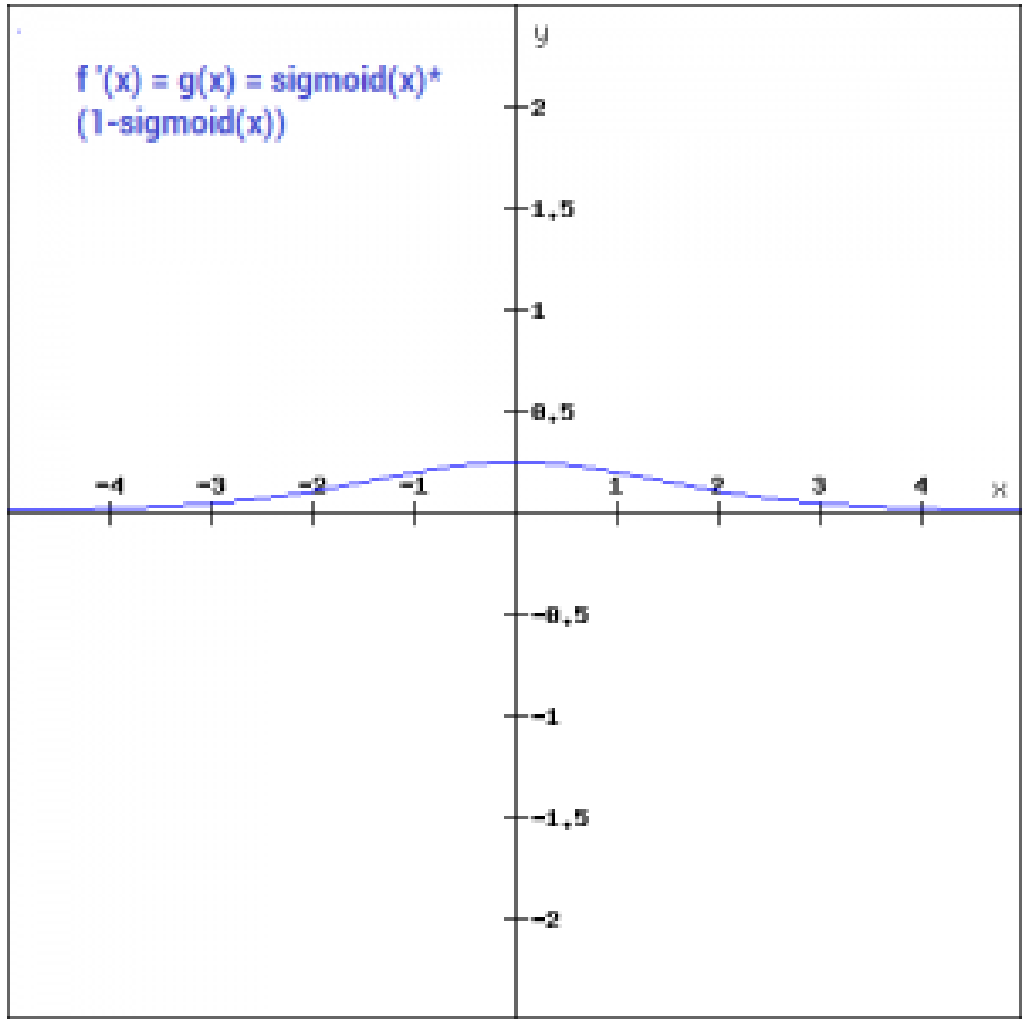
```
import numpy as np

def sigmoid_function(x):

    z = (1/(1 + np.exp(-x)))

    return z

sigmoid_function(7),sigmoid_function(-22)
```

Output:

```
(0.9990889488055994, 2.7894680920908113e-10)
```

- The sigmoid function is not symmetric around 0, thus output of all neurons is of same sign.
  - This can be addressed by scaling the sigmoid function which is exactly what happens in the *tanh function*
- As input values move away from 0, the output value becomes less sensitive. Even a large change in input values results in little to no change in the output value

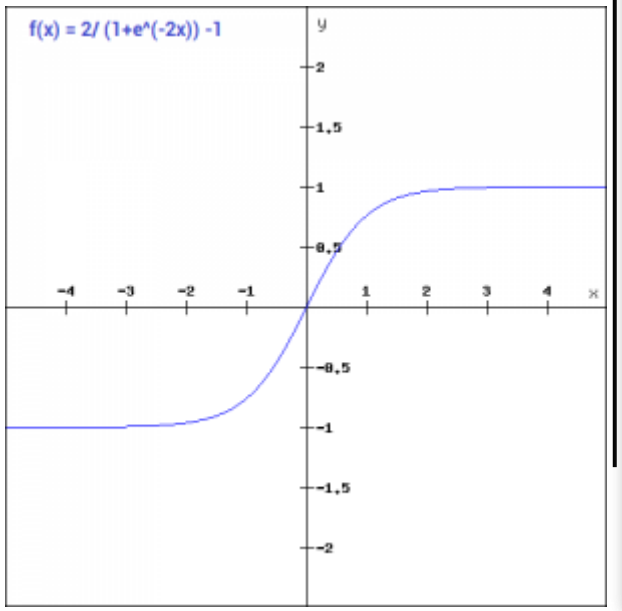
Sigmoid (gradient)



- Gradient values are significant for range -3 and 3 but graph is much flatter in other regions. This implies that for values > 3 or < -3, will have very small gradients. As the gradient value approaches zero, the network is not really learning.

Tanh

$\tanh(x) = 2/(1+e^{-2x}) - 1$



Python Code:

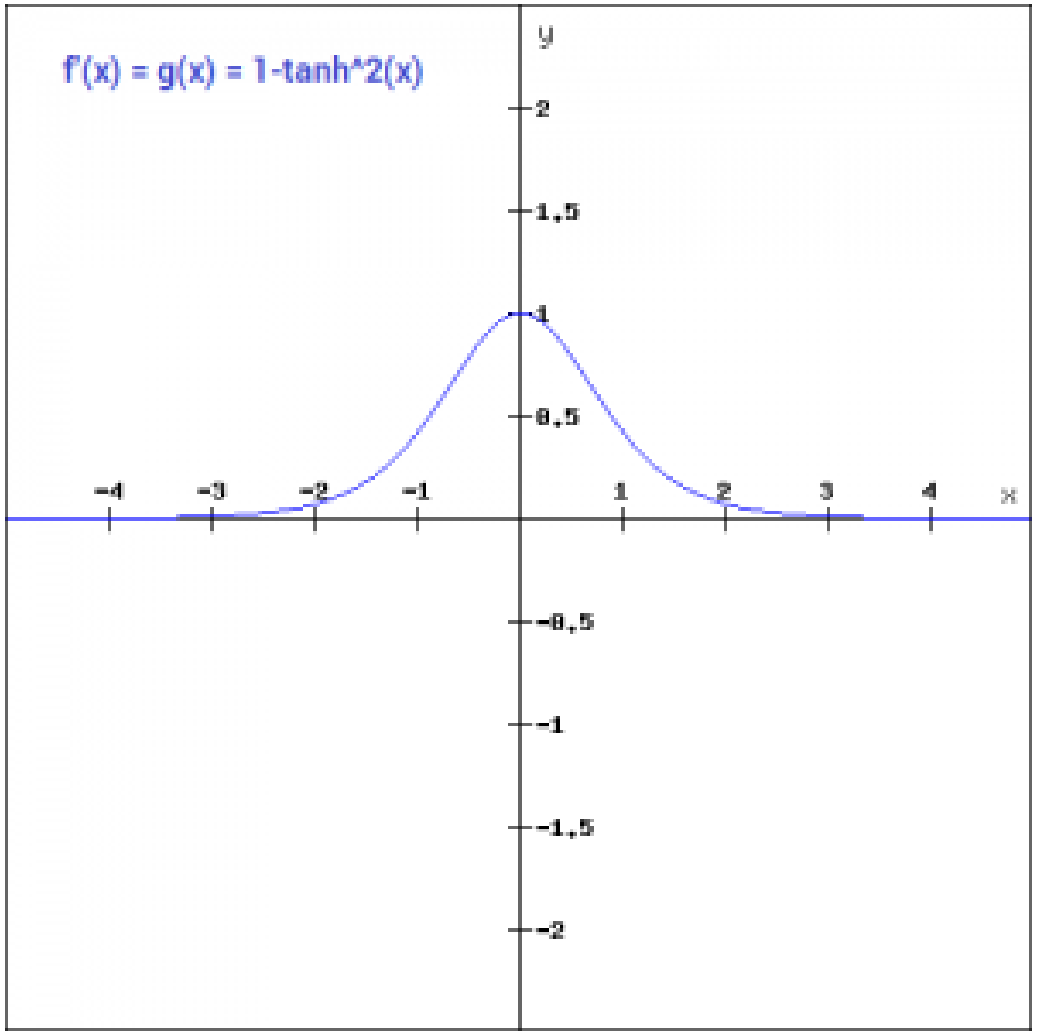
```
import numpy as np

def tanh_function(x):

    z = (2/(1 + np.exp(-2*x))) -1

    return z

tanh_function(0.5), tanh_function(-1)
```

Output:

```
(0.4621171572600098, -0.7615941559557646)
```

- Symmetric around the origin. The range of values in this case is from -1 to 1.
- Thus the inputs to the next layers will not always be of the same sign.

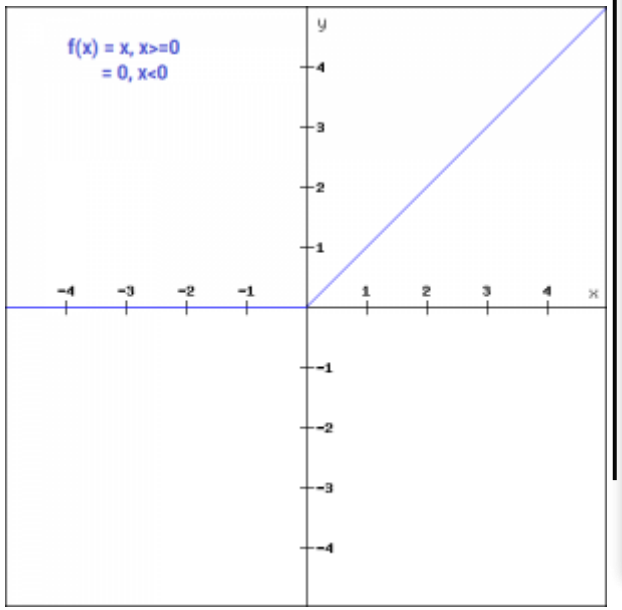
Tanh (gradient)



- The gradient is steeper as compared to the sigmoid function.
- Usually, tanh is preferred over the sigmoid function since it is zero centered & gradients are not restricted to move in a certain direction.

ReLu (Rectified linear Unit)

$f(x)=max(0,x)$



Python Code:

```
def relu_function(x):
    if x<0:
        return 0
    else:
        return x
relu_function(7),
relu_function(-7)
```

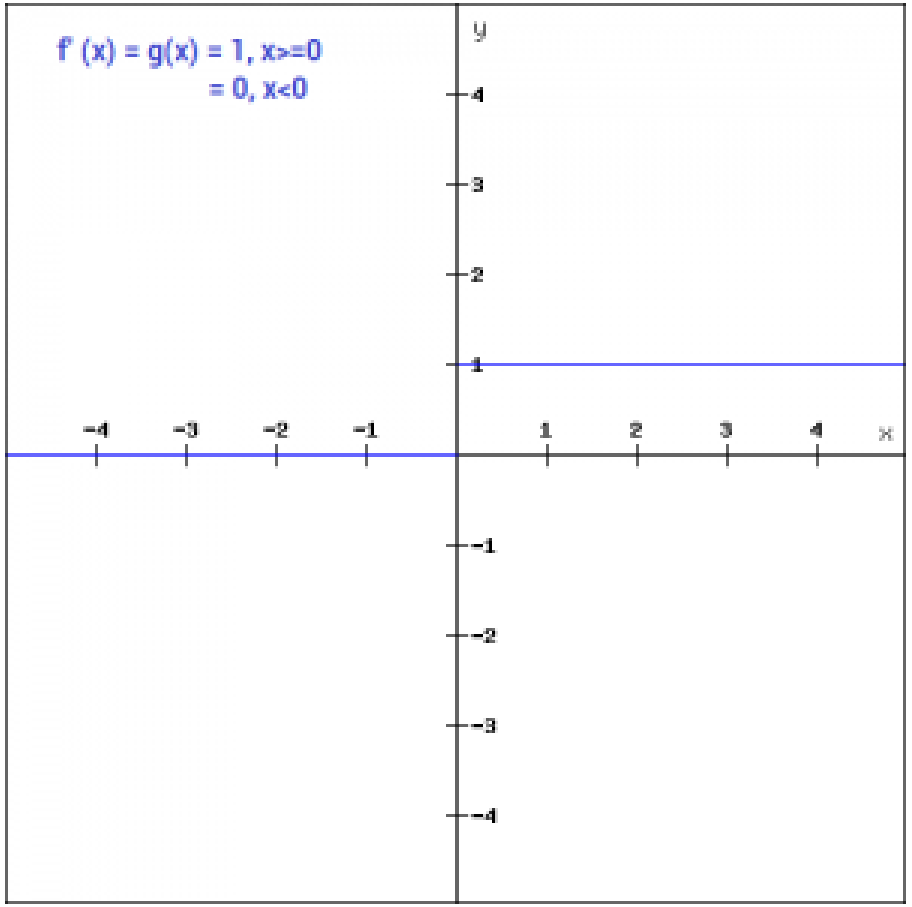
Output:

```
(7, 0)
```

- As compared to other activation function it does not activate all the neurons at same time
- The neurons will only be deactivated if the output of linear transformation is < 0.
- Since only a certain no. of neurons are activated, the ReLU is far more computationally efficient compared to sigmoid & tanh function

ReLu (gradient)

f'(x) = 1, x>=0, f'(x) = 0, x<0

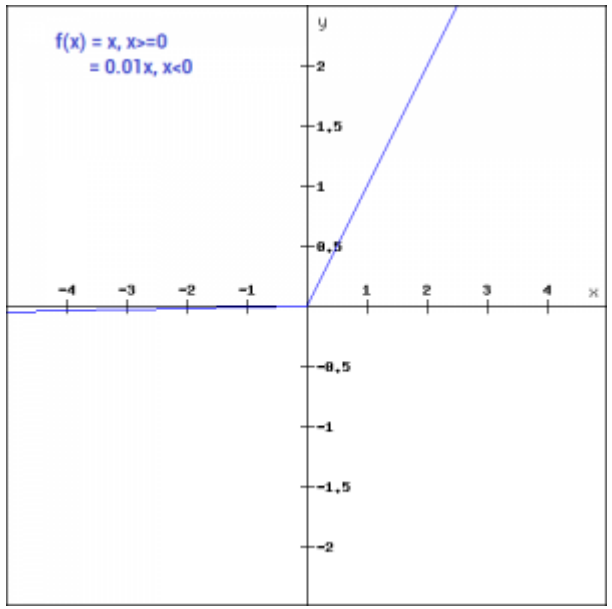


- At negative side of graph, the gradient value is 0. Thus during the backpropogation process, the weights and biases for some neurons are not updated.
- This may create dead neurons which never get activated. This is taken care of by the *'Leaky' ReLU function*.

Leaky ReLu

f(x)= 0.01x, x<0

= x, x>=0



Python Code:

```
def
leaky_relu_function(x):
    if x<0:
        return 0.01*x
    else:
        return x
leaky_relu_function(7),
leaky_relu_function(-7)
```

Output:

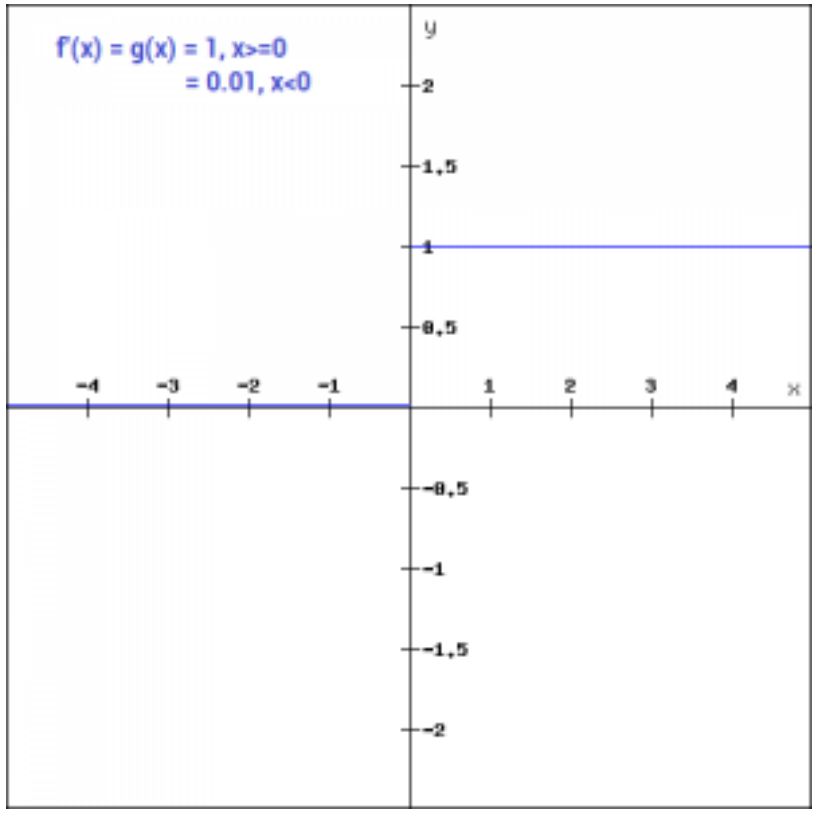
```
(7, -0.07)
```

- The ReLU function, the gradient is 0 for x<0, which would deactivate the neurons in that region
- Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x, we define it as an extremely small linear component of x
- Hence we would no longer encounter dead neurons in that region

Leaky ReLu (gradient)

f'(x) = 1, x>=0

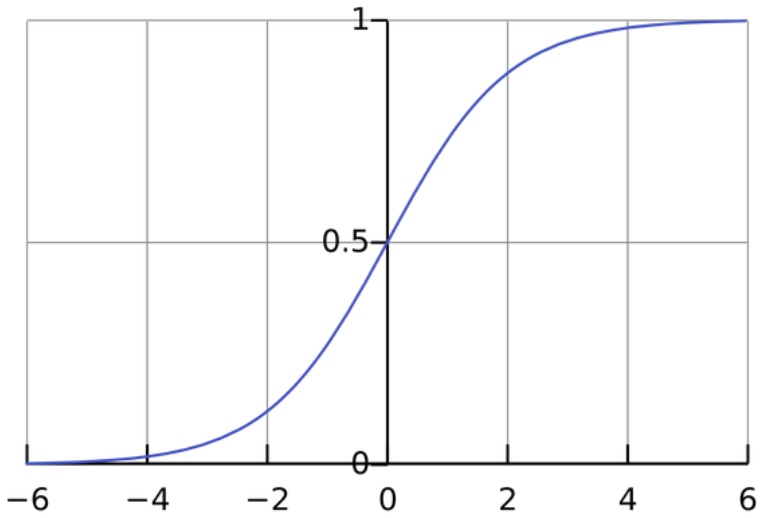
=0.01, x<0



- By making this small modification, the gradient of the left side of the graph comes out to be a non-zero value.
- Hence it no longer encounter dead neurons in that region.

Softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



Python Code:

```
def softmax_function(x):
    z = np.exp(x)
    z_ = z/z.sum()
    return z_
softmax_function([0.8, 1.2, 3.1])
```

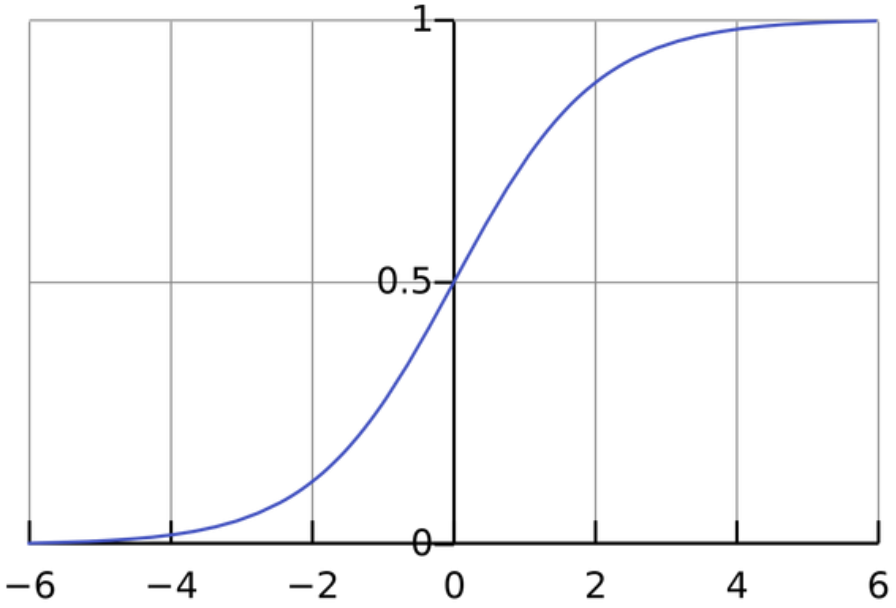
Output:

```
array([0.08021815, 0.11967141, 0.80011044])
```

- Softmax function is a combination of multiple sigmoids. Since, sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class..
- The softmax function can be used for multiclass classification problems. This function returns the probability for a datapoint belonging to each individual class

Softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



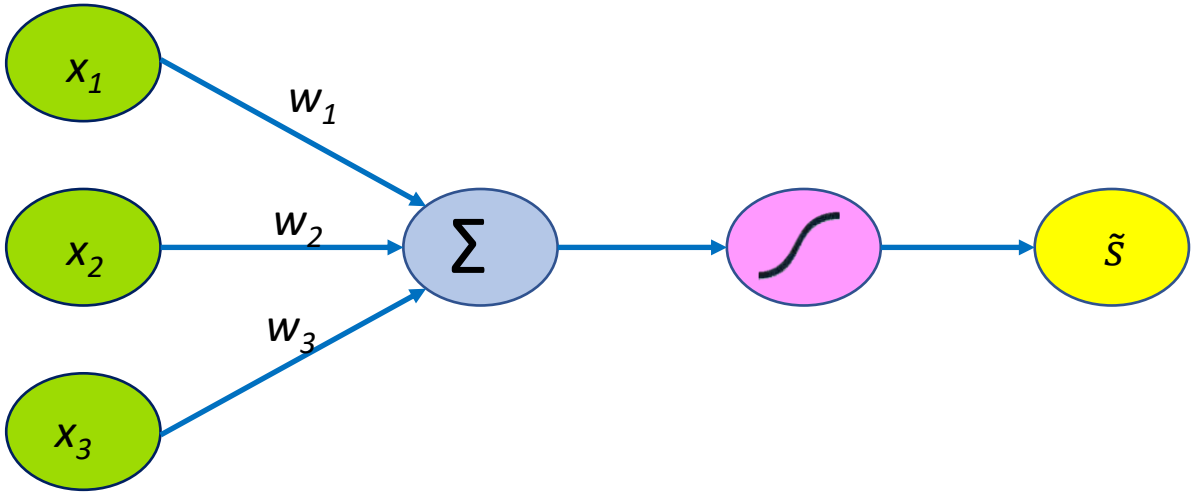
- For a multiclass problem, the output layer would have as many neurons as the number of classes in the target. if you have three classes, there would be three neurons in the output layer. Let the output from the neurons as [1.2 , 0.9 , 0.75].
- Applying the softmax function over these values, you will get the following result – [0.42 , 0.31, 0.27].
- These represent the probability for the data point belonging to each class.

Note that the sum of all the values is 1

Building Neural Networks with Perceptron

The Perceptron: Basic neural network building block

- Perceptron is a structural building block of deep learning.
- **Perceptron is a single layer neural network** and a multi-layer perceptron is called Neural Networks.



a. All the inputs **x** are multiplied with their respective weights **w**.
b. Add all the multiplied values and call them Weighted Sum.
c. Apply that weighted sum to the correct **Activation Function** to get the output (s)

Non-linear activation function

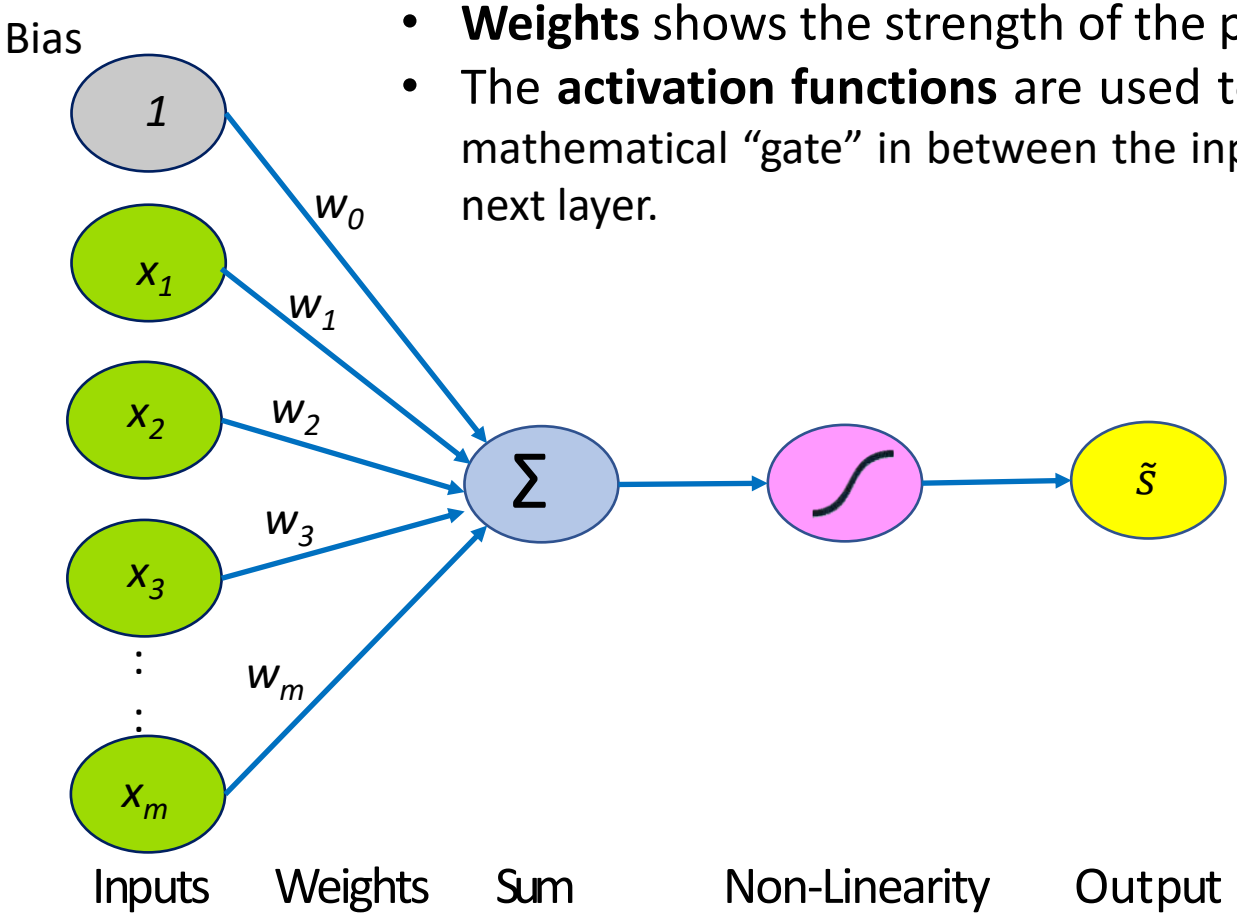
Linear combination of inputs

$$\tilde{s} = F\left(\sum_{i=1}^{m} x_i w_i\right)$$

Output

The Perceptron: Forward Propagation

- **Bias** value allows you to shift the activation function left or right regardless of the input.
- **Weights** shows the strength of the particular node.
- The **activation functions** are used to map the input between the required values. It is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer.



Mathematical Function

Output $\tilde{S}$ = $F(\ w_0 + \sum_{i=1}^{m} x_i\, w_i)$ Linear combination of inputs

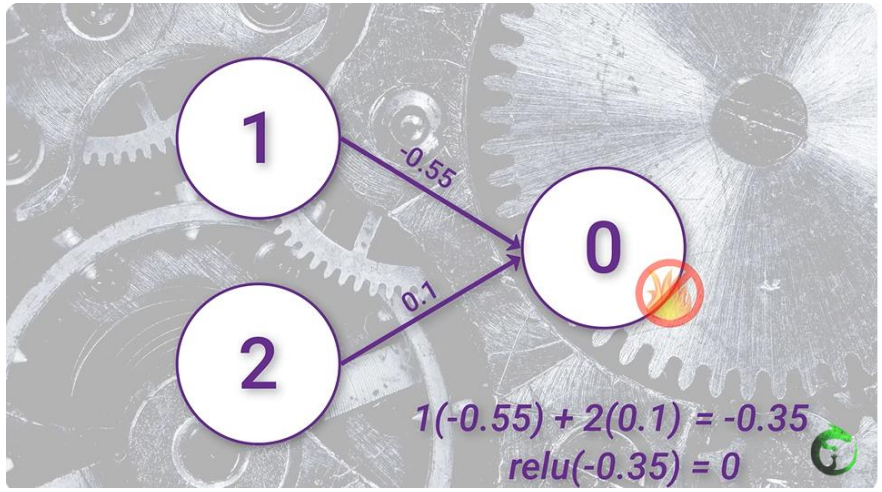
Non-linear activation function

Bias

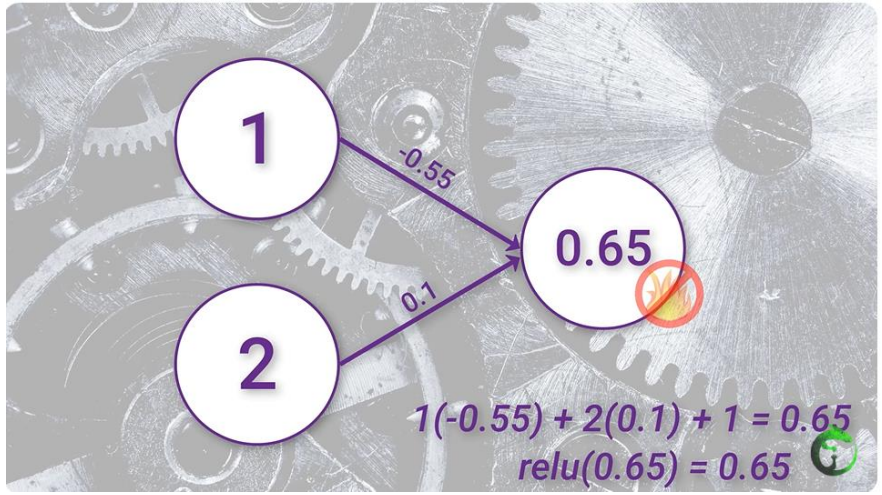
Bias in an Artificial Neural Network

Importance of Bias in neural network

1. **Handling Offsets and Shifts:**
   - It helps the network account for any systematic offsets or shifts in the input features. Without bias, the network might struggle to fit the data properly.
2. **Expressiveness:**
   - They provide additional degrees of freedom, enabling the network to learn more complex and flexible decision boundaries. This is important for capturing non-linear relationships in the data.
3. **Aiding Learning:**
   - During the training process, the bias terms are adjusted along with the weights to minimize the error in the predictions. This helps the network adapt to the specific characteristics of the training data and generalize well to unseen data.
4. **Addressing Imbalances:**
   - In some cases, the data might be imbalanced, meaning that certain classes or ranges of input values are more prevalent than others. Bias terms can help the network account for these imbalances and avoid being skewed towards the majority class or input range.
5. **Improved Model Performance:**
   - Including bias terms often leads to better model performance. They allow the neural network to learn and represent more complex functions, making it more capable of capturing the underlying patterns in the data.

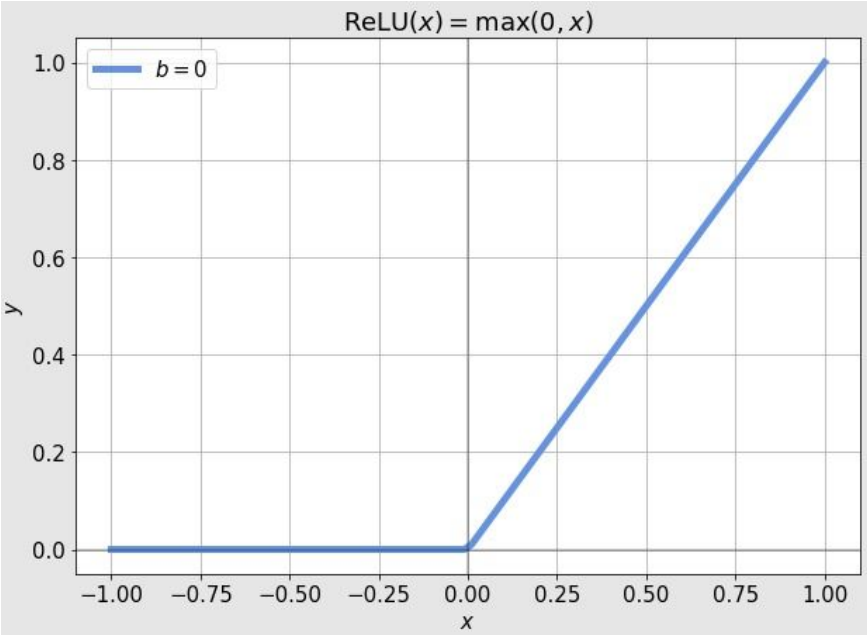


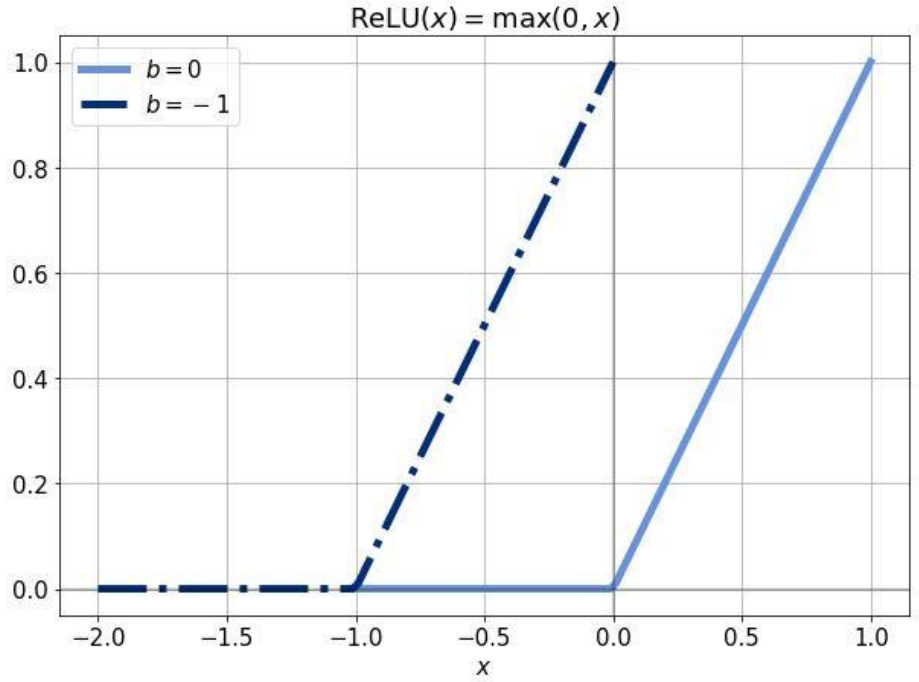
Bias = 0, The Neuron has not fired



Bias = 1, The Neuron has fired

Bias in an Artificial Neural Network

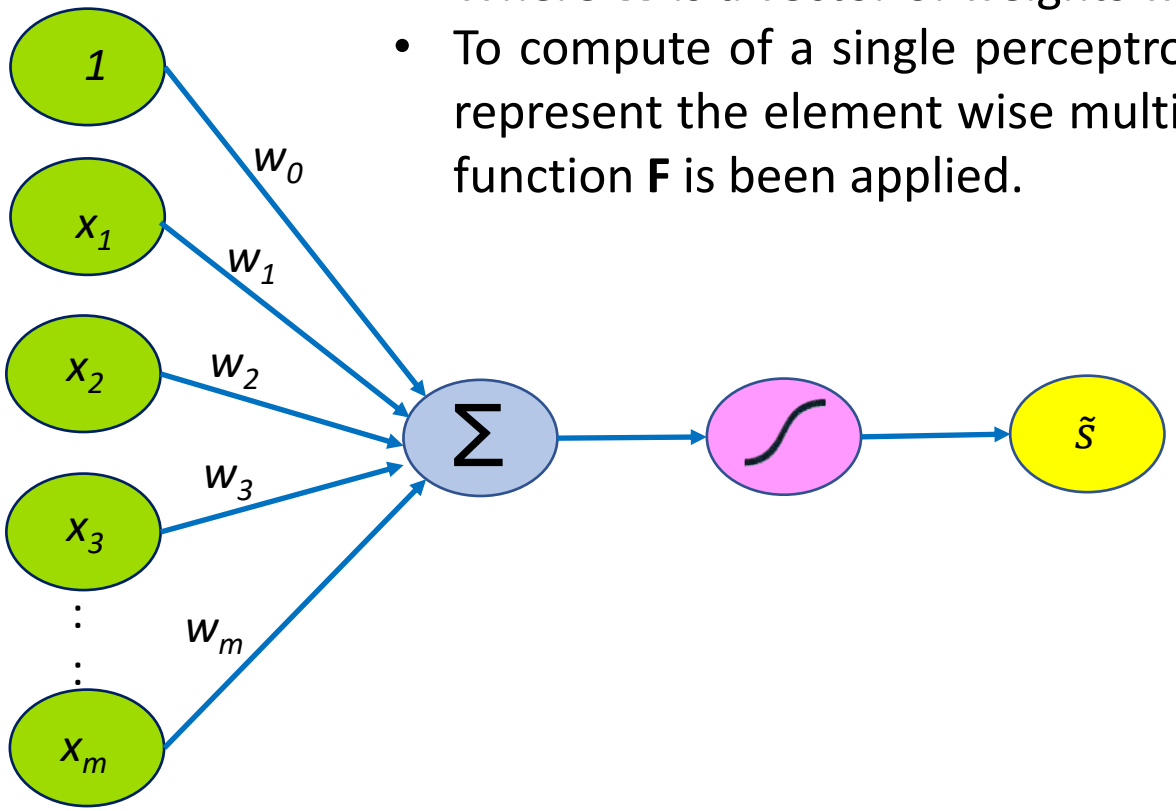




- For all the values of x less than or equal to 0, the neural network will have the issue of vanishing gradient. This occurs when the input approaches zero or negative and the gradient approaches zero. Thus, the network cannot perform backpropagation and cannot learn. This problem is not restricted to ReLU but also applies to other activation functions like sigmoid, tanh, etc.
- If we include the bias term ‘b’ in the activation function, it would allow the neural network to shift the activation function to the left and to the right by simply modifying the values of b. It would allow the initialization of the derivatives of the error function to non-zero for the values of x between 1 and 0, as shown in the Figure above.

The Perceptron: Forward Propagation

- After summation of all the x's value into a vector X, which is now a vector of inputs x
- Where **W** is a vector of weights w's.
- To compute of a single perceptron take dot product of input X and weight W which represent the element wise multiplication and summation and then linear activation function **F** is been applied.

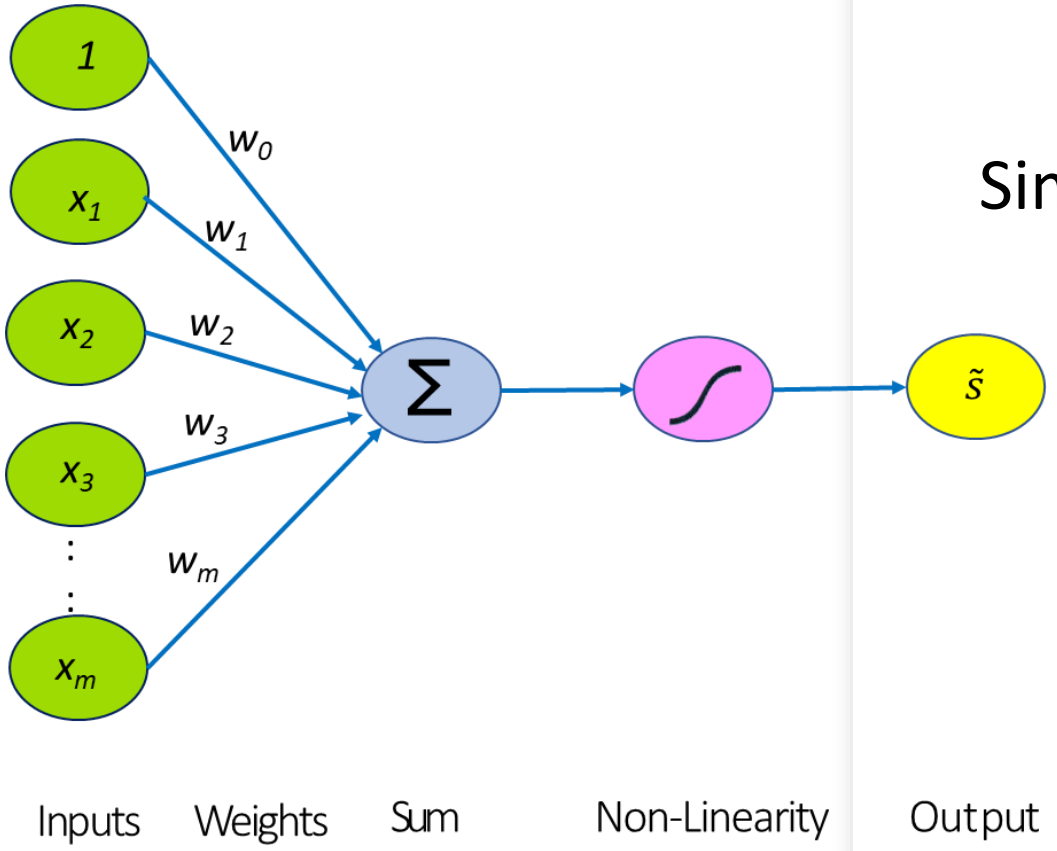


$$\tilde{s} = F\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

$$\tilde{s} = F\left(w_0 + X^T W\right)$$

$$\text{where: } X = \begin{bmatrix} X_1 \\ \vdots \\ X_m \end{bmatrix} \text{ and } W = \begin{bmatrix} W_1 \\ \vdots \\ W_m \end{bmatrix}$$

The Perceptron: Simplified



Simple equation of perceptron working

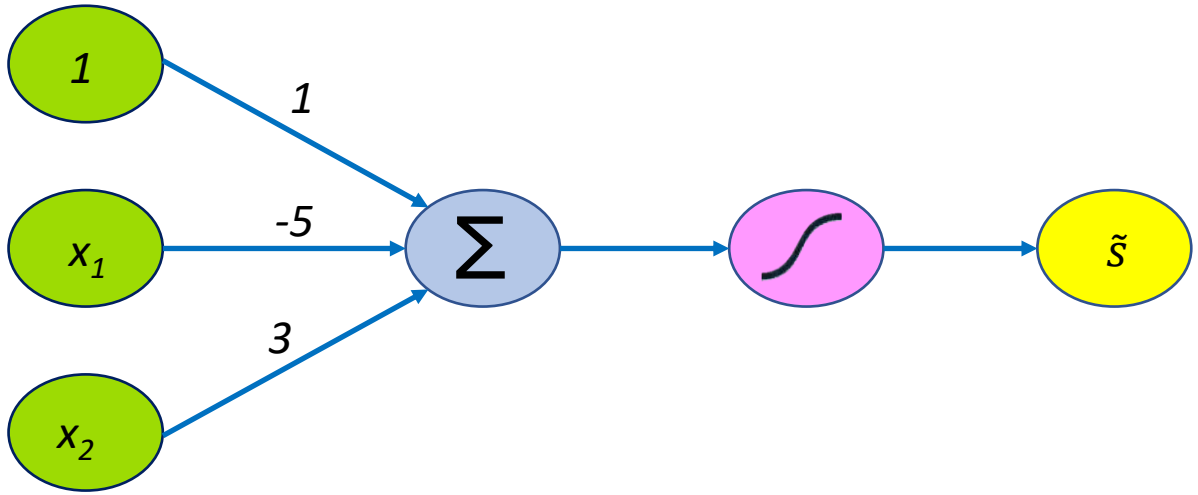
$$\tilde{s} = F(w_0 + X^T W)$$

Input Vector Weight Vector

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

Example for Perceptron

- Let x series be the input with weights $w_0 = 1$, where w_1 and w_2 be -5 and 3 applying on the same formula for getting the output.
  - Multiplying the inputs with the respective weights.
  - Take the sum of the result and pass through the activation function to get the output $\tilde{s}$.



$$W_0 = 1 \quad \text{and} \quad W = \begin{bmatrix} -5 \\ 3 \end{bmatrix}$$

$$\tilde{s} = g(W_0 + X^T W)$$

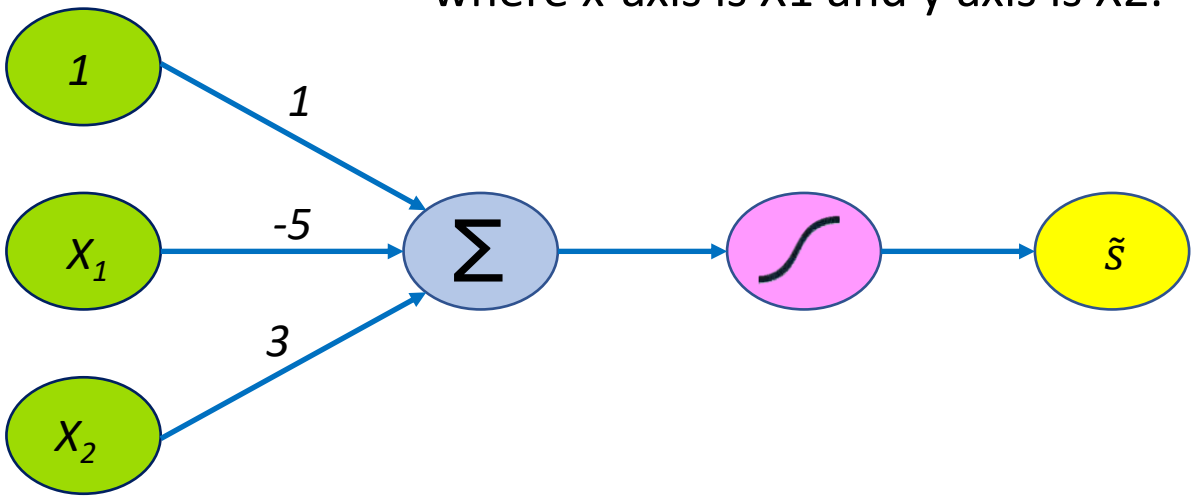
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} -5 \\ 3 \end{bmatrix}\right)$$

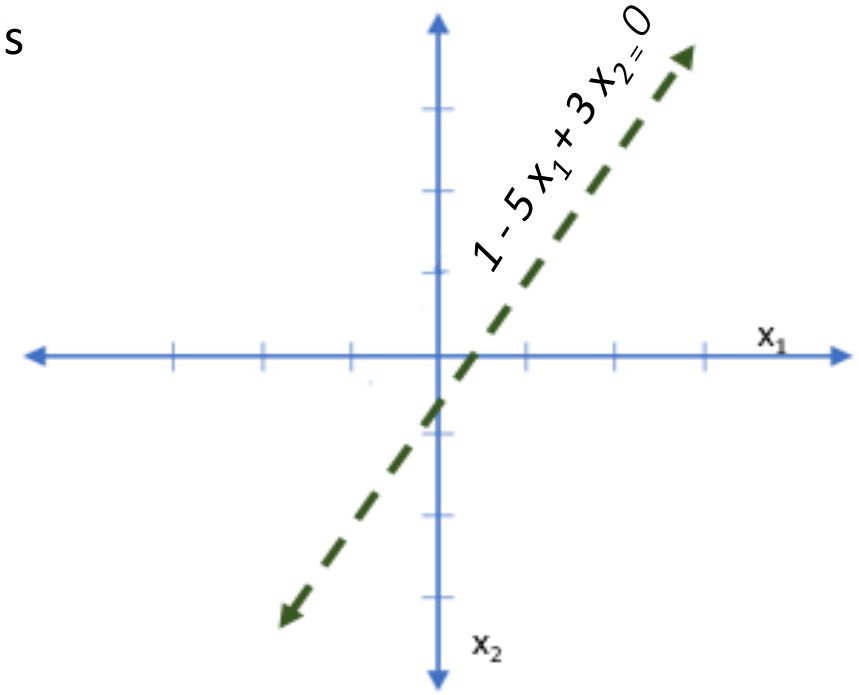
$$\tilde{s} = \underbrace{(1 - 5x_1 + 3x_2)}_{\text{Output}}$$

Example for Perceptron

$$\tilde{s} = (1 - 5\,x_1 + 3\,x_2)$$

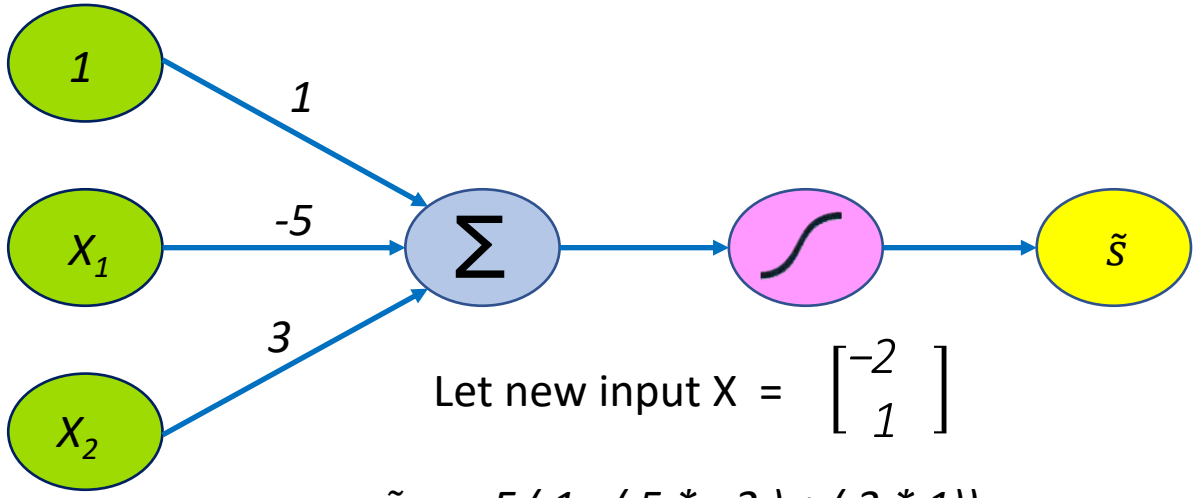
Graph generated for the output equation is where x-axis is X1 and y axis is X2.





This perceptron acts like a **linear classifier** where the decision boundary is the straight line which is defined by the equation. If input happens to lie on or on the right side of this line then you have an activation or an output will fire if it happens to lie on the left side of the line the output will not fire – hence it behaves like a linear classifier. By simply changing the weights and the bias one can change the position and orientation of this line.

Example for Perceptron

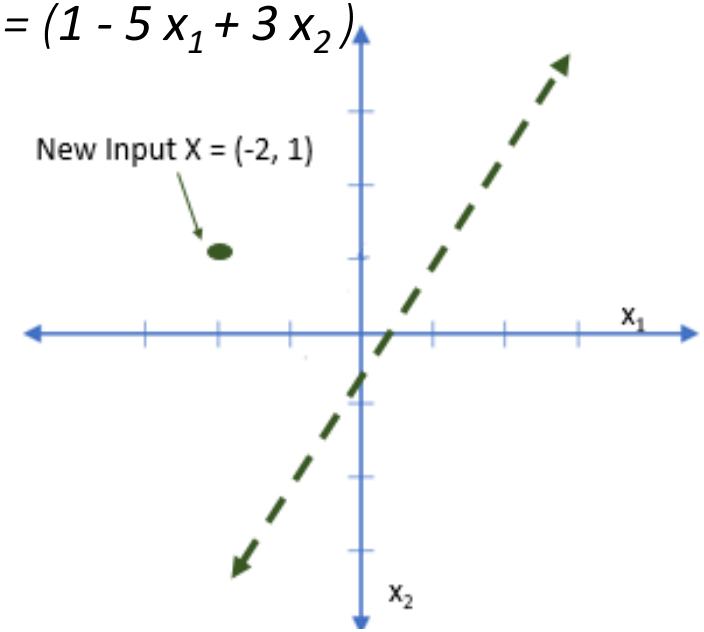


Let new input X = $\begin{bmatrix} -2 \\ 1 \end{bmatrix}$

$\tilde{s} = F(1 - (5 * -2) + (3 * 1))$

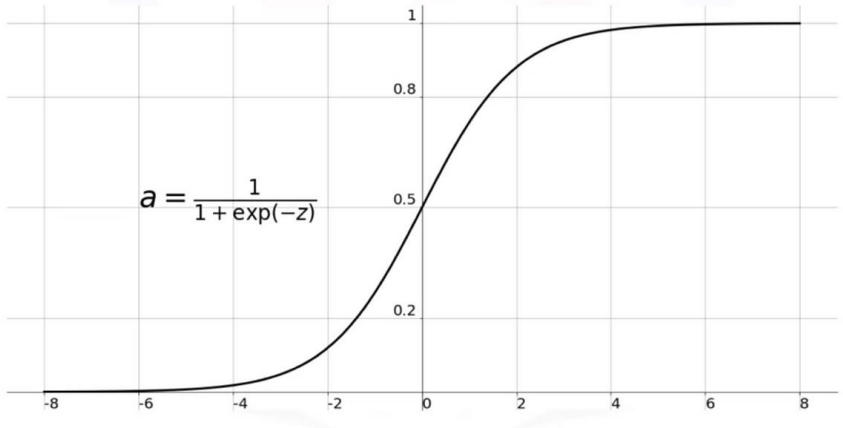
$= F(-6) \approx 0.002$

$\tilde{s} = (1 - 5x_1 + 3x_2)$



Sigmoid Function

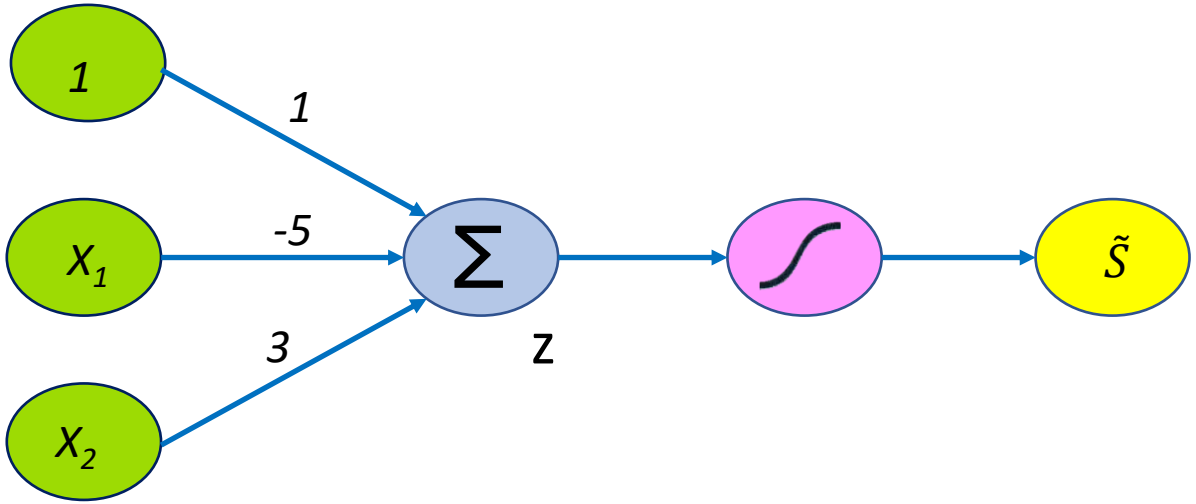
- With a new input value X or test example on the generated equation gives the result -6. After applying the sigmoid function it results into very low output i.e. 0.002
- Because sigmoid collapse everything into 0 and 1. Anything greater than 0 is going to be greater than 0.5 and anything below 0 is going to be less than 0.5.

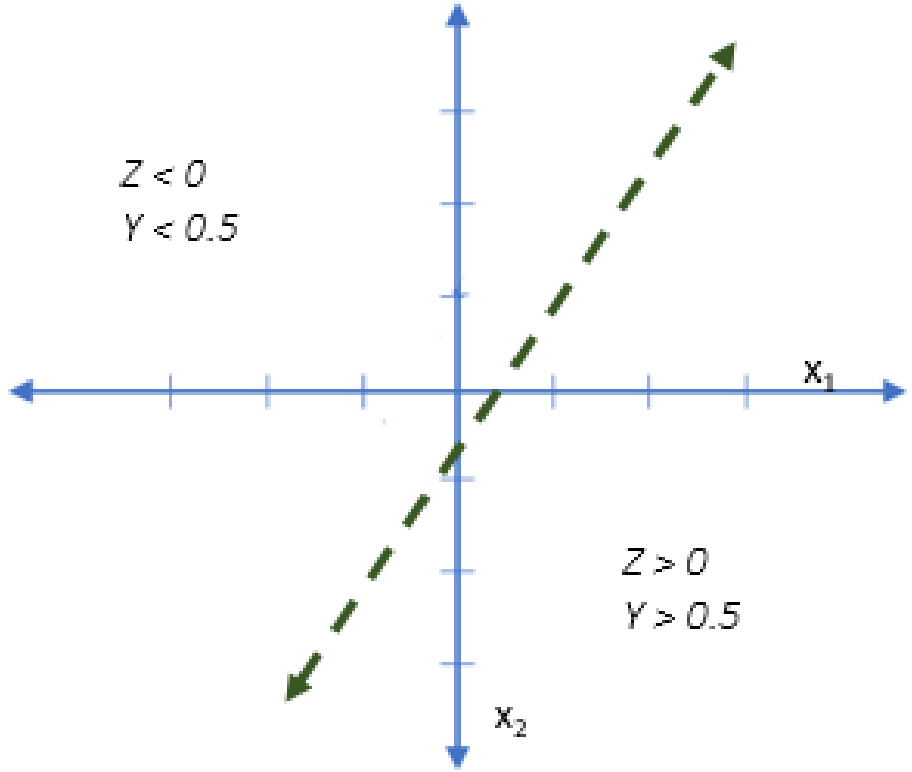


Generalizing the idea of sigmoid function for entire feature space for any point in the entire plot if the neurons lies in the left side of the line means neurons will be less than 0.5 and right side means greater than 0.5.

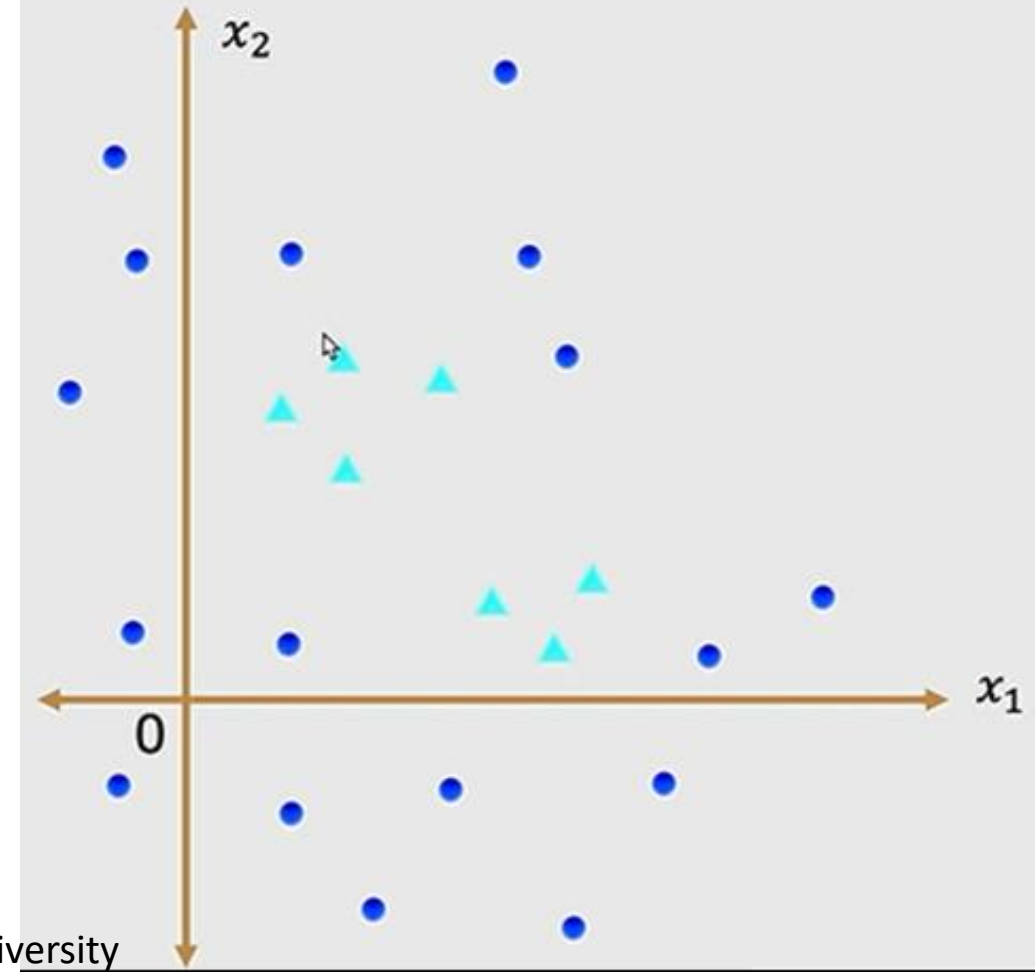
$$\tilde{s} = g(1 - 5x_1 + 3x_2)$$

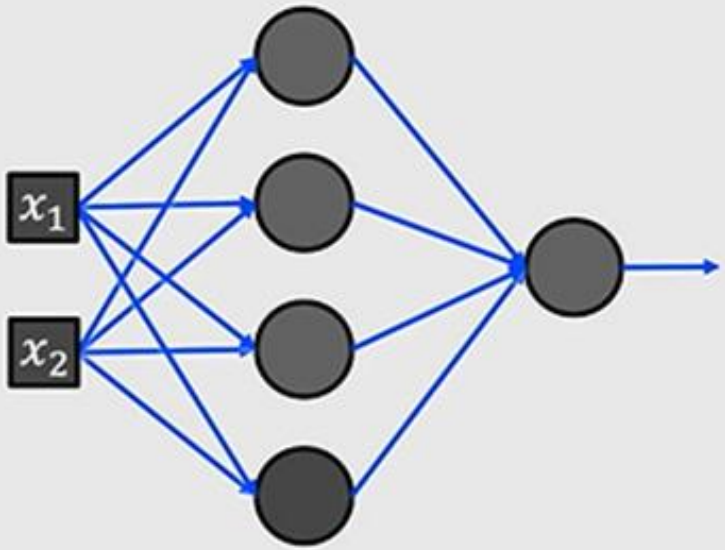
Z will be less then 0 if it lies on the left side and greater than 0 if on right side. (*Z be the output of the dot product that is elementwise multiplication of input with respective weights and that's get fed into activation function)*





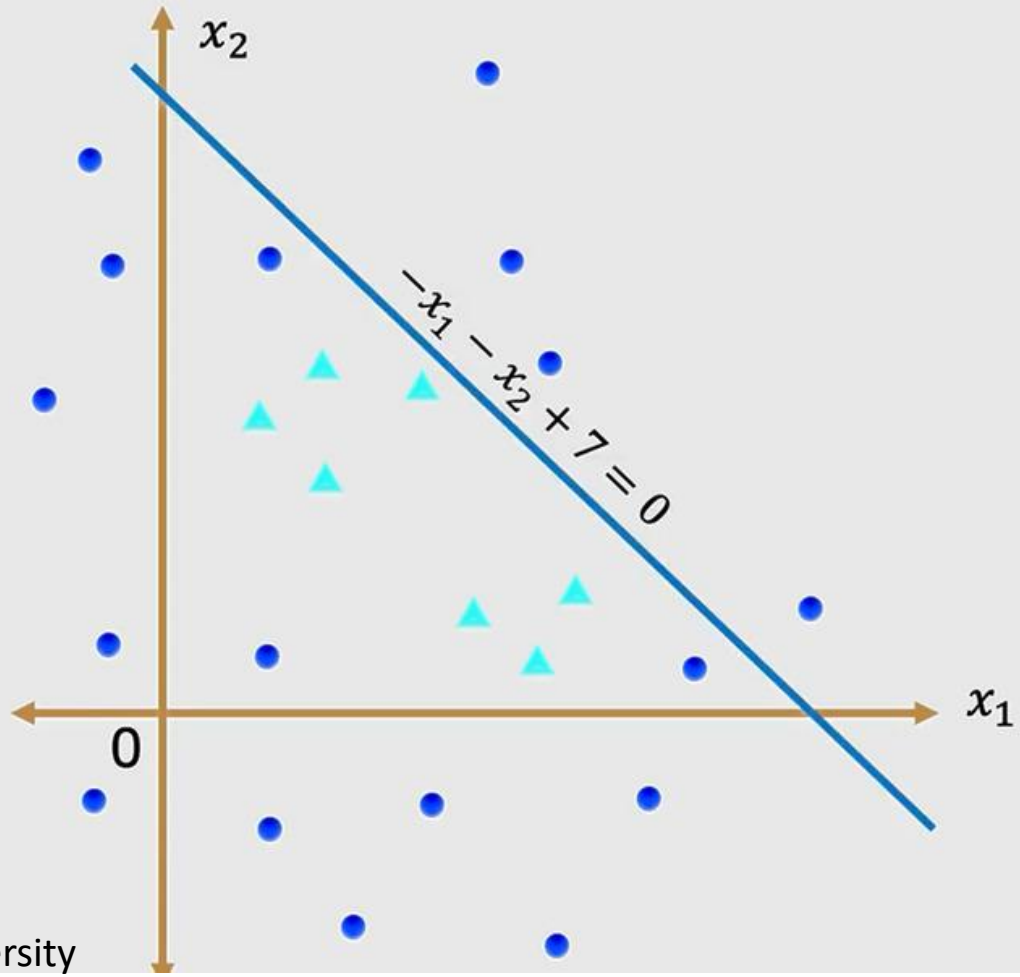
Linear Classifier for Complex Regions

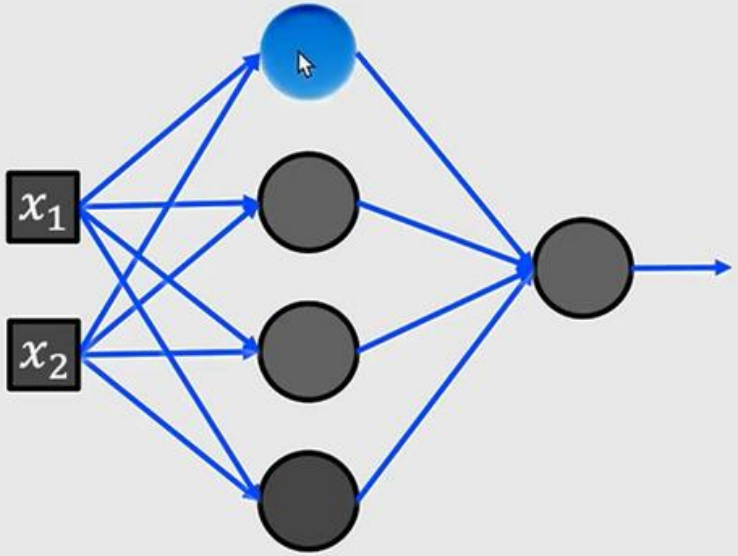




Courtesy: Columbia University

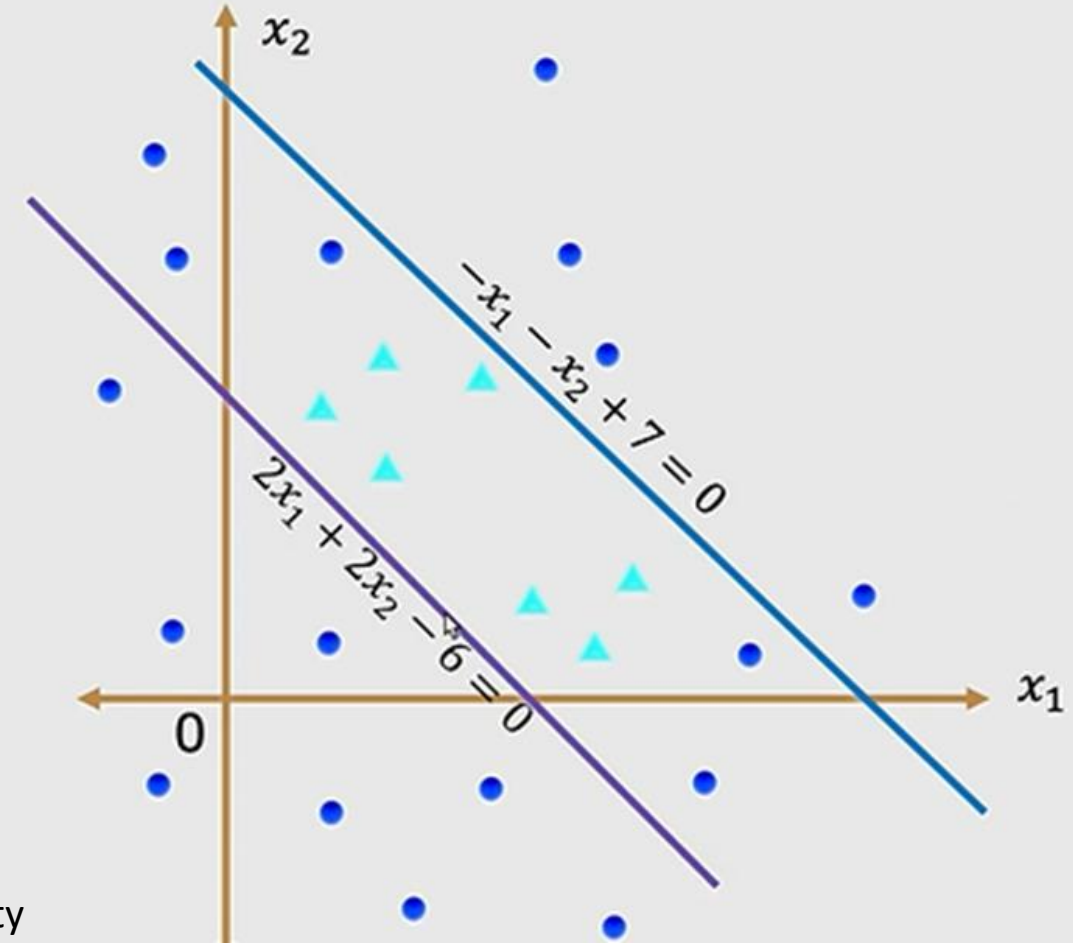
Linear Classifier for Complex Regions

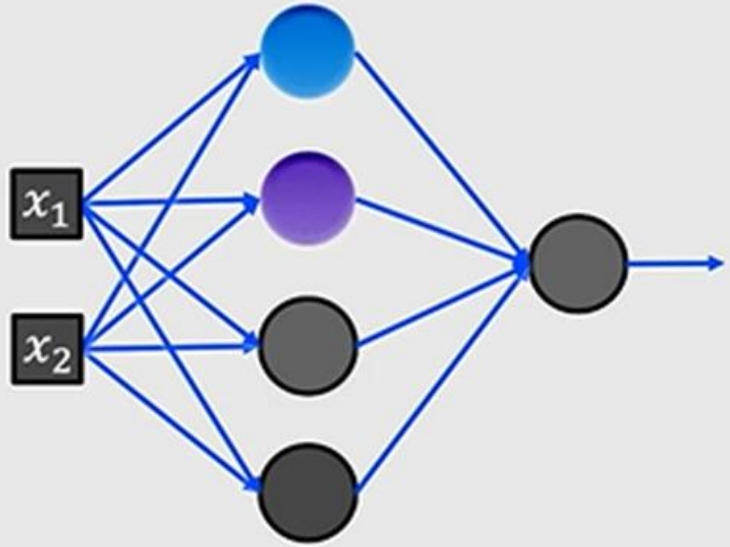




Courtesy: Columbia University

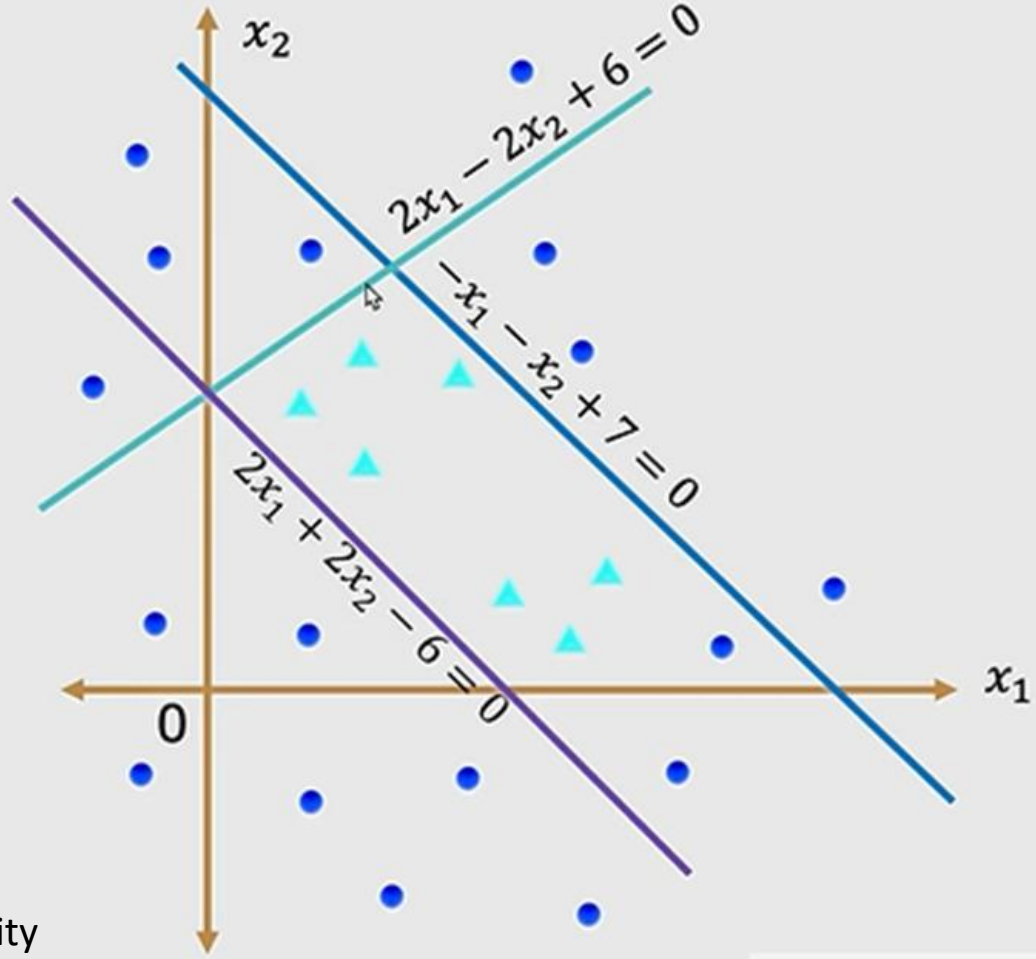
Linear Classifier for Complex Regions

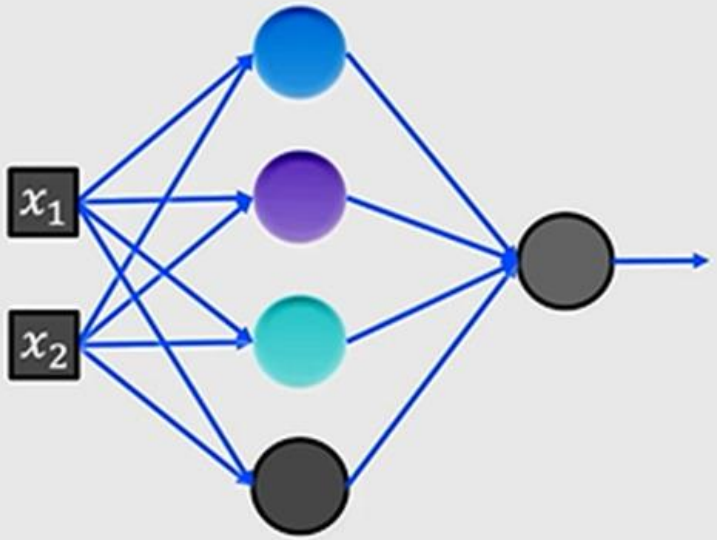




Courtesy: Columbia University

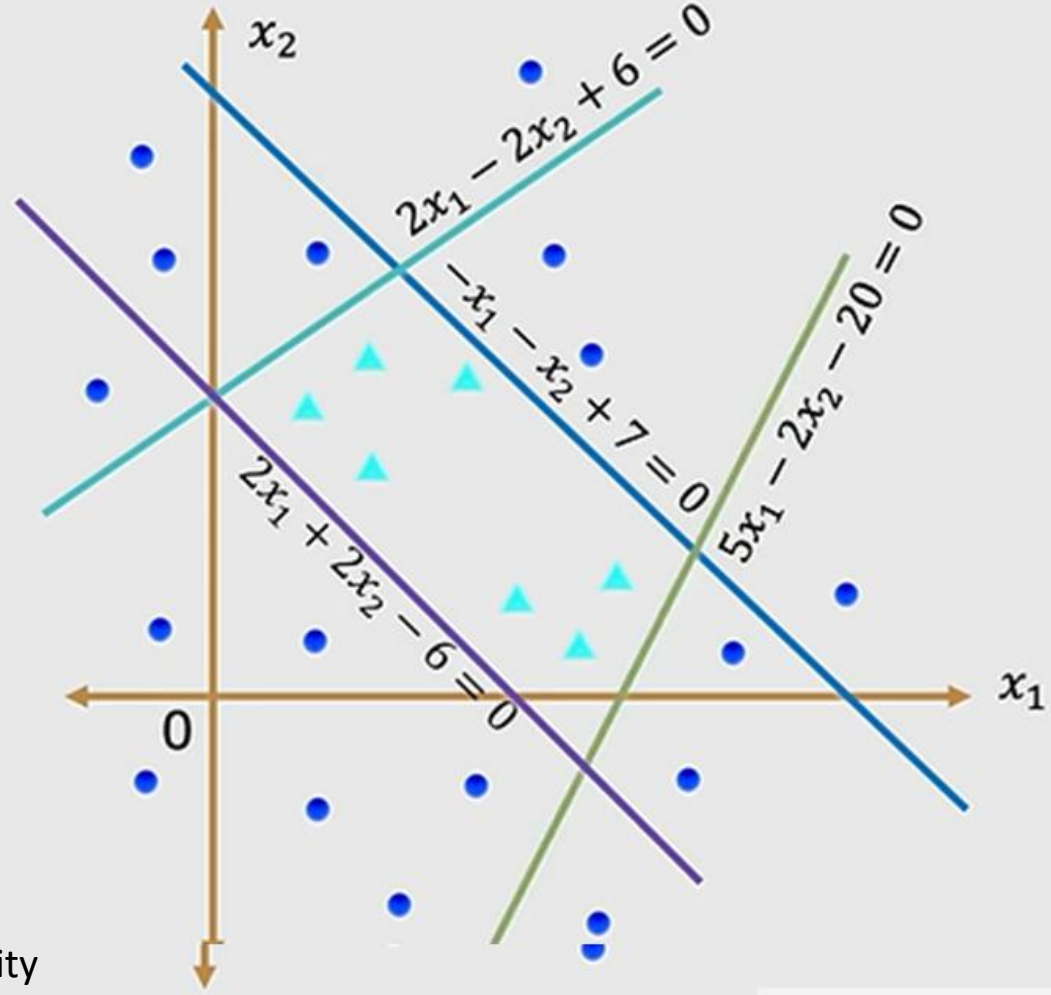
Linear Classifier for Complex Regions

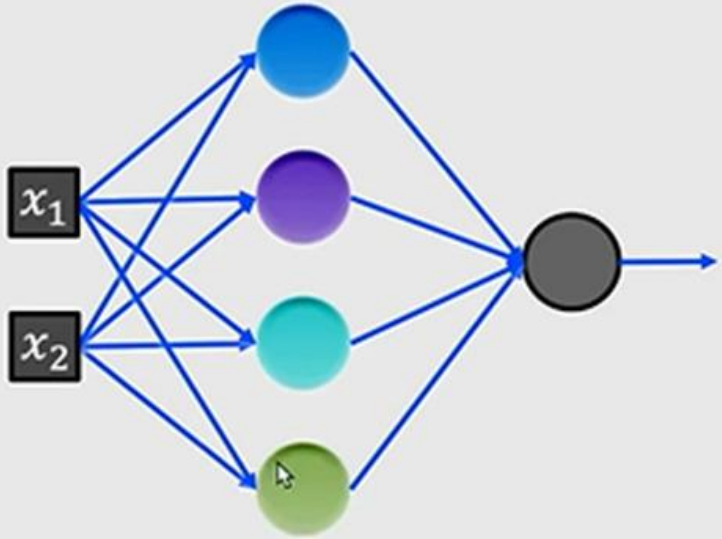




Courtesy: Columbia University

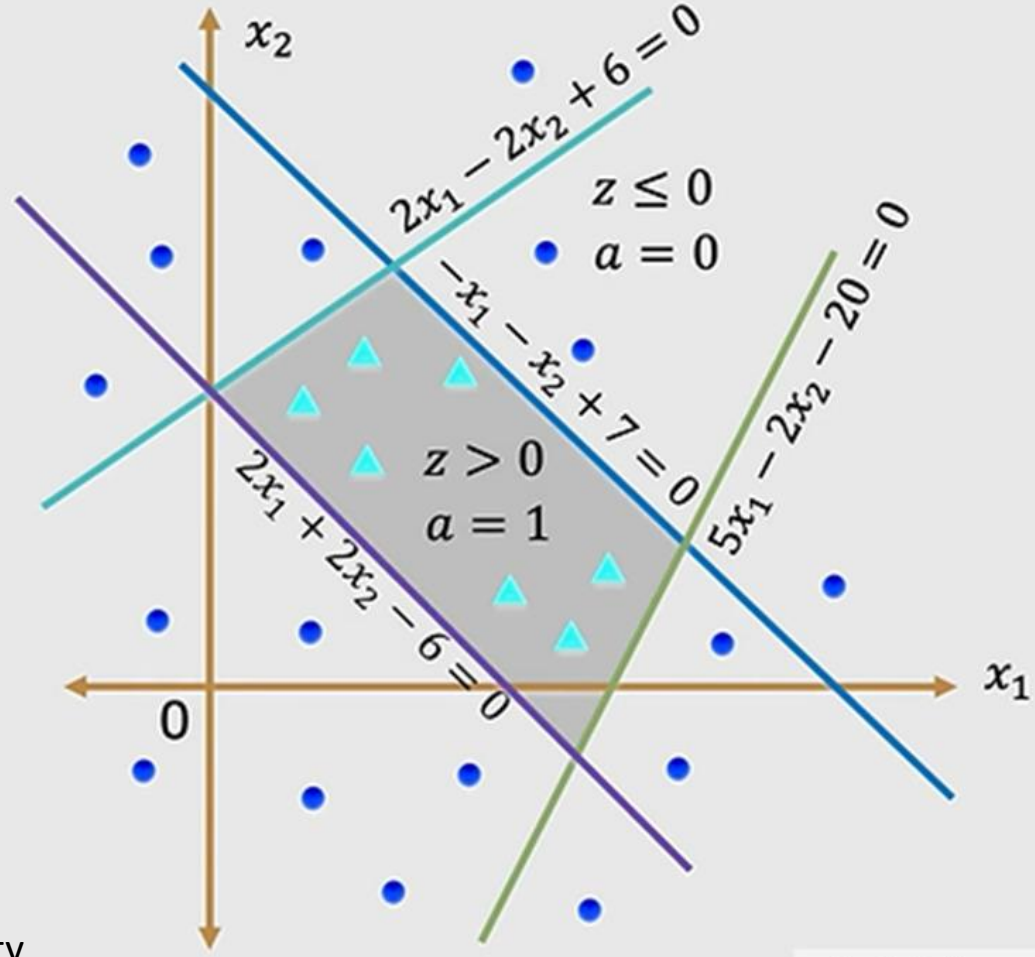
Linear Classifier for Complex Regions

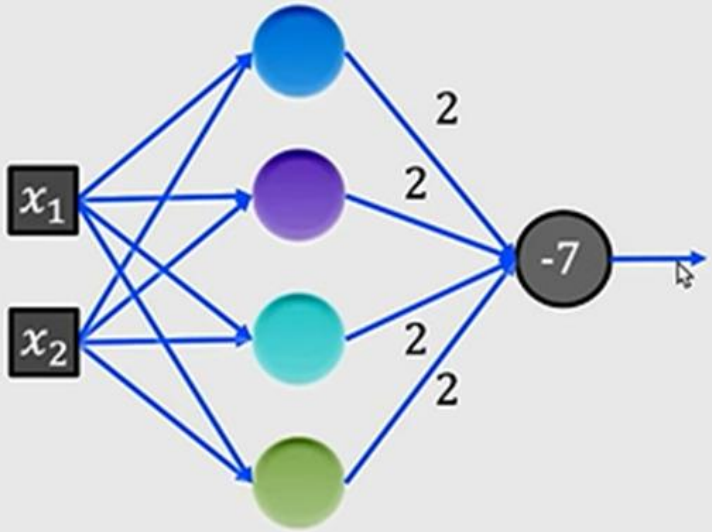




Courtesy: Columbia University

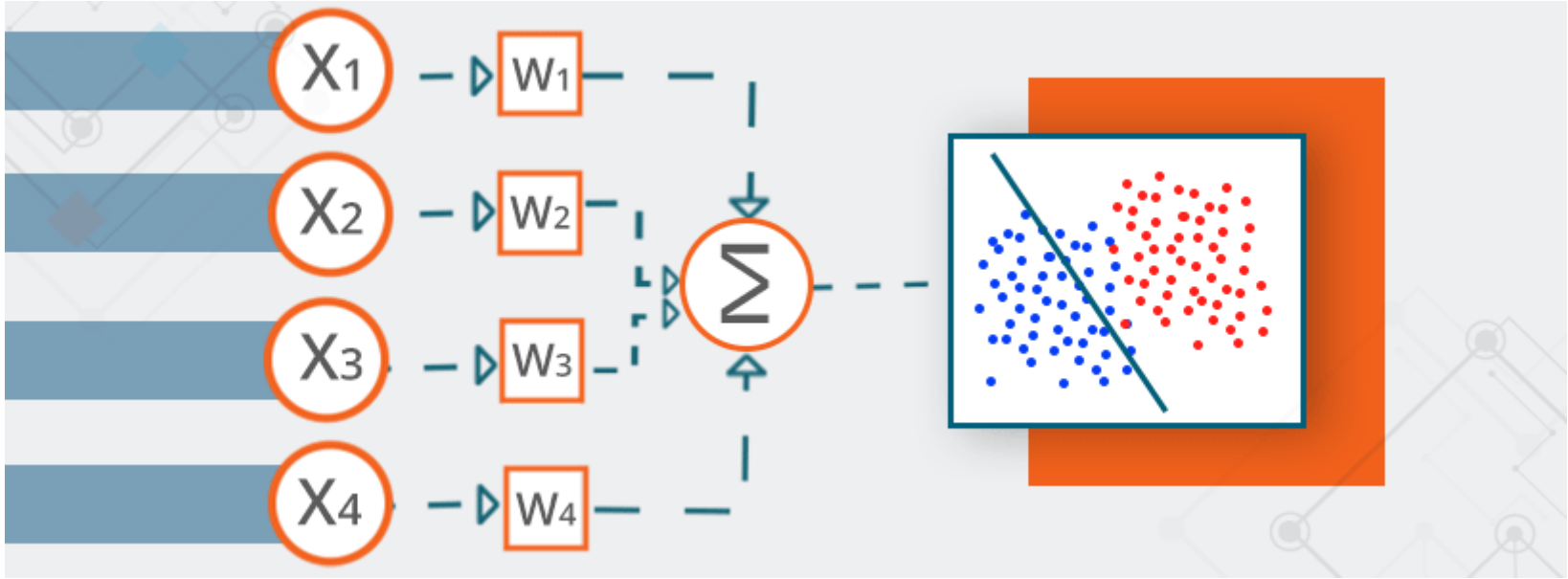
Linear Classifier for Complex Regions





Courtesy: Columbia University

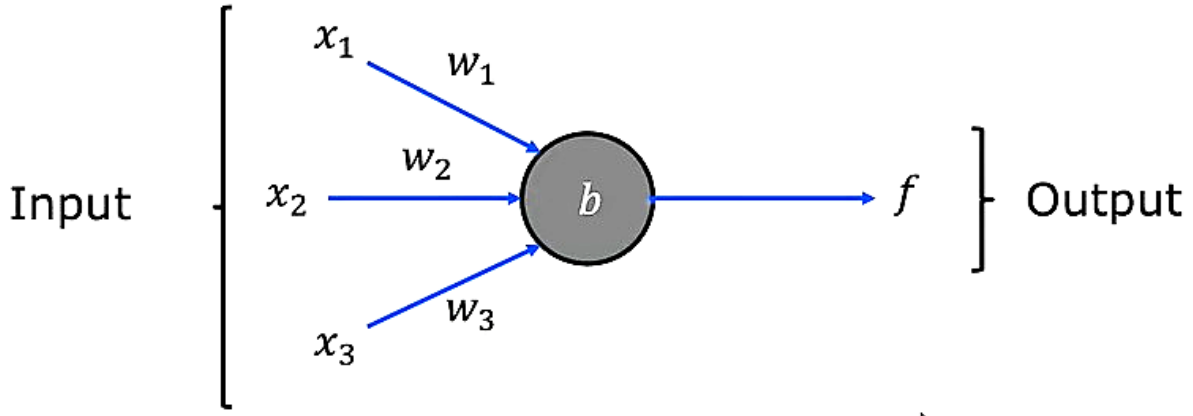
Perceptron Learning



Source: Edureka

Perceptron

Takes several inputs and gives a single binary output (Decision)



$$f = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq -b \\ 1 & \text{if } \sum_j w_j x_j > -b \end{cases} \qquad f = \begin{cases} 0 & \text{if } \boldsymbol{w}.\boldsymbol{x} + b \leq 0 \\ 1 & \text{if } \boldsymbol{w}.\boldsymbol{x} + b > 0 \end{cases}$$

where, $w_j = weights$

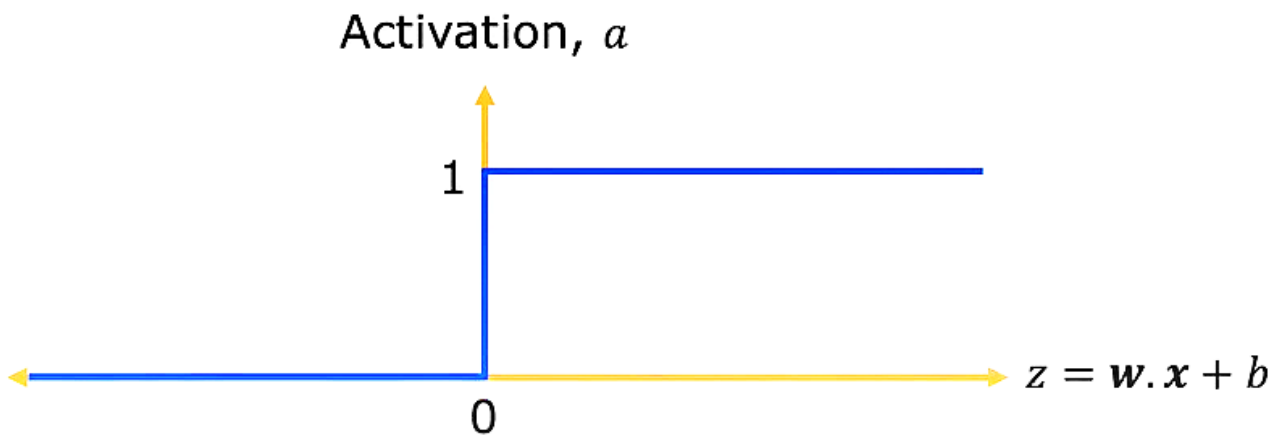
$b = bias\ (threshold)$

Courtesy: Columbia University

Activation function of Perceptron

Let: $z = \boldsymbol{w}.\boldsymbol{x} + b$

Activation: $a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$

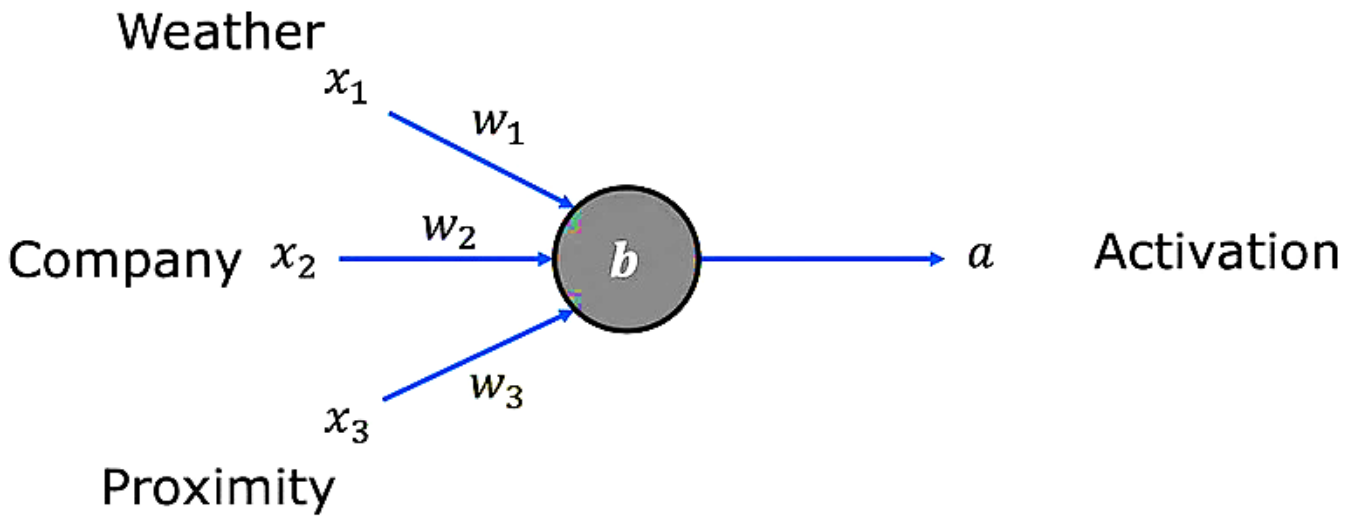


Activation Function for a Perceptron is a Step Function

Courtesy: Columbia University

Perceptron: Example

Will you go to the movies?



x_1: Is the weather good?

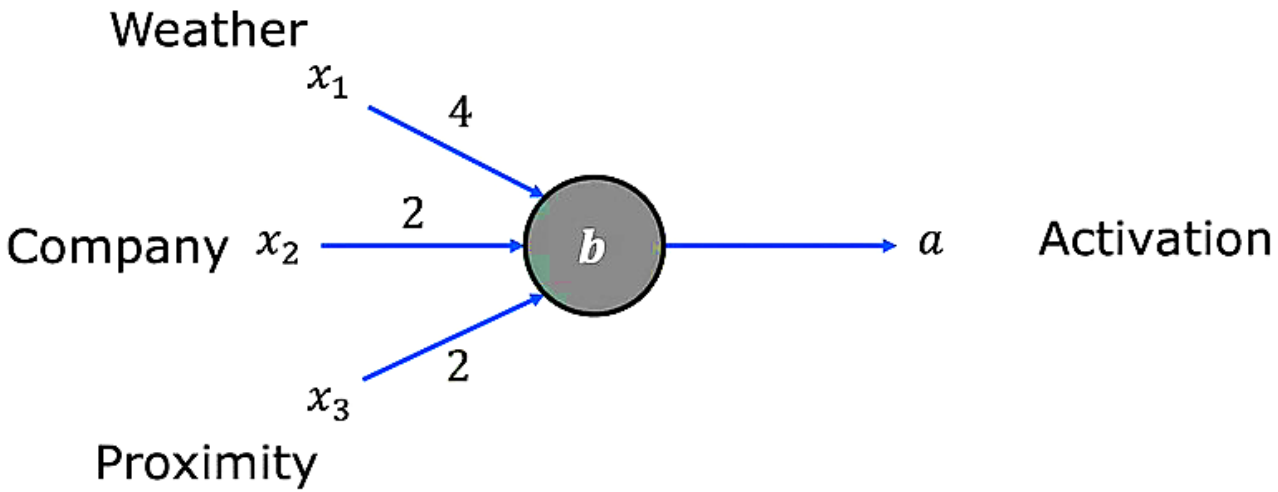
x_2: Do you have company?

x_3: Is the theater nearby?

Courtesy: Columbia University

Perceptron: Example

Will you go to the movies?



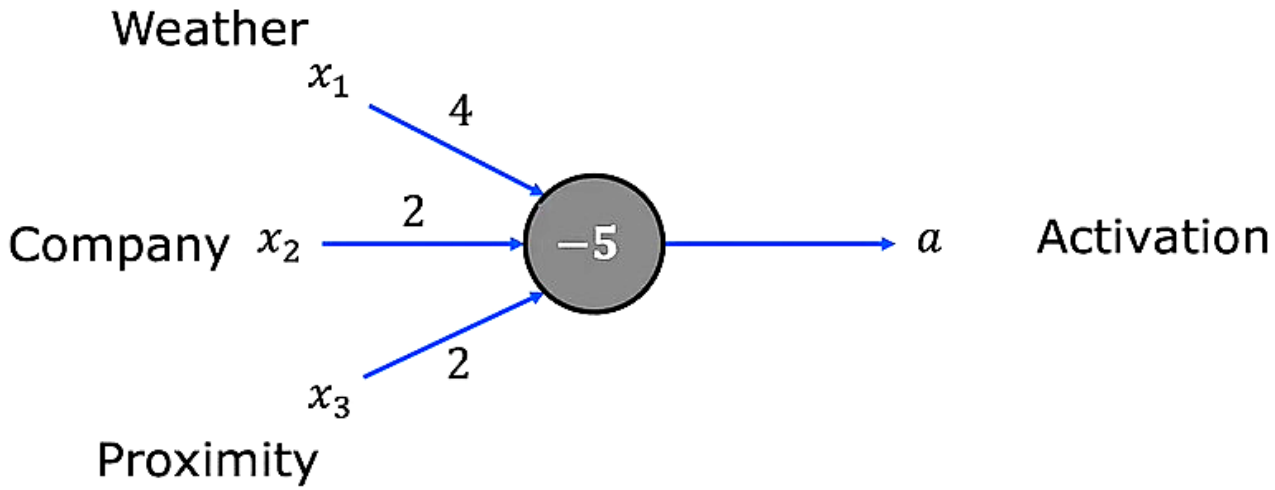
Assume that weather is the most important factor:

$$w_1 = 4, w_2 = 2, w_3 = 2$$

Courtesy: Columbia University

Perceptron: Example

Will you go to the movies?



Assume that weather is the most important factor:

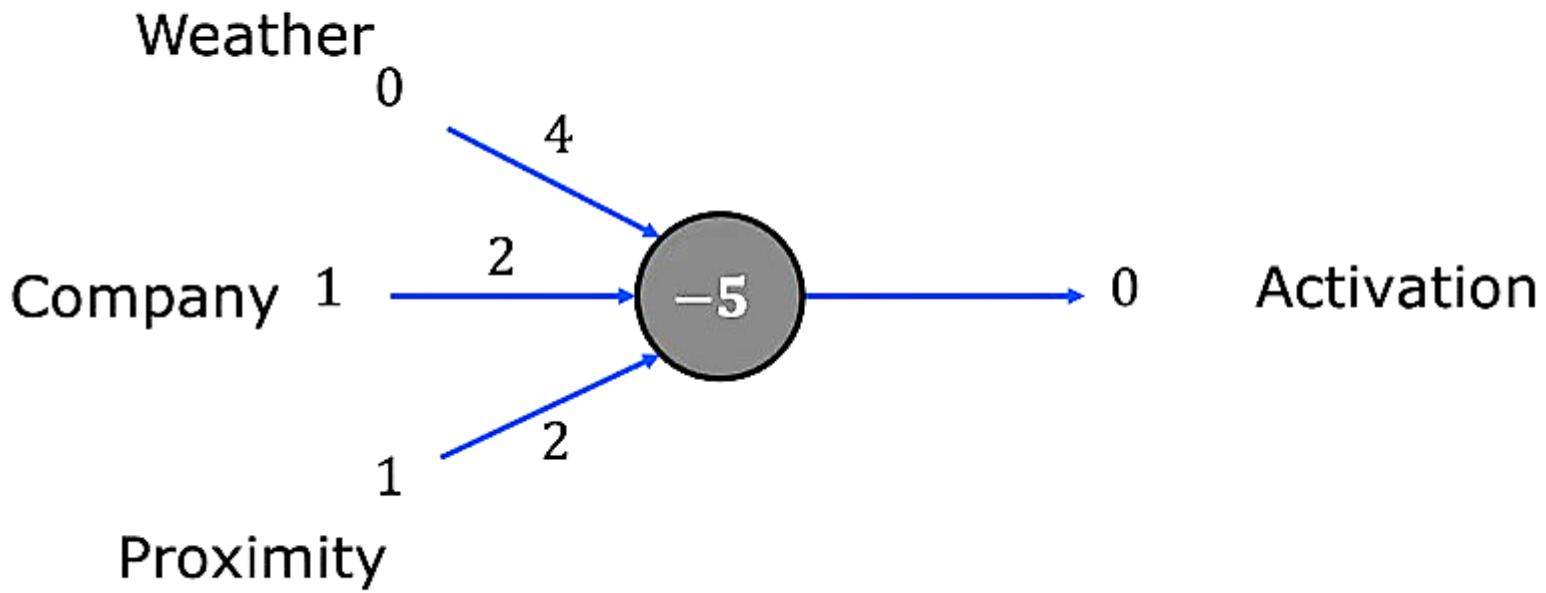
$$w_1 = 4, w_2 = 2, w_3 = 2$$

Go to the movies only if the weather is good and if one of the other factors is in favor: $b = -5$

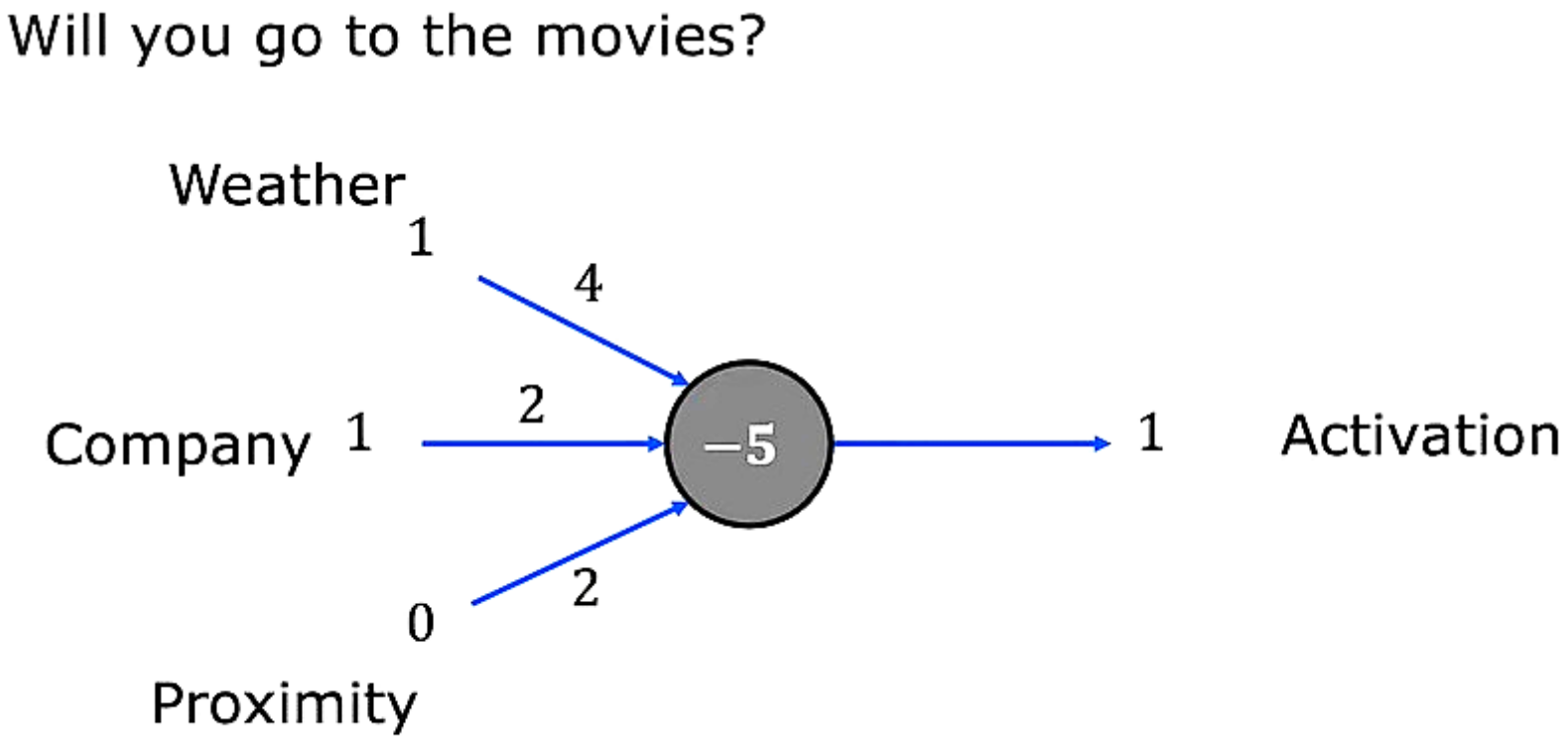
Courtesy: Columbia University

Perceptron: Example

Will you go to the movies? No

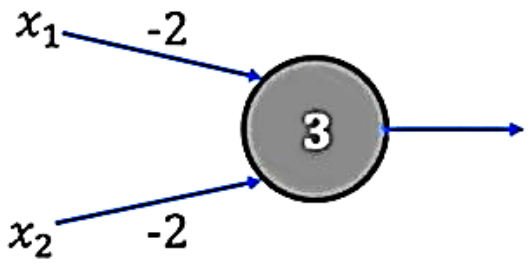


Perceptron: Example



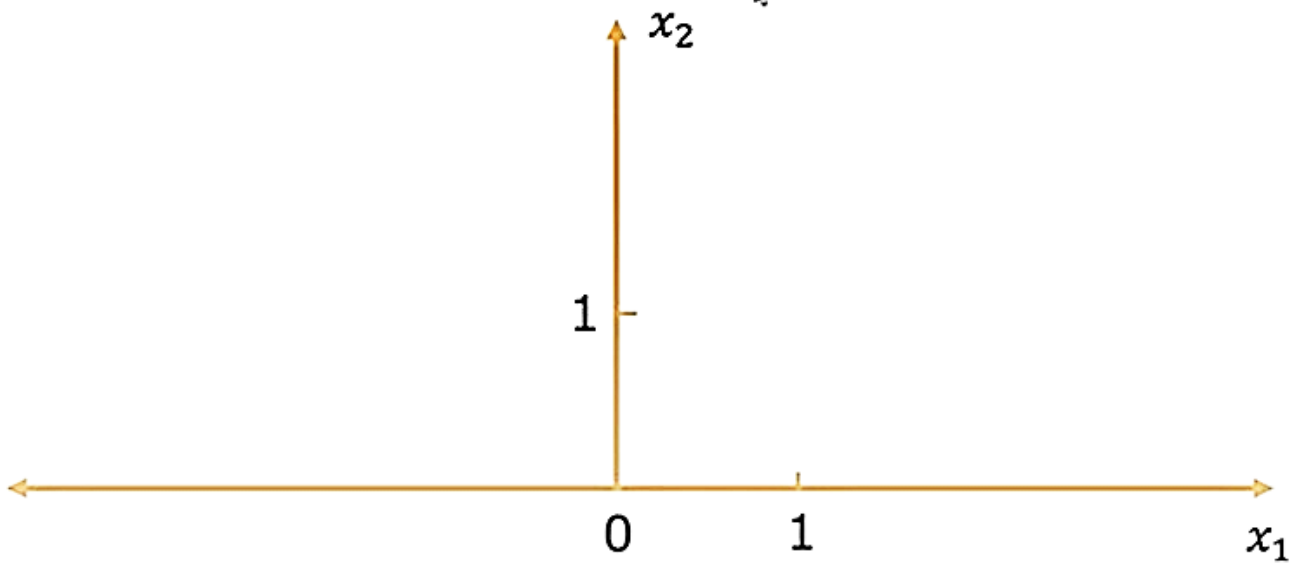
Courtesy: Columbia University

Perceptron as a Linear Classifier



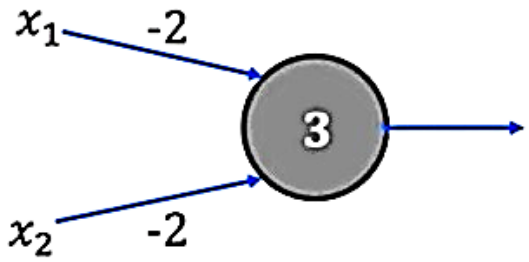
$$a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

$$z = -2x_1 - 2x_2 + 3$$



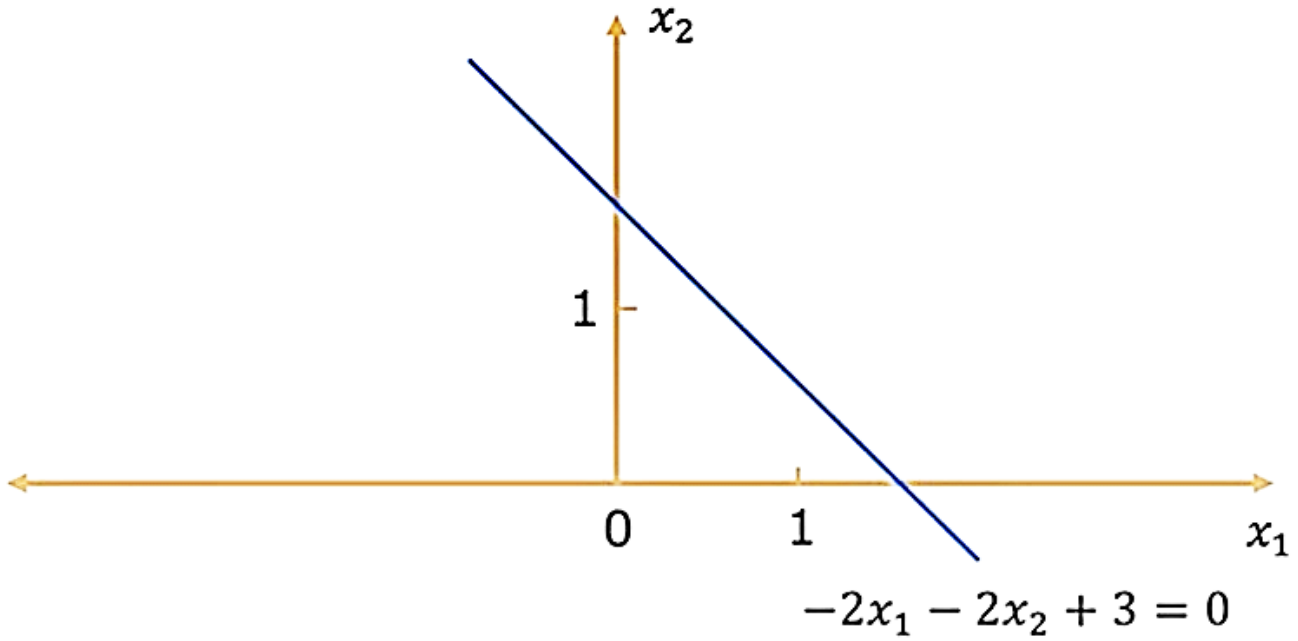
Courtesy: Columbia University

Perceptron as a Linear Classifier



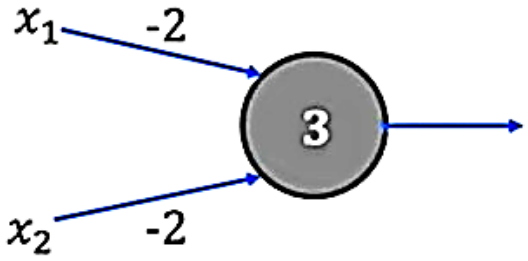
$$a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

$$z = -2x_1 - 2x_2 + 3$$



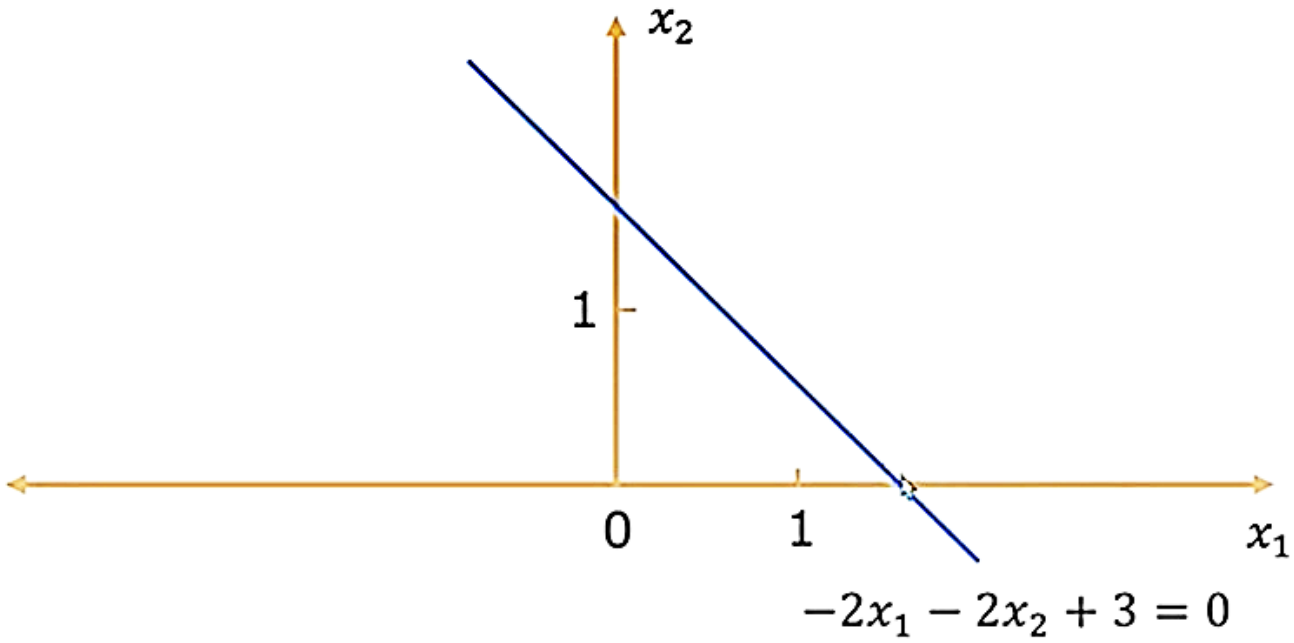
Courtesy: Columbia University

Perceptron as a Linear Classifier



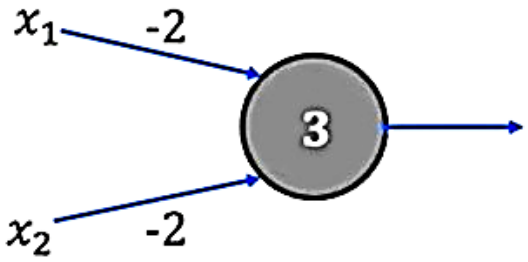
$$a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

$$z = -2x_1 - 2x_2 + 3$$



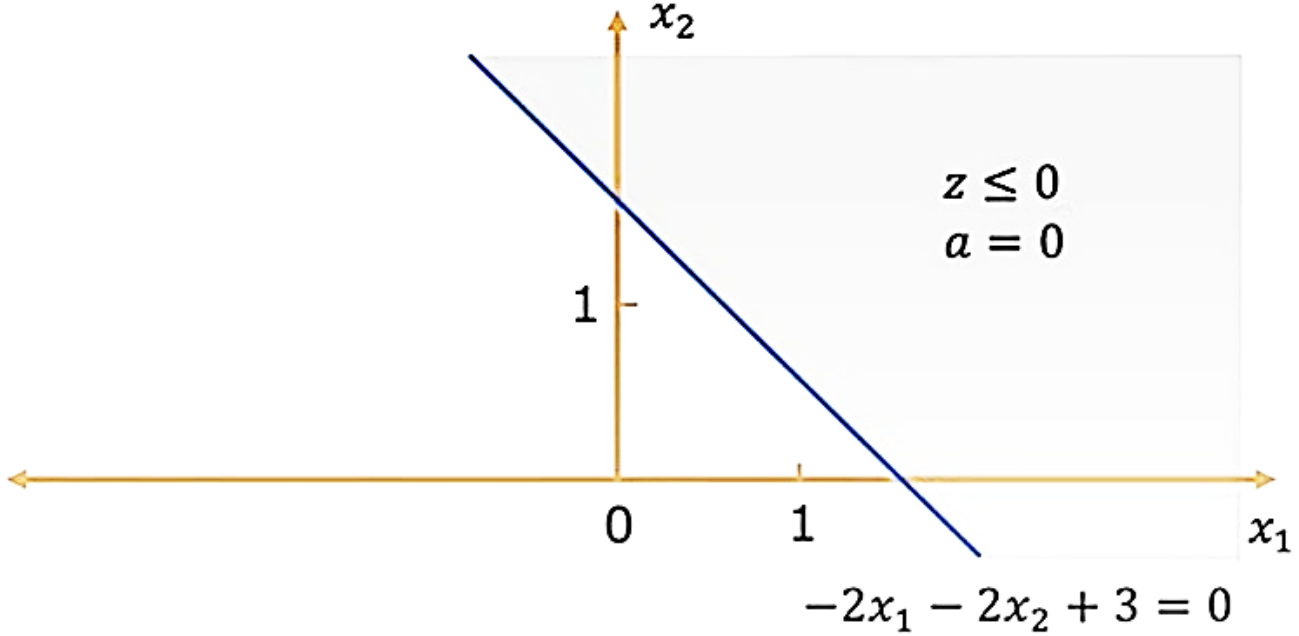
Courtesy: Columbia University

Perceptron as a Linear Classifier



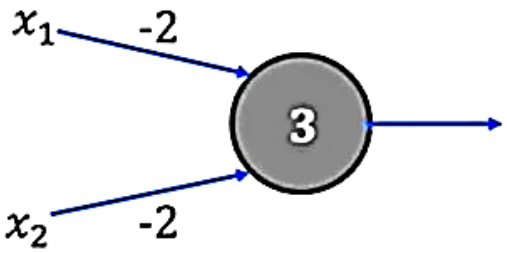
$$a = f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

$$z = -2x_1 - 2x_2 + 3$$



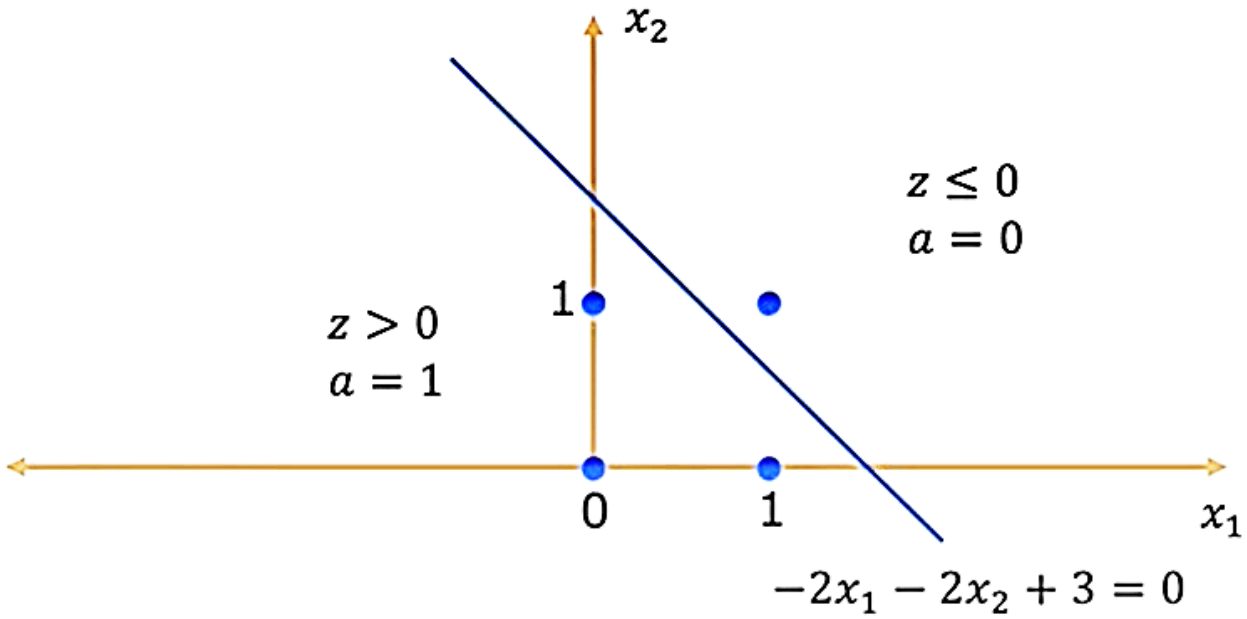
Courtesy: Columbia University

Perceptron as a Linear Classifier



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

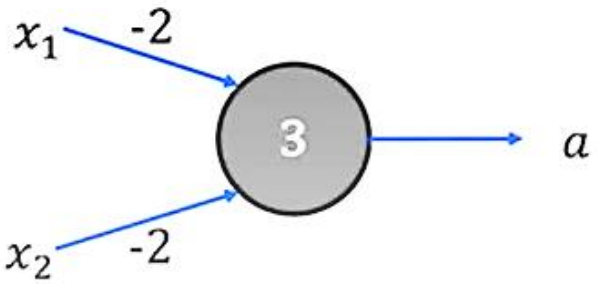
$$z = -2x_1 - 2x_2 + 3$$



Courtesy: Columbia University

Perceptron as a NAND Gate

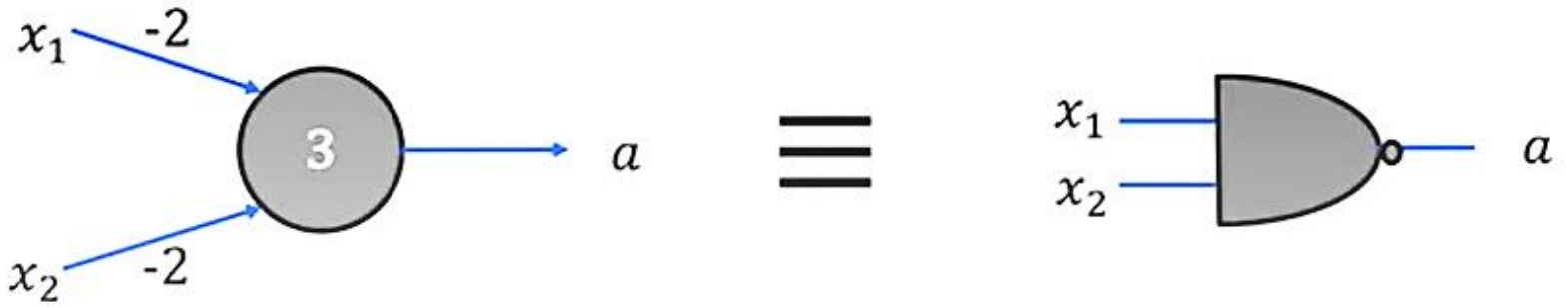
x_1	x_2	a
0	0	1
0	1	1
1	0	1
1	1	0



Courtesy: Columbia University

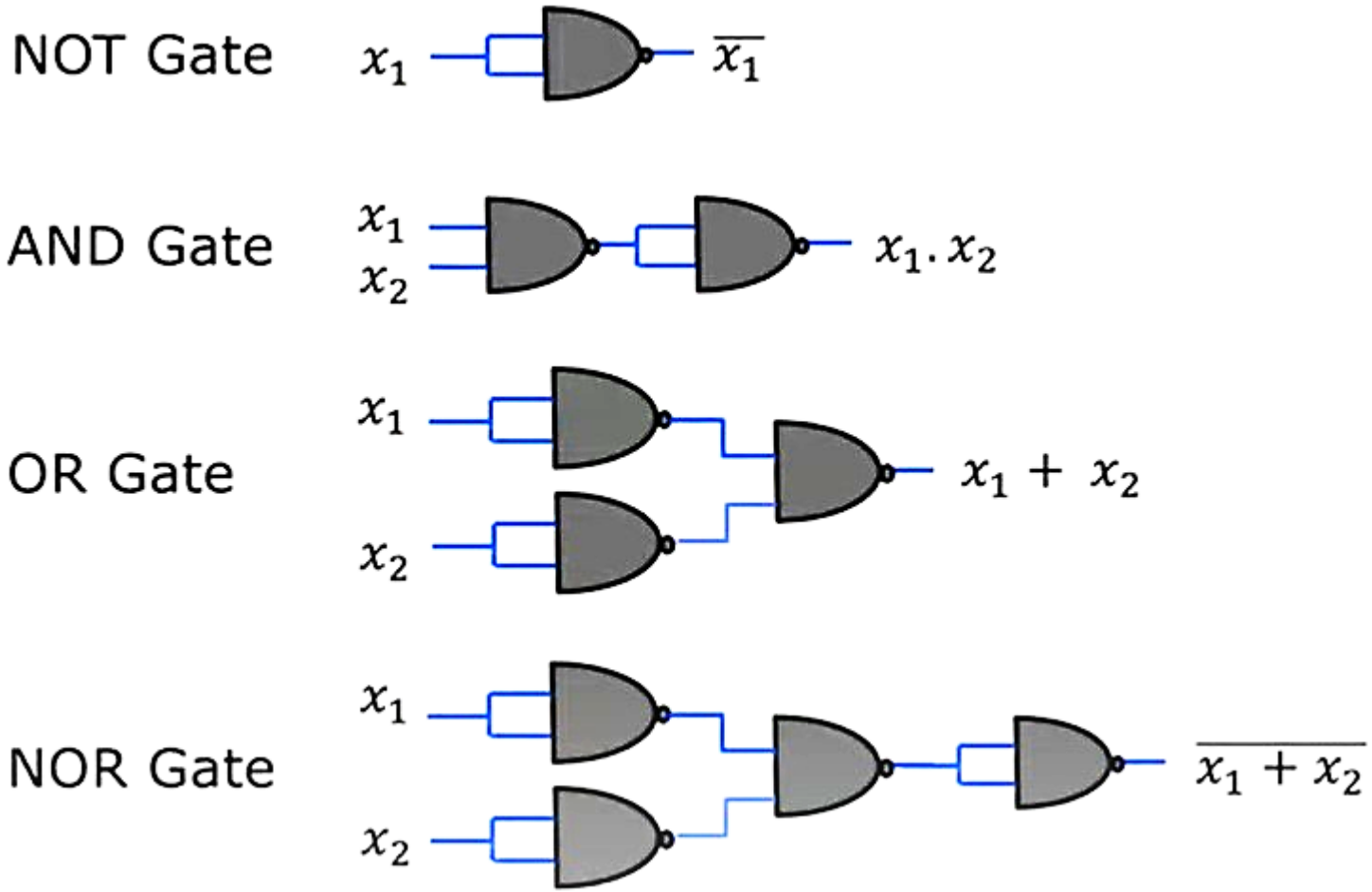
Perceptron as a NAND Gate

x_1	x_2	a
0	0	1
0	1	1
1	0	1
1	1	0



Courtesy: Columbia University

NAND: Universal Logic Gate



The Perceptrons are Universal for Computation

Courtesy: Columbia University

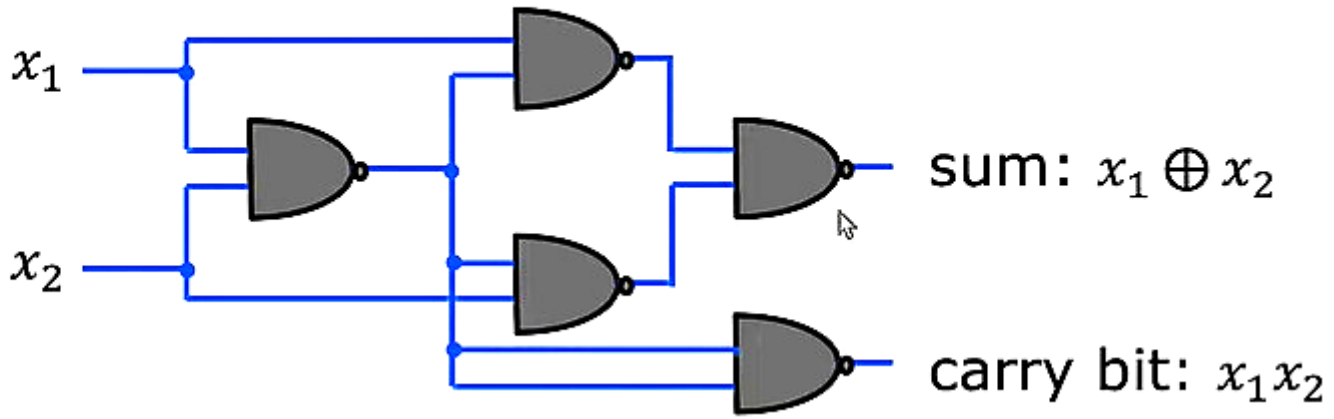
Perceptrons: Computational Universality

Any Digital Logic Circuit can be implemented using Perceptron

Courtesy: Columbia University

Perceptrons: 1-bit Adder

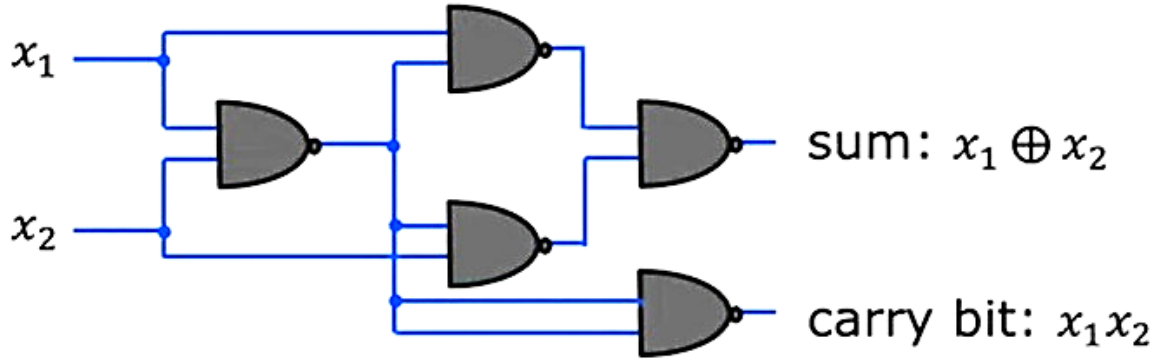
Digital Logic Circuit



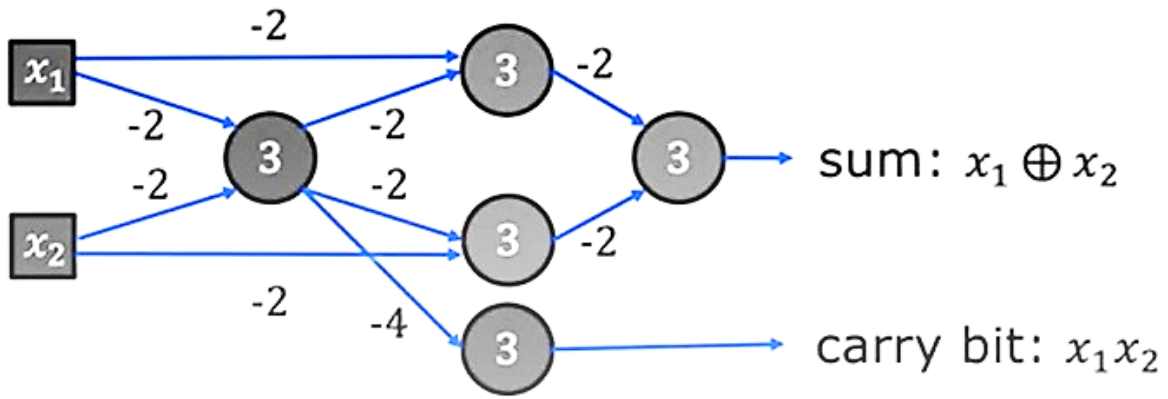
Courtesy: Columbia University

Perceptrons: 1-bit Adder

Digital Logic Circuit

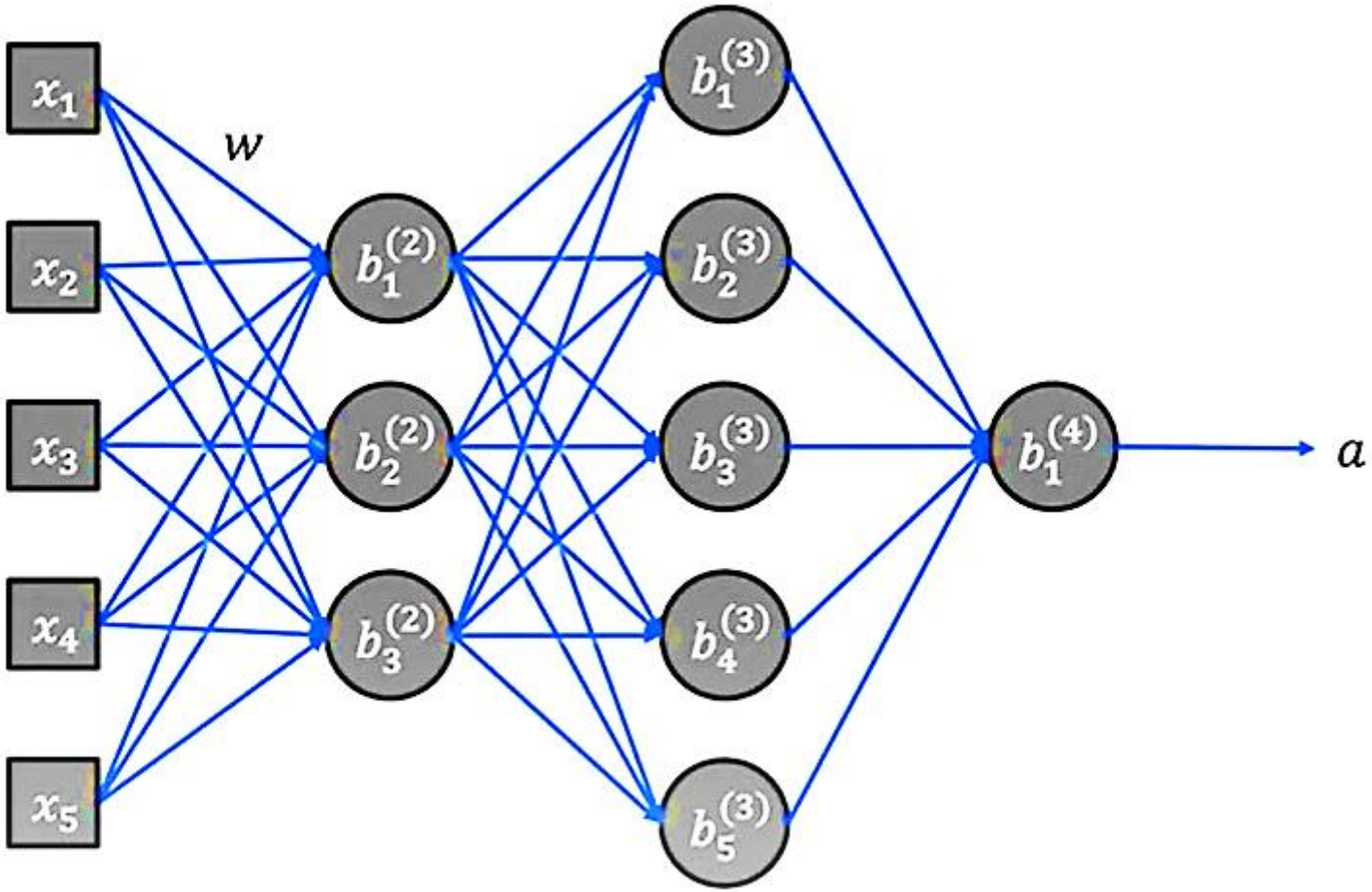


Equivalent Perceptron Network



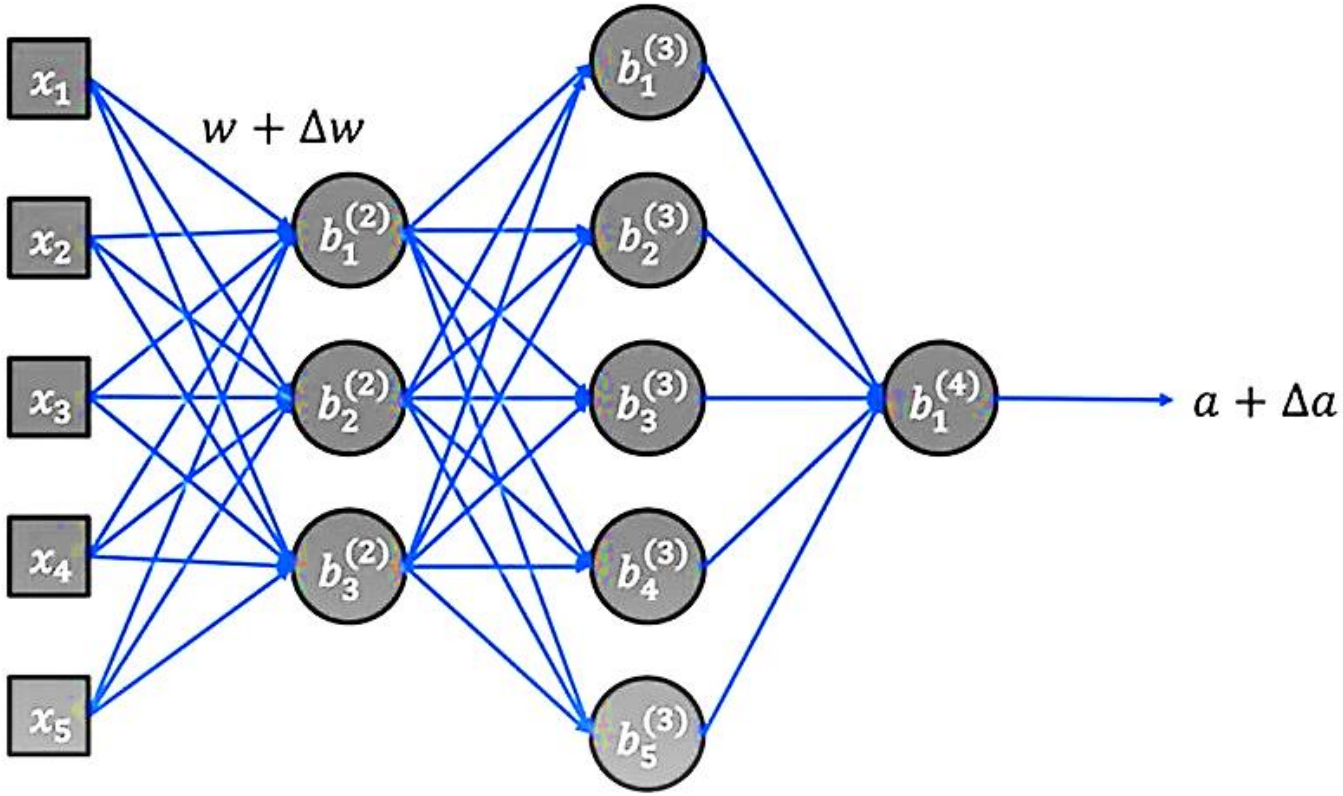
Courtesy: Columbia University

Adjusting a Perceptron Network



Courtesy: Columbia University

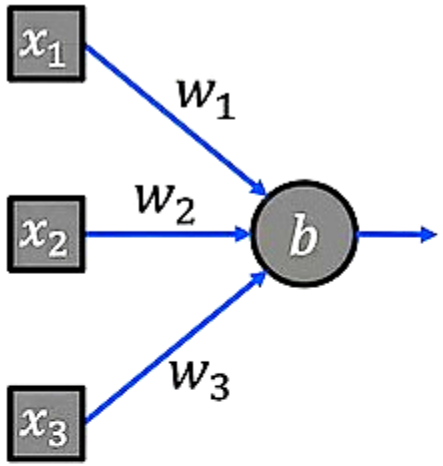
Adjusting a Perceptron Network



We want a small change in Weight (or Bias) to lead to a small change in the Activation towards the Desired Activation

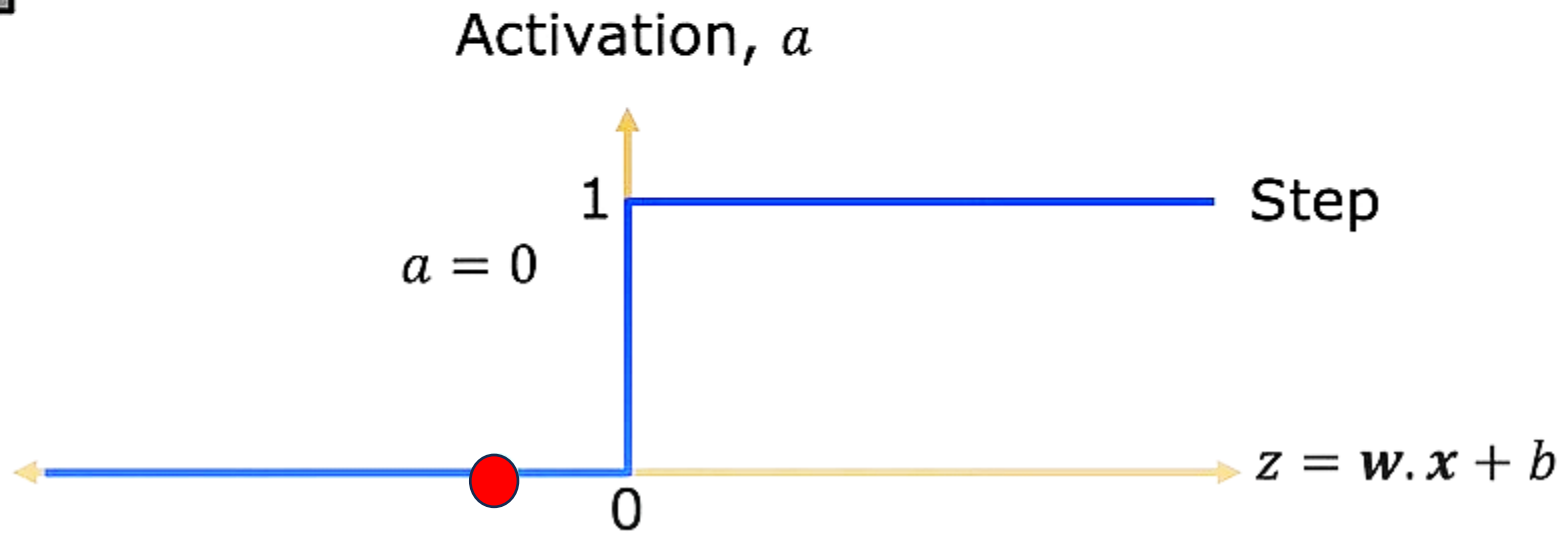
Courtesy: Columbia University

Adjusting Single Perceptron



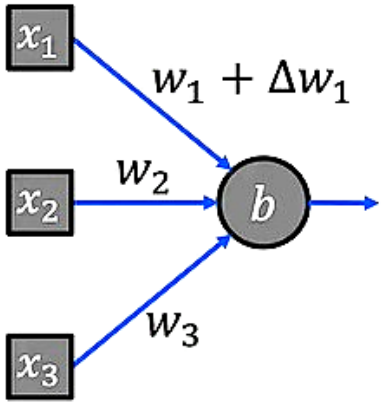
$$a = f(z)$$

$$\text{where, } f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$



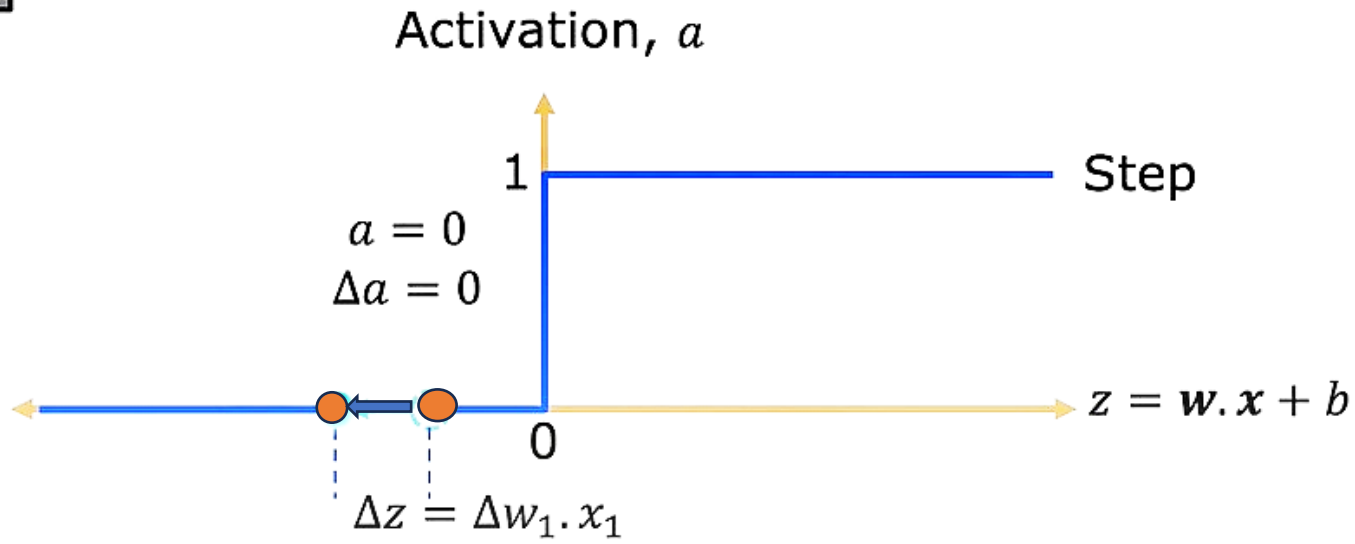
Courtesy: Columbia University

Adjusting Single Perceptron



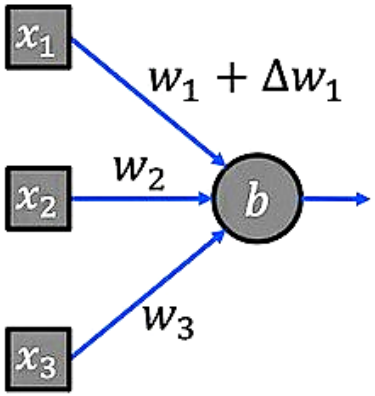
$$a + \Delta a = f(z + \Delta z)$$

where, $f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$



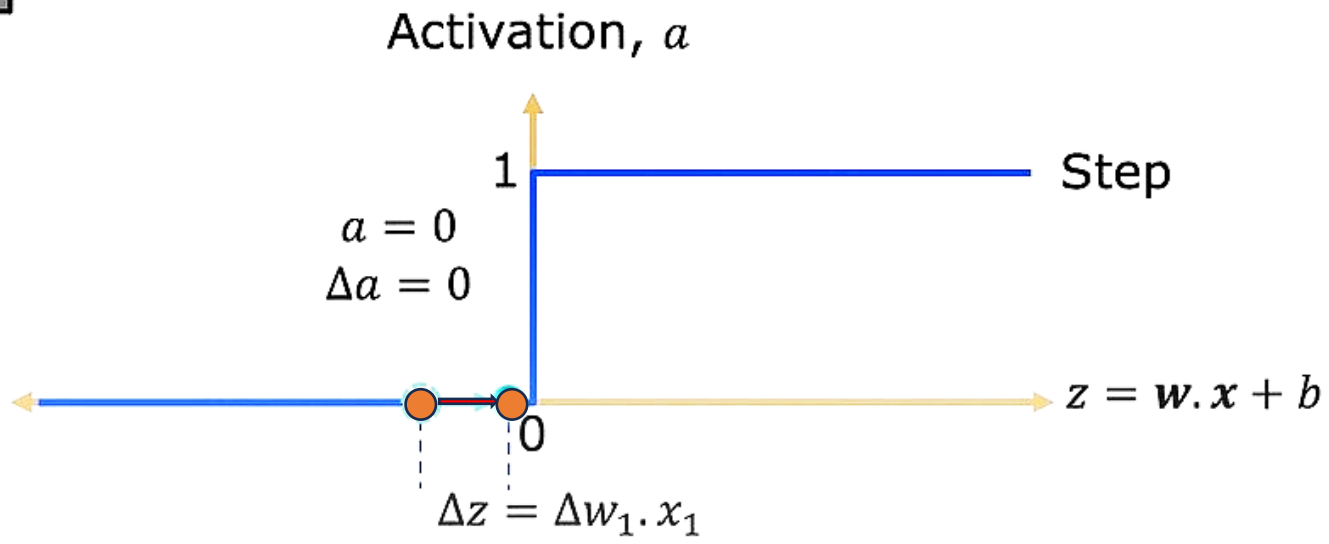
Courtesy: Columbia University

Adjusting Single Perceptron



$$a + \Delta a = f(z + \Delta z)$$

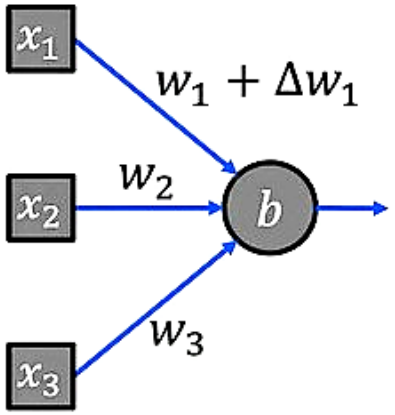
where, $f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$



Courtesy: Columbia University

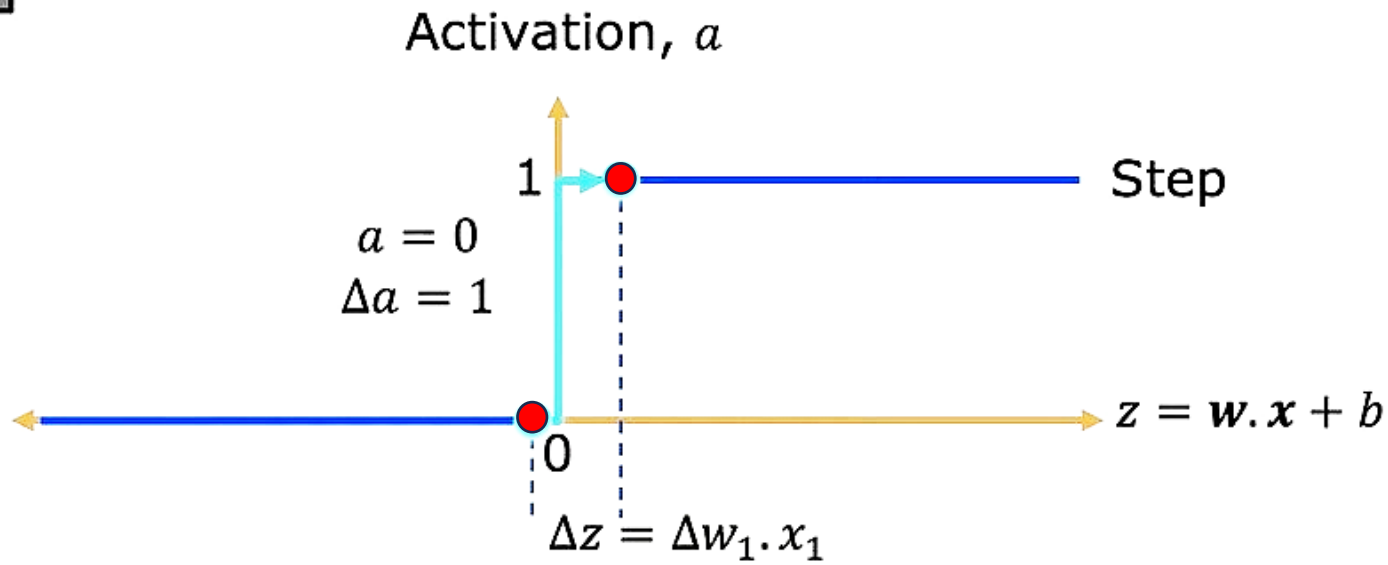
Change the weight in Either Direction Gives the same activation

Adjusting Single Perceptron



$$a + \Delta a = f(z + \Delta z)$$

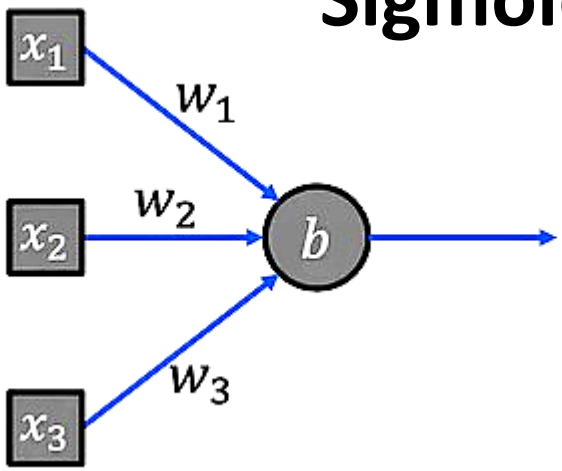
$$\text{where, } f(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$



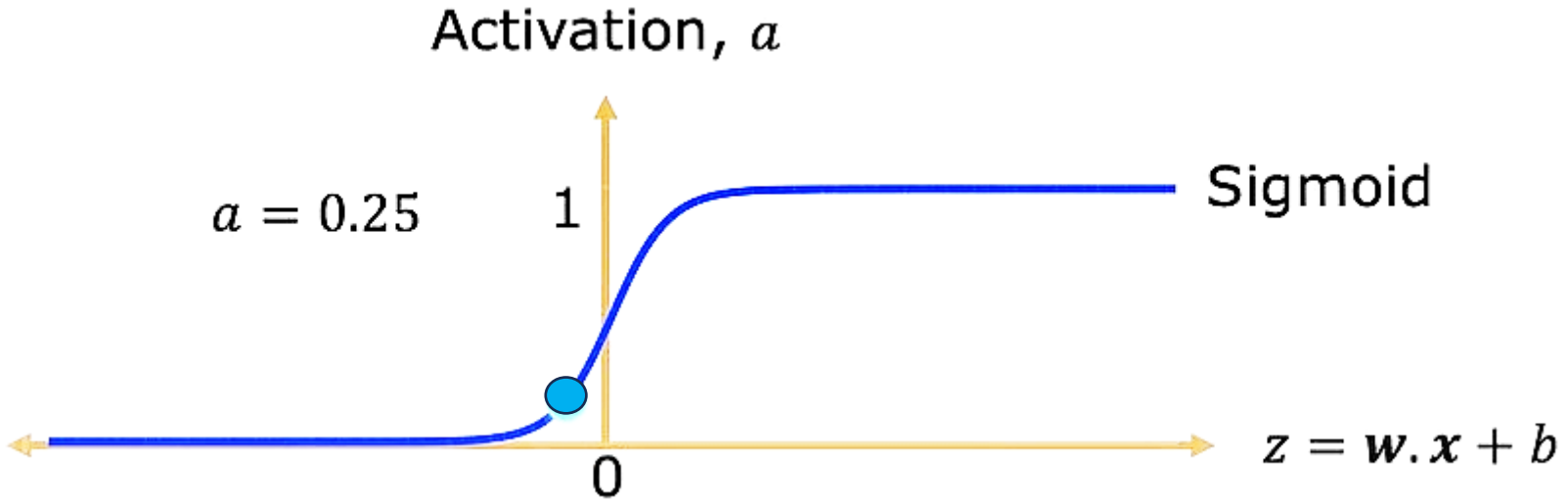
Small change in Weight can also completely flip the Activation

Courtesy: Columbia University

Sigmoid Neuron

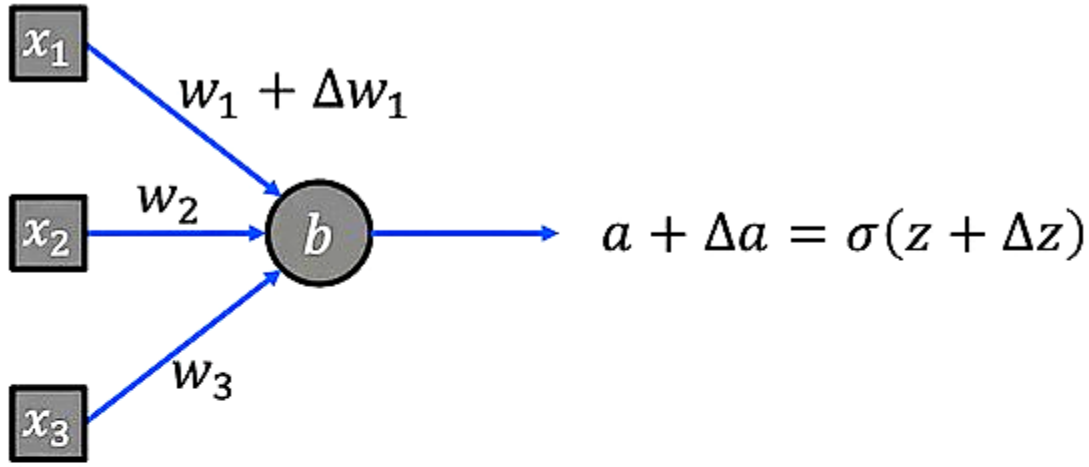


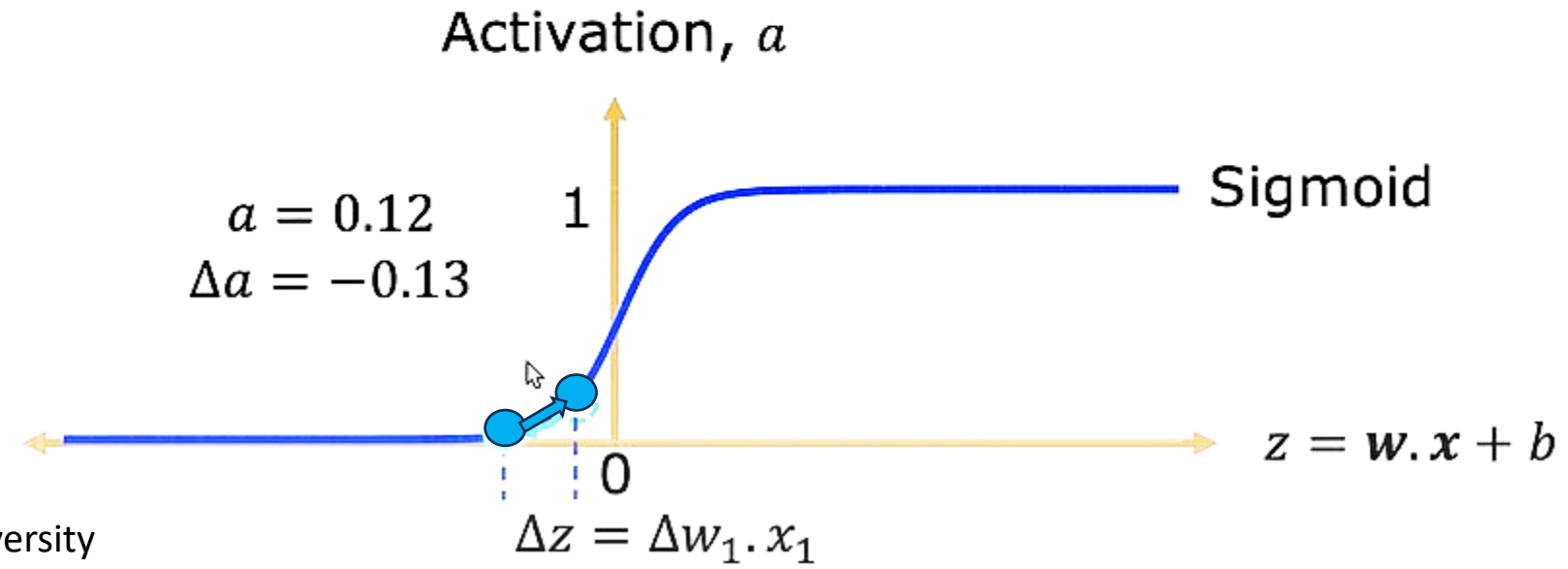
$$a = \sigma(z) = \frac{1}{1 + e^{-z}} \text{ (Sigmoid)}$$



Courtesy: Columbia University

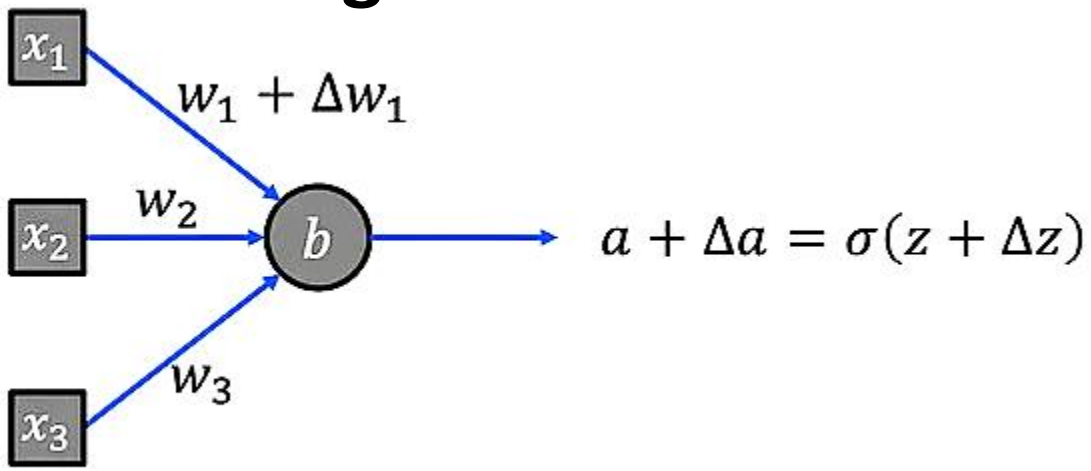
Sigmoid Neuron

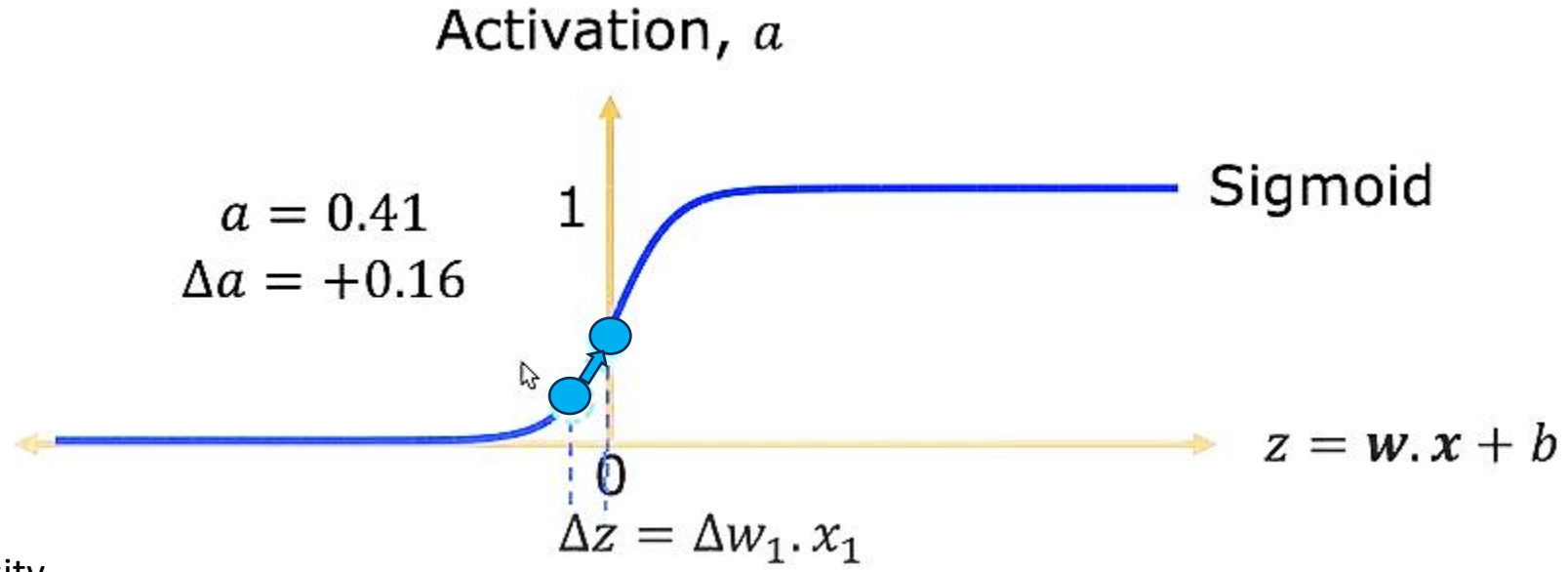




Courtesy: Columbia University

Sigmoid Neuron

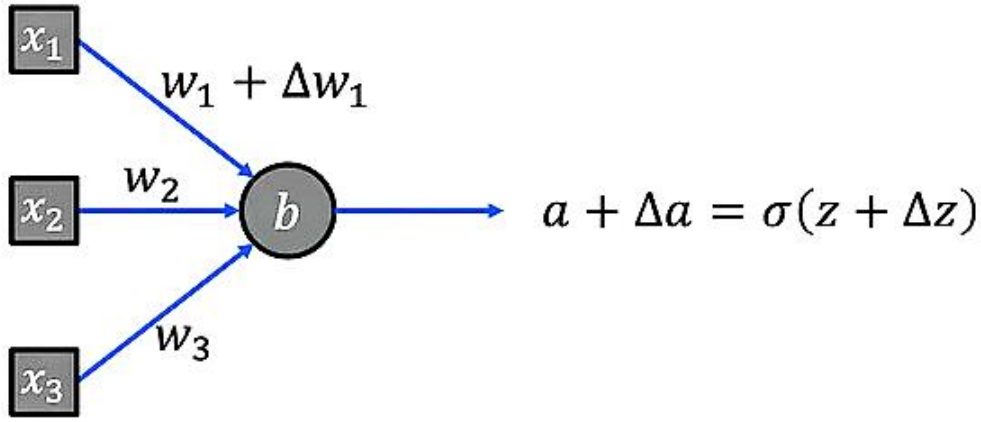


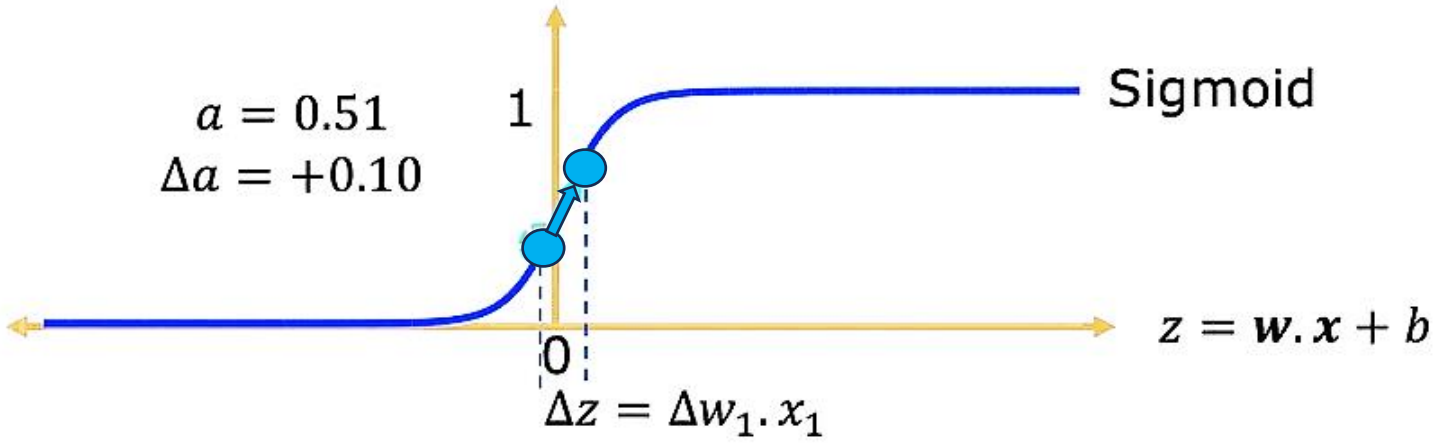


Courtesy: Columbia University

Sigmoid Neuron

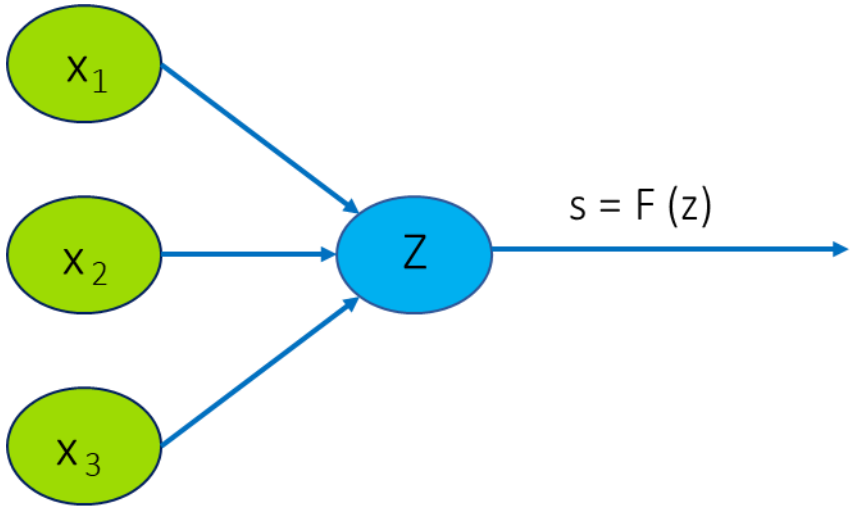
Courtesy: Columbia University





Output **transitions smoothly** with changes in Weights and Biases

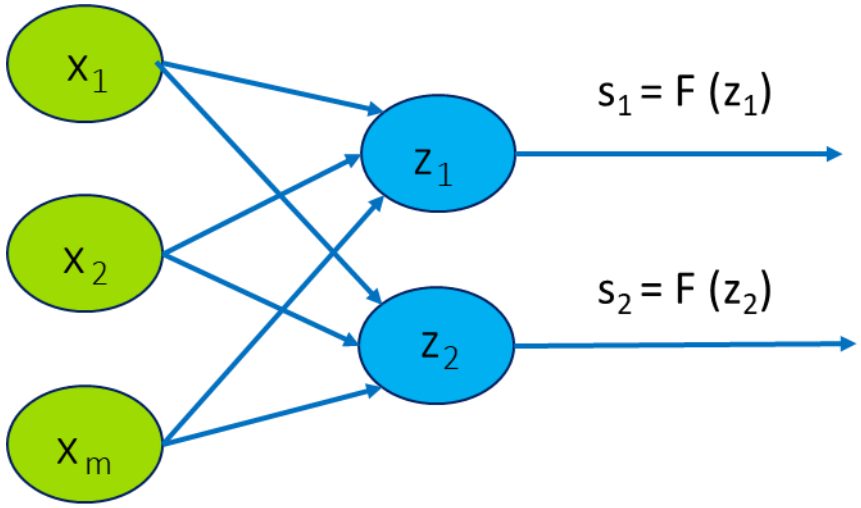
The Perceptron: Simplified



- Here z be the output of the dot product that is elementwise multiplication of input with respective weights and that's get fed into activation function.
- S is the output of Z after applying the activation function.

$$Z = w_0 + \sum_{j=1}^{m} x_j w_j$$

Multi Output Perceptron

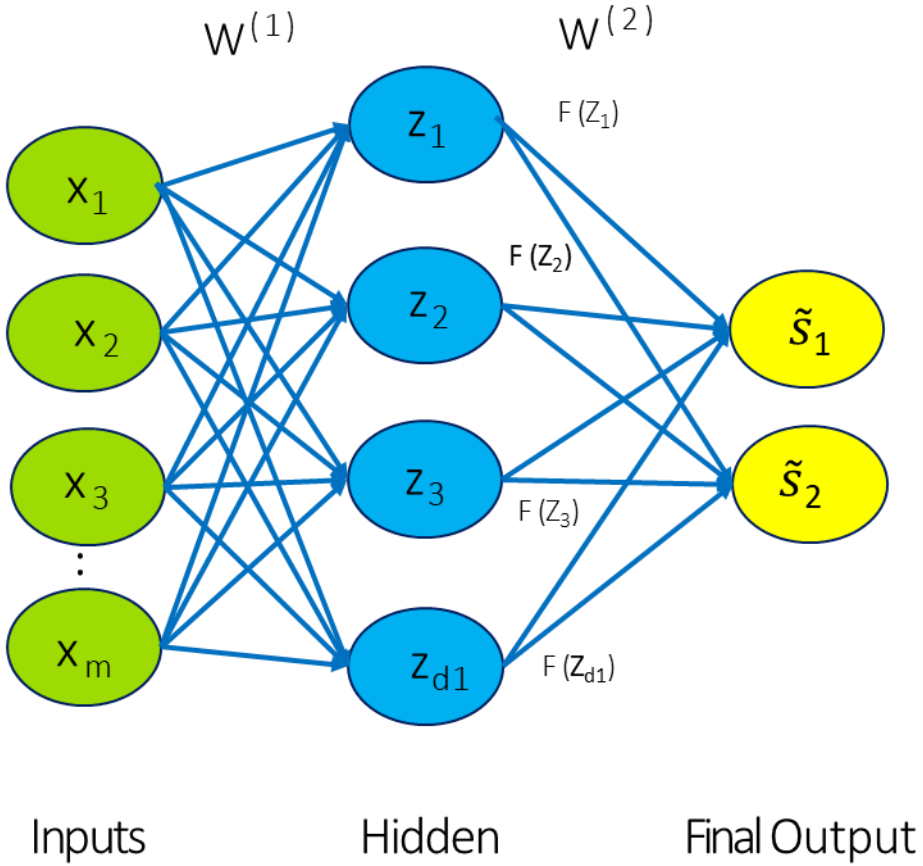



```
import tensorflow as tf

layer = tf.keras.layers.Dense(
    units=2)
```

- Multi output perceptron is the combination of single output perceptron having two output. Here S1 and S2 are the output of two perceptron connecting to the previous layer with different set of weights.
- Because all inputs are densely connected to all the outputs, these types of layers are often called ***Dense*** layers.

$$z_i = w_{o,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

Single Layer Neural Network



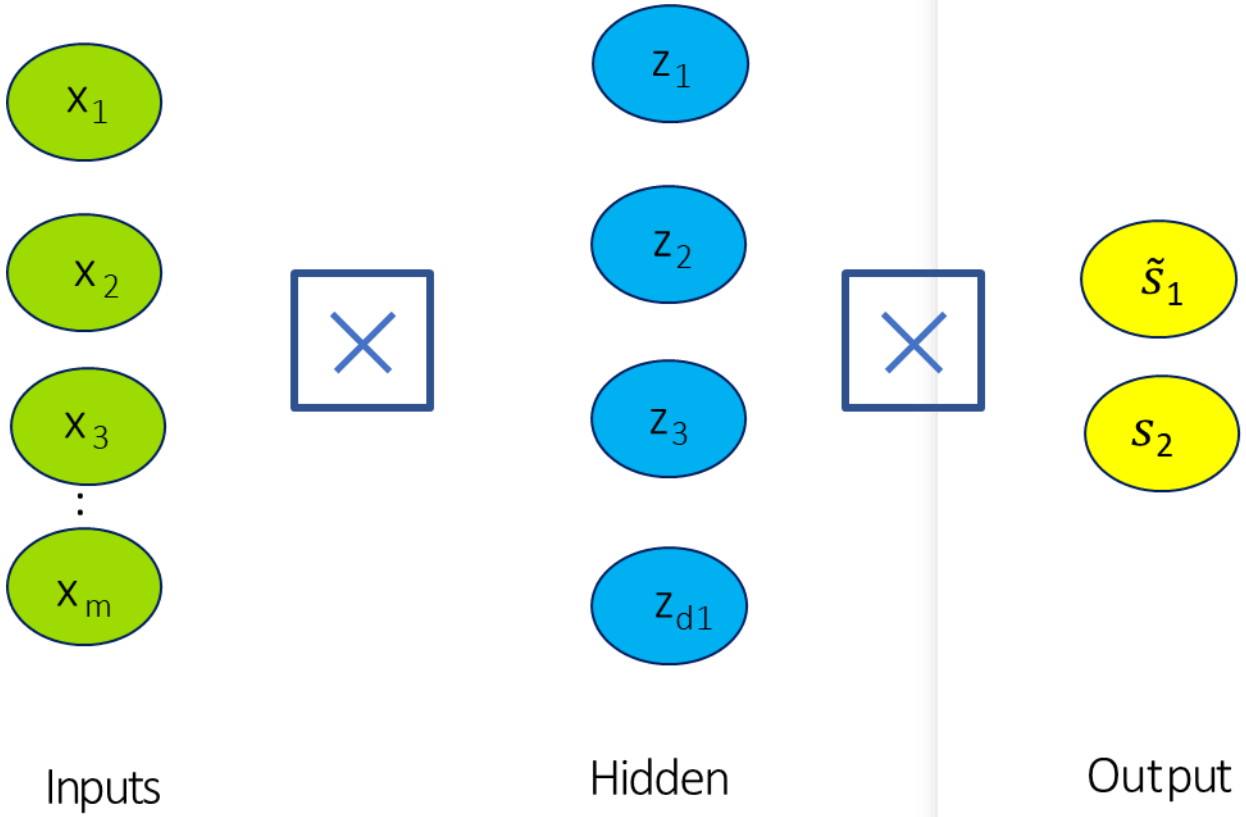
- Here is the **hidden layer** feeding the output layer. State of hidden layer is not directly insert or observed.
- Two transformation occurred in layers that are W^1 and W^2 , which has their own weight matrices

$$Z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}$$

$$\tilde{s} = F(w_{0,i}^{(2)} + \sum_{j=1}^{d} z_j w_{j,i}^{(2)})$$

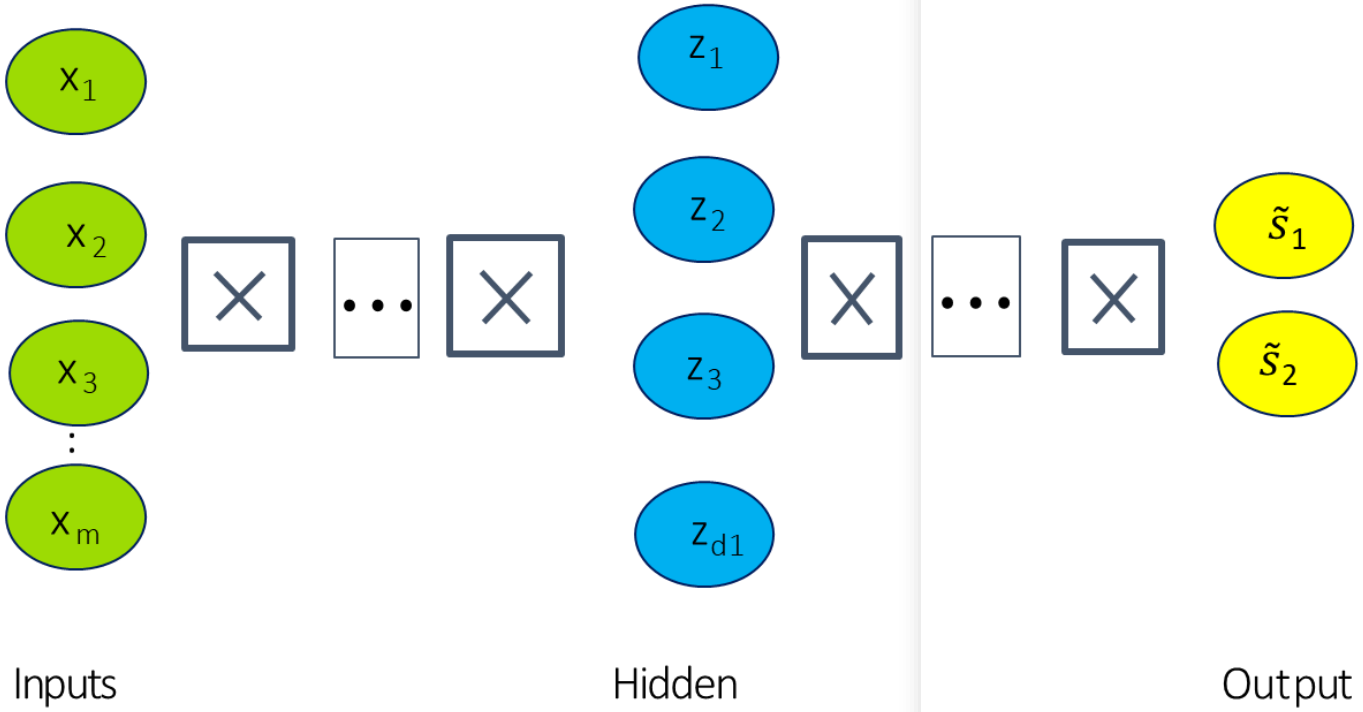
Since we have a transformation between the inputs and the hidden layer and the hidden layer and the output layer, each of these transformation will have their own weight matrices : W1 and W2 which corresponds to the first layer and the second layer.

Multi Output Perceptron




```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

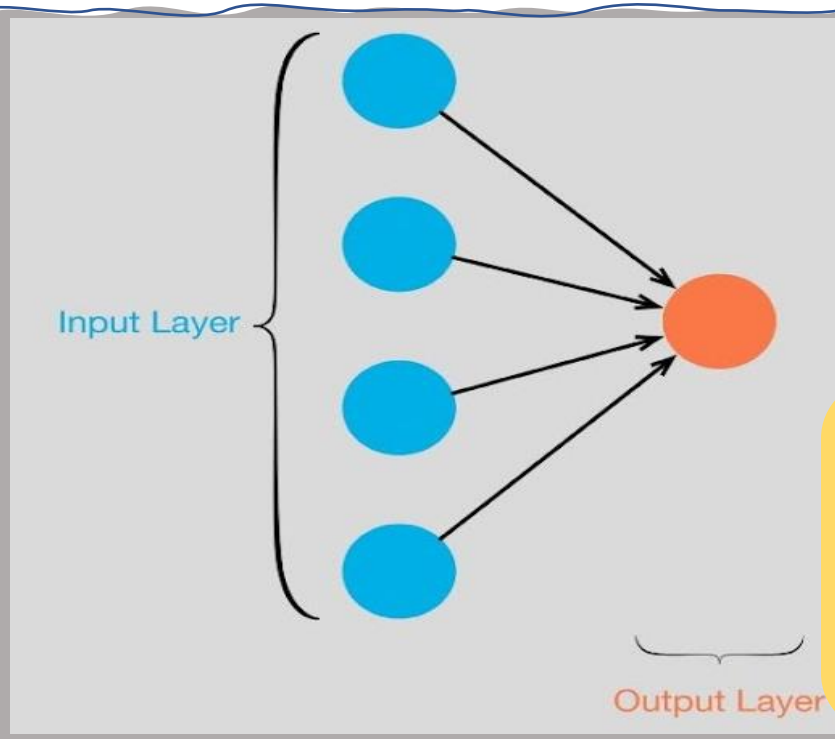
Deep Neural Network




```python
import tensorflow as tf

model = tf.keras.Sequential([
  tf.keras.layers.Dense(n_1),
  tf.keras.layers.Dense(n_2),
  ⋮
  tf.keras.layers.Dense(2)
])
```

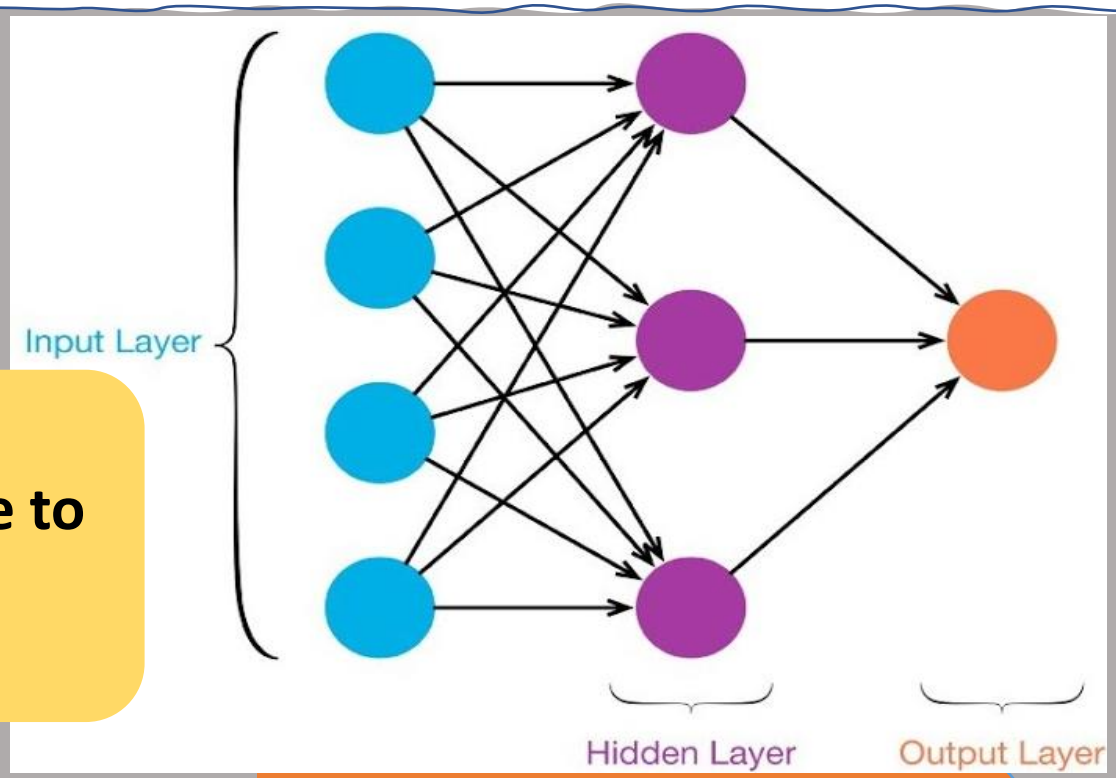
$$z_{k,i} = W_{0,i}^{k} + \sum_{d=1}^{d_{k-1}} F\left(z_{k-1,j}\right) W_{j,i}^{k}$$

Summary



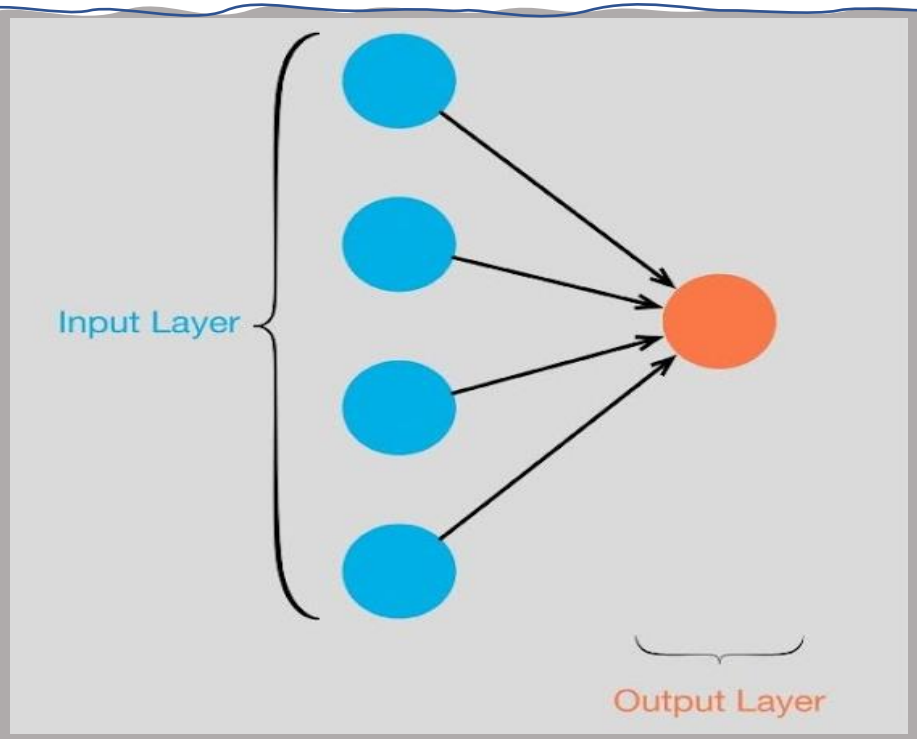
Single Layer

Which one to choose



Multi-Layer

Summary

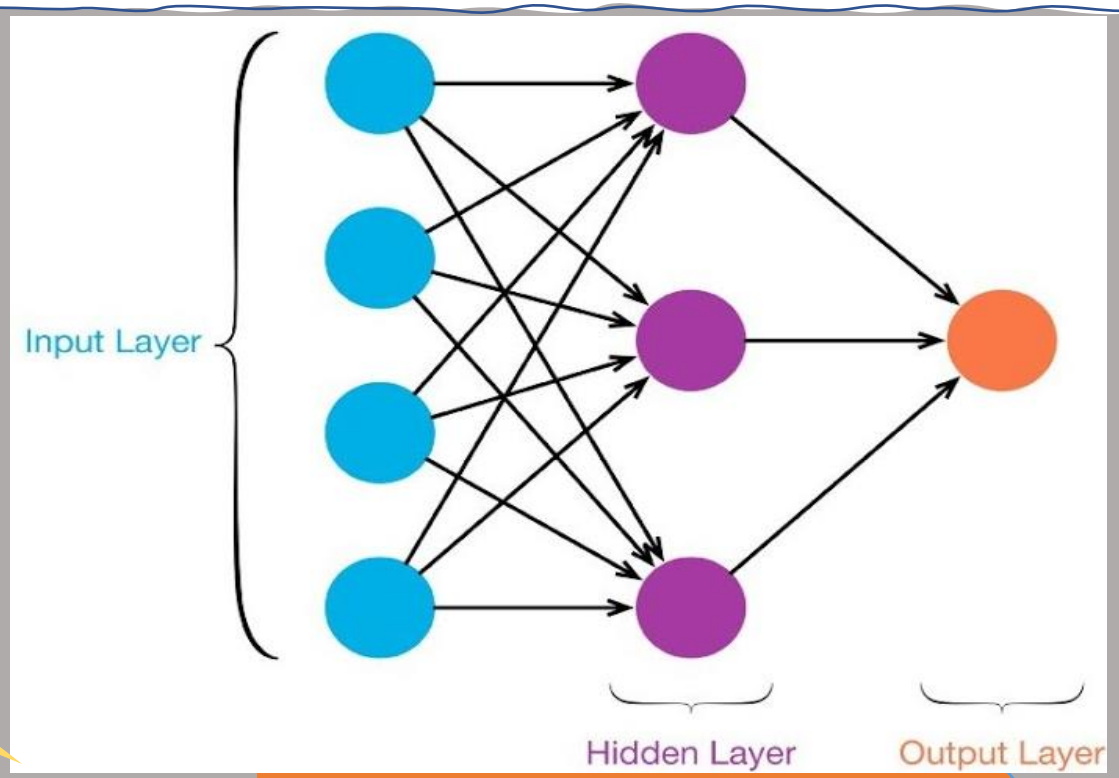


Single Layer

It is more suitable to simple and linear problems

Summary

Multi-layer perceptron is more suitable for complex and non-linear problems.



Multi-Layer

Applying Neural Networks