Introduction to ANN, MLP, Learning, and Backpropagation

Motivation

- Origins: Algorithms that try to mimic the brain.

Question: What is this?

Human Brain and Biological Neurons

- Human brain contains billion of neurons (~10 billion)
- Each neuron is a cell that uses biochemical reactions to **receive**, **process** and **transmit** information
- Neurons are connected together through ***synapses*** (~10K)

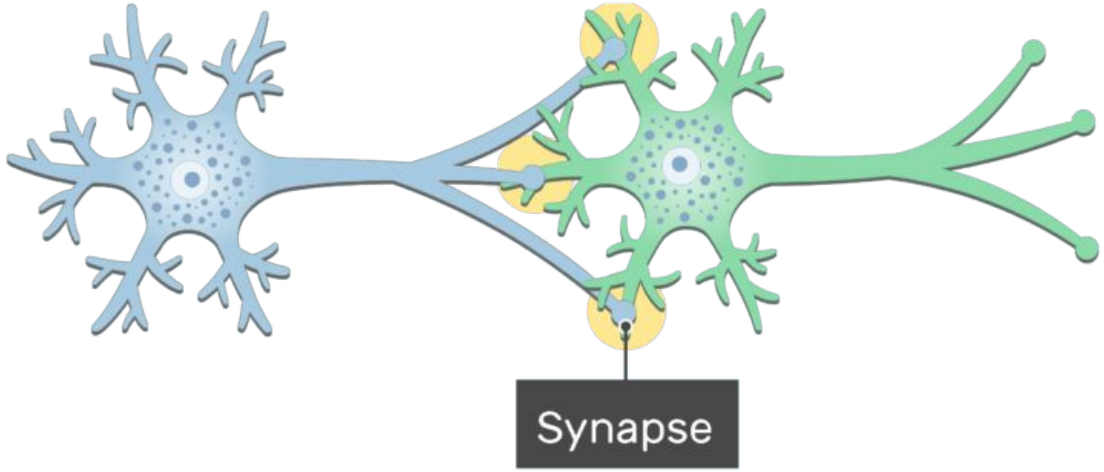
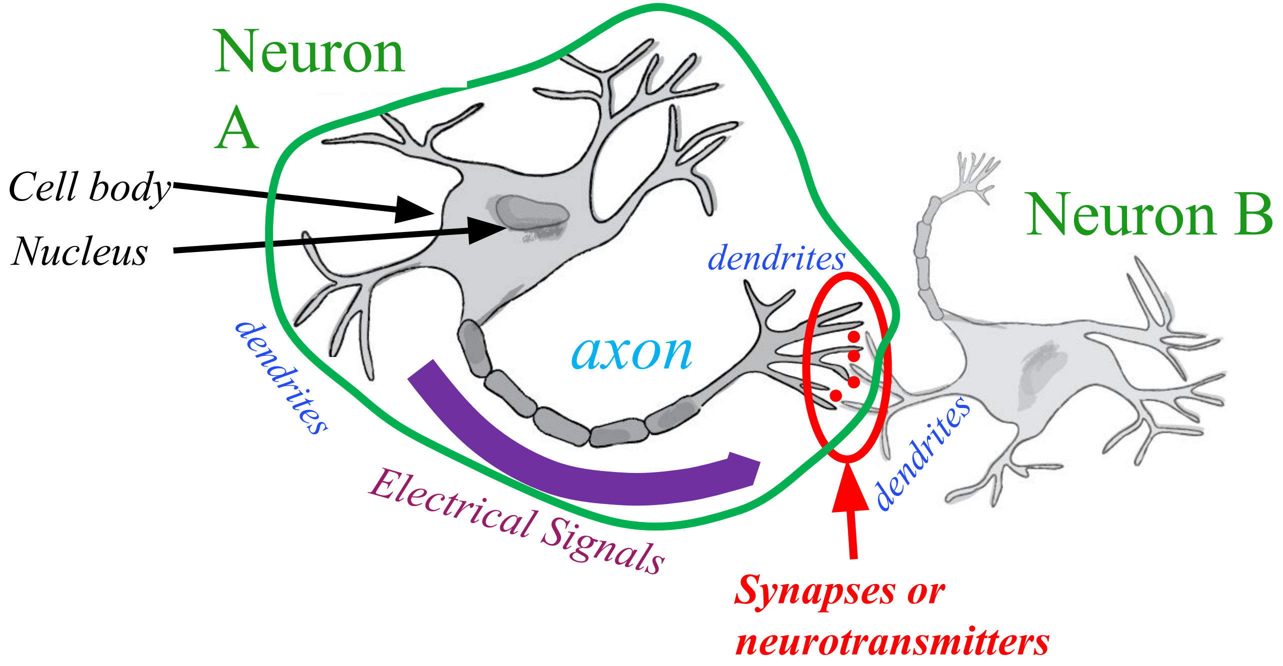


Image source: https://beautifulnow.is/discover/wellness/new-brain-flows-are-beautiful-now

Image source: https://www.getbodysmart.com/nervous-system/neuron-synapse-structure

Human Brain and Biological Neurons (cont'd)

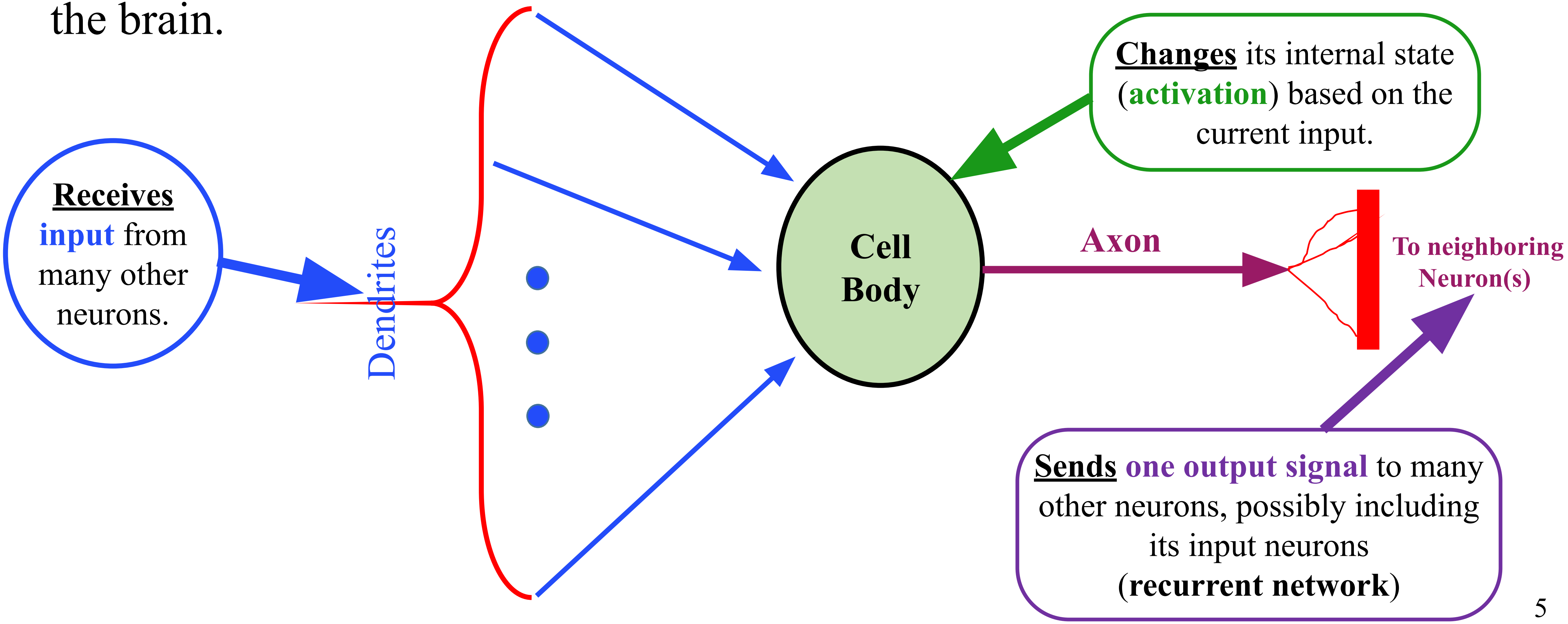
- A neuron accepts (**and combines**) inputs through ***dendrites*** from other neurons
- If a given neuron *combined* input above a **threshold**, the neuron discharges a spike (**electrical pulse**) that travels from the body, down the **axon**, to the next **neuron(s)**
- The strength of the signal that reaches the next neuron depends on factors such as the amount of neurotransmitter (***synapses***) available



https://natureofcode.com/book/chapter-10-neural-networks/

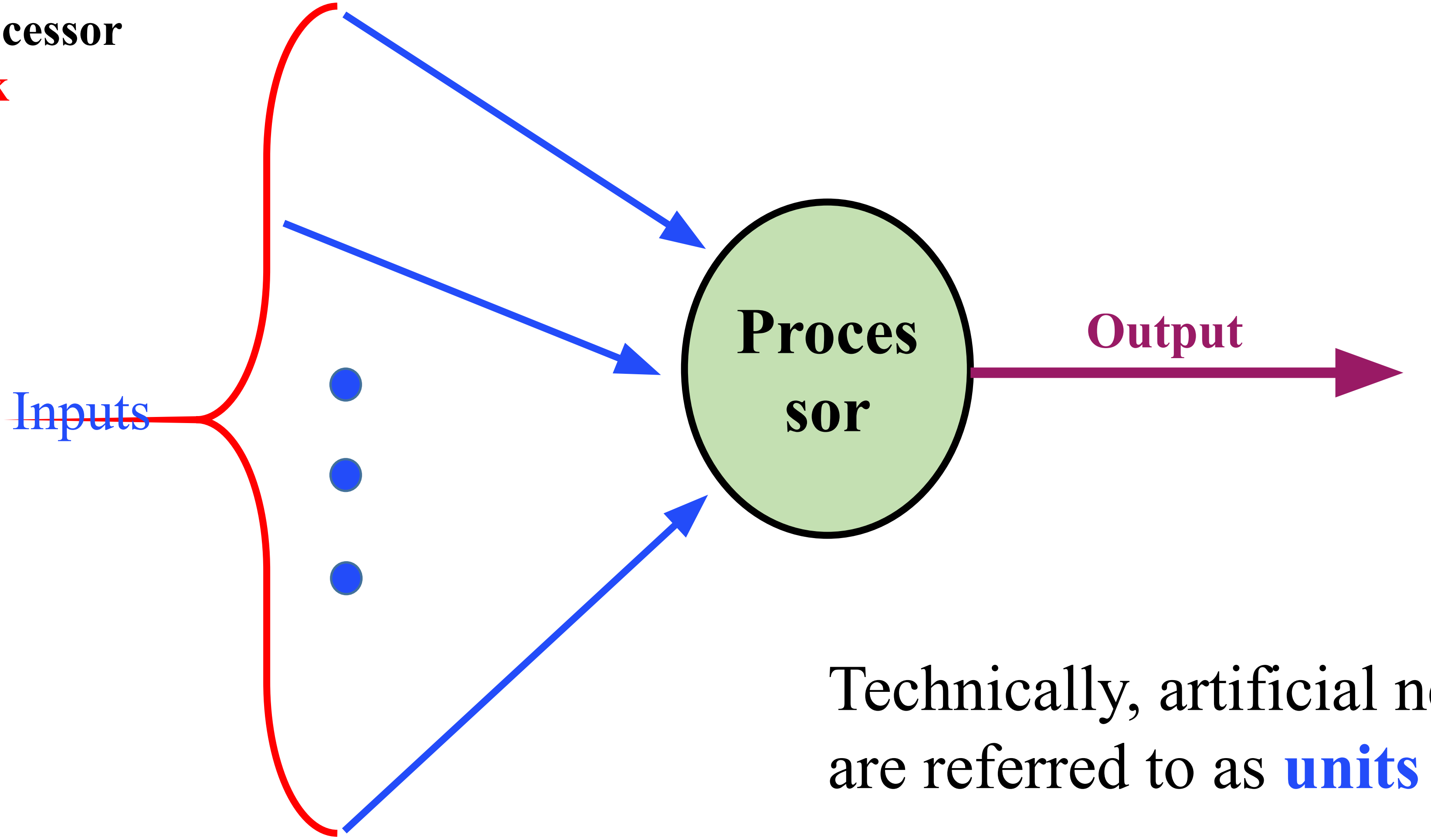
Modeling of a Biological Neuron

- A mathematical model of the neuron (called the perceptron) has been introduced in an effort to mimic our understanding of the functioning of the brain.



Artificial Neuron

- An artificial neuron is an imitation of a human neuron
  - **Dendrites: Input**
  - **Cell body: Processor**
  - **Synaptic: Link**
  - **Axon: Output**

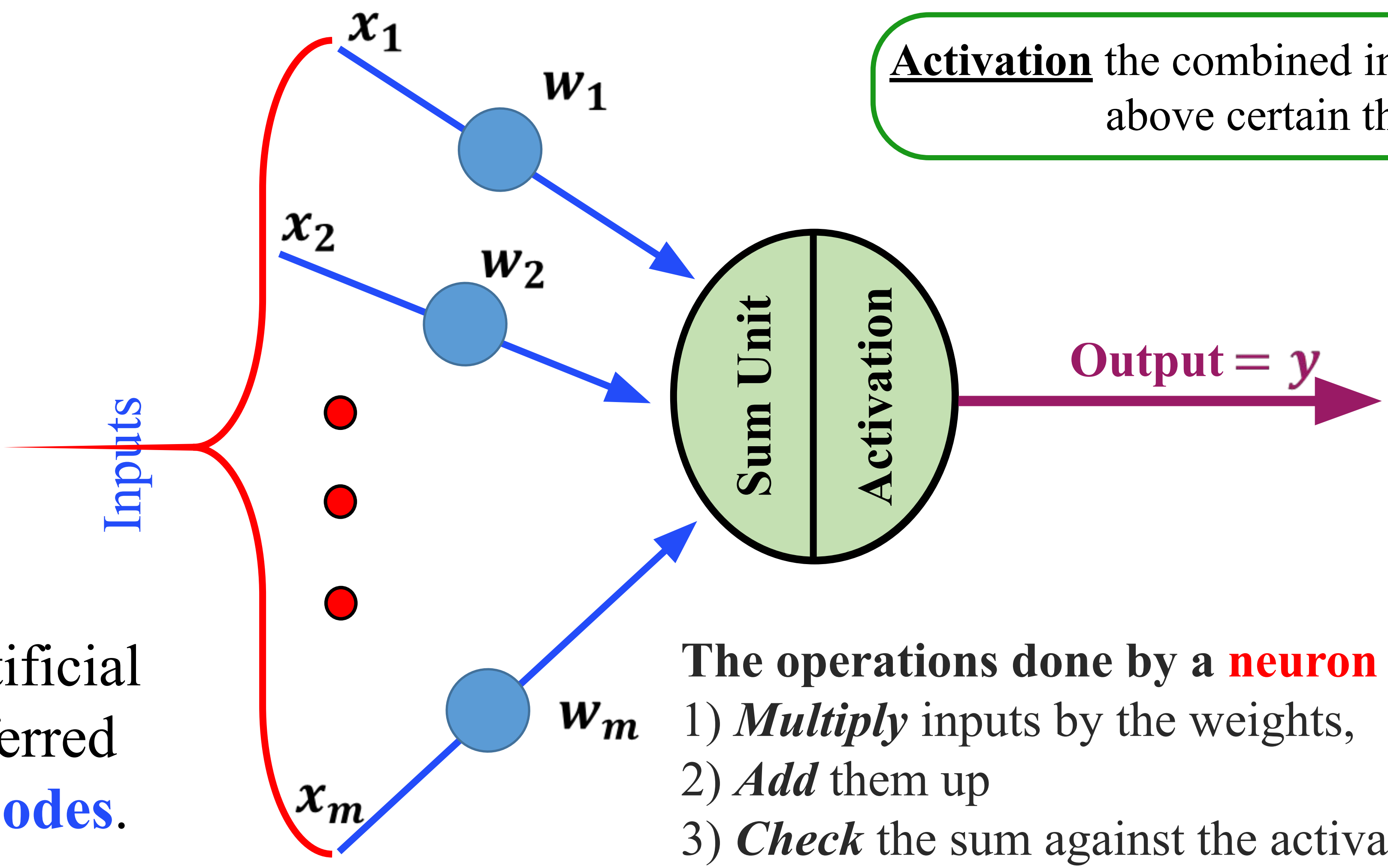


Technically, artificial neurons are referred to as **units** or **nodes**.

Artificial Neuron (cont’d)

Multiple inputs (x) each of which has a different strength, i.e., a **weight *w***

<u>Activation</u> the combined input must be above certain threshold

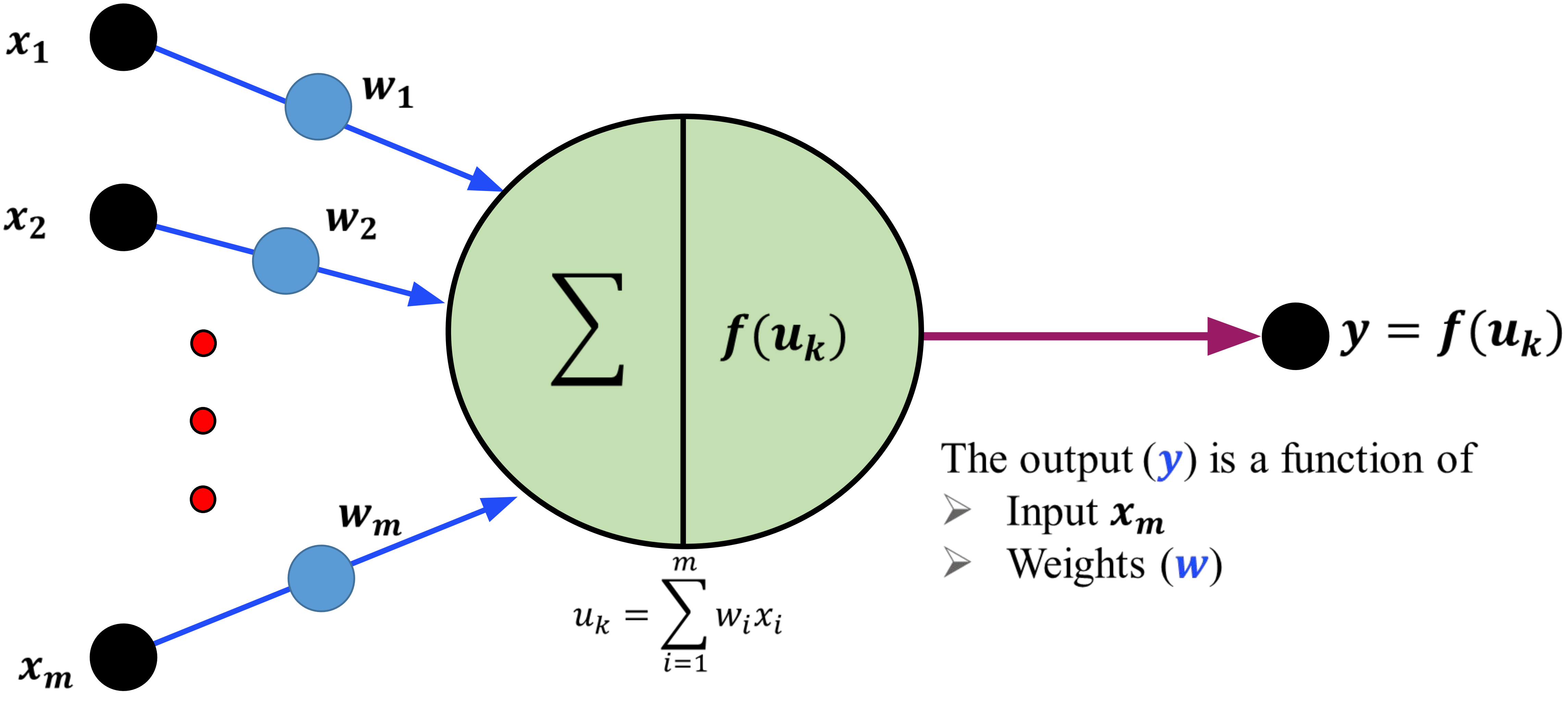


Technically, artificial neurons are referred to as **units** or **nodes**.

The operations done by a neuron are:

1) ***Multiply*** inputs by the weights,
2) ***Add*** them up
3) ***Check*** the sum against the activation and get y

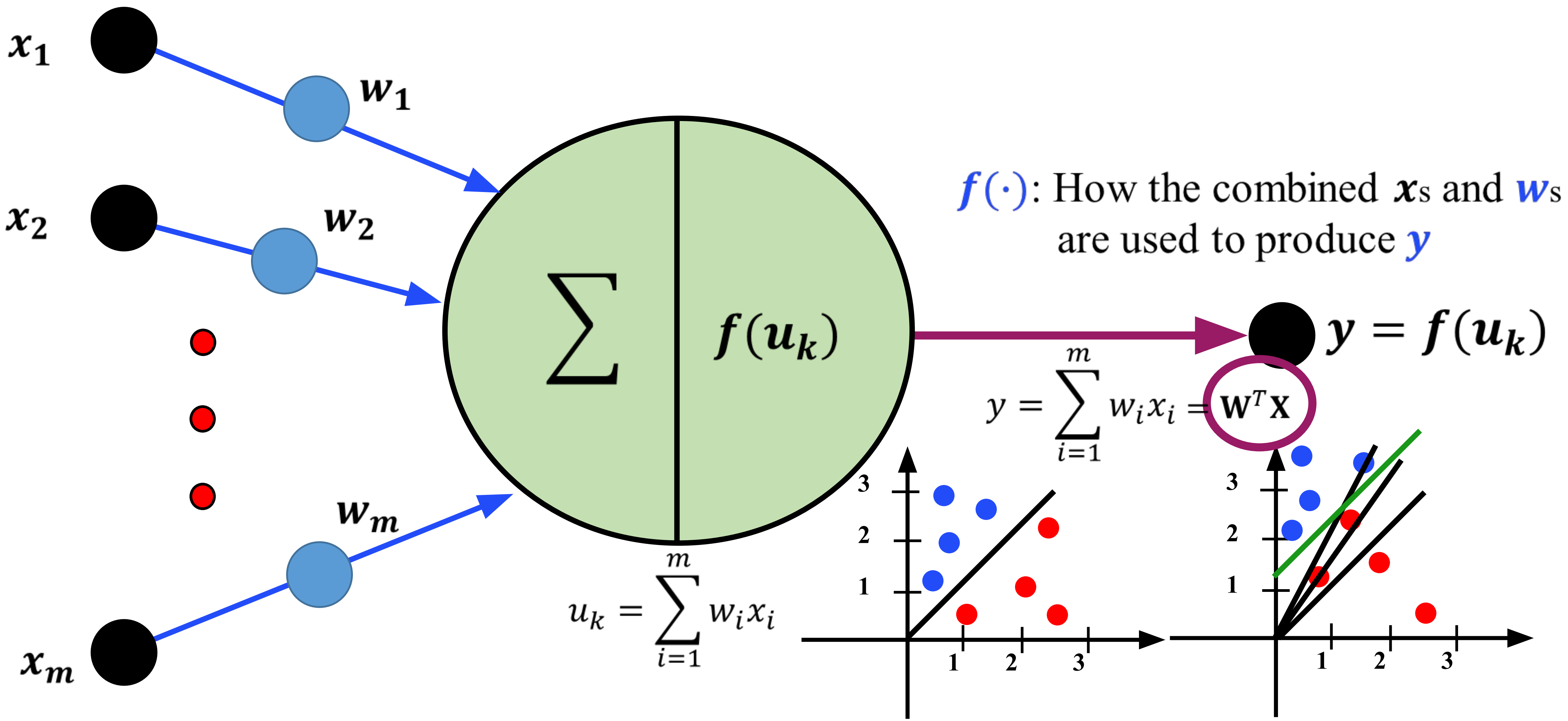
Artificial Neuron (cont'd)



The output (y) is a function of

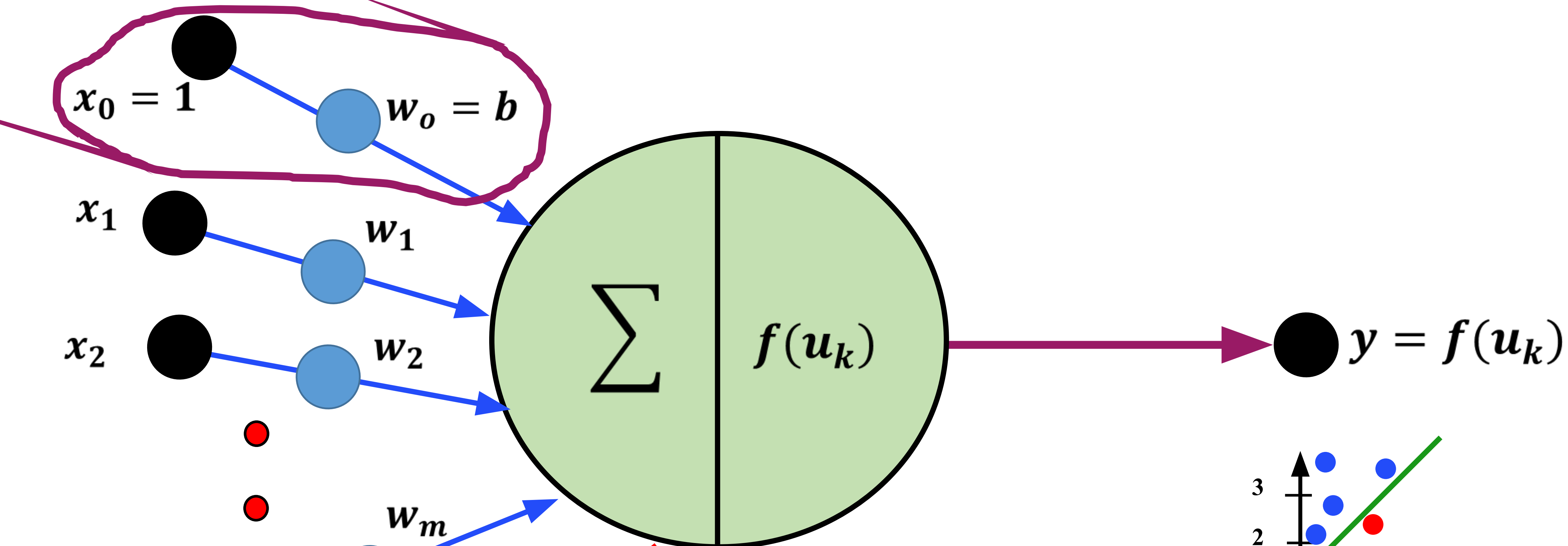
- Input x_m
- Weights (w)

Artificial Neuron (cont'd)



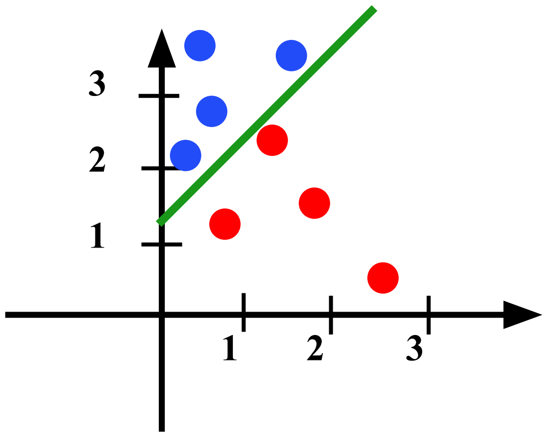
A **bias value (b)** is important to **full control** of the **activation function** (i.e., the output) for successful learning. This is a sort of **<u>regularization</u>**

Artificial Neuron (cont'd)



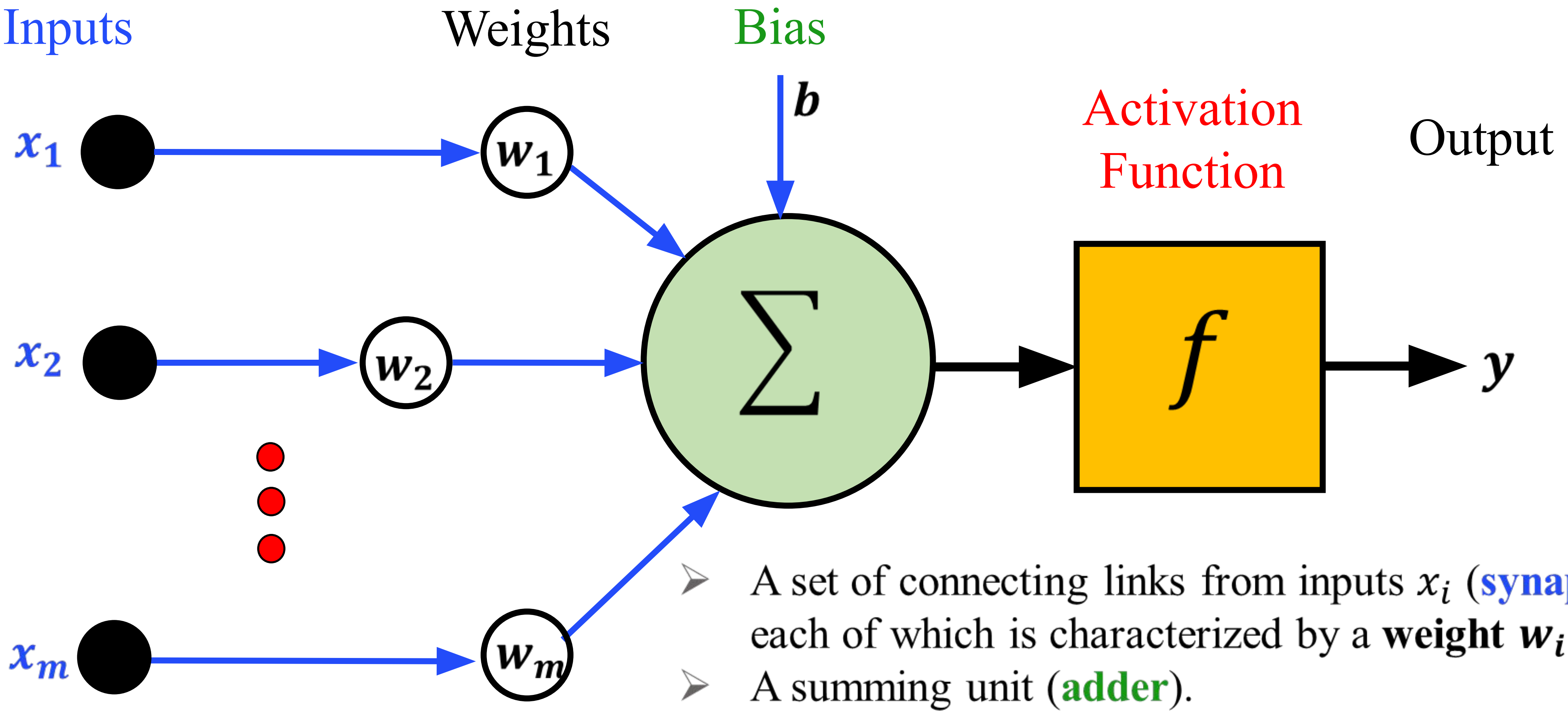
$$\cancel{u_k = \sum_{i=1}^{m} w_i x_i}$$

$$u_k = b + \sum_{i=1}^{m} w_i x_i$$



Artificial Neuron Network (ANN)

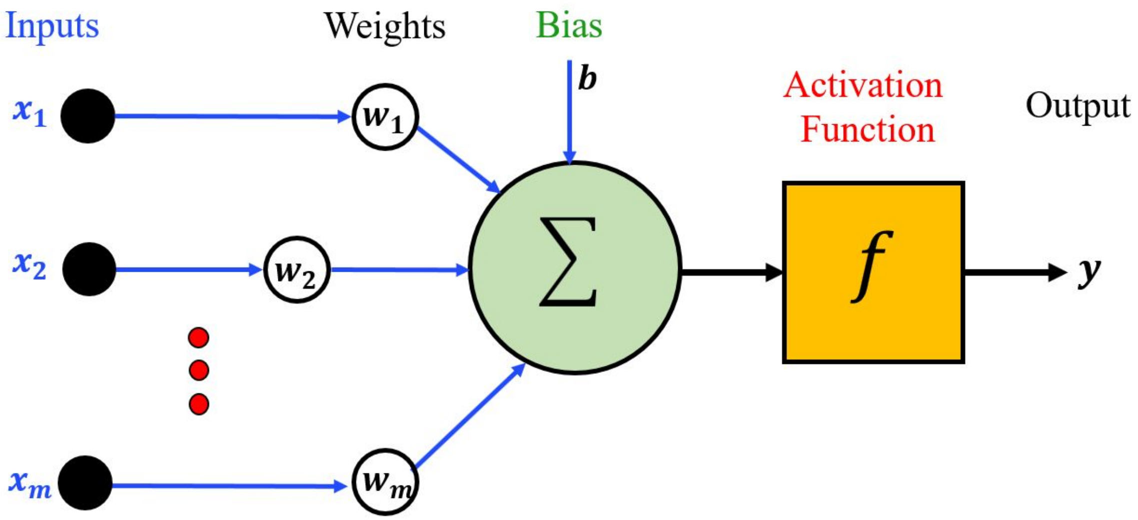
Basic Elements of any ANN:



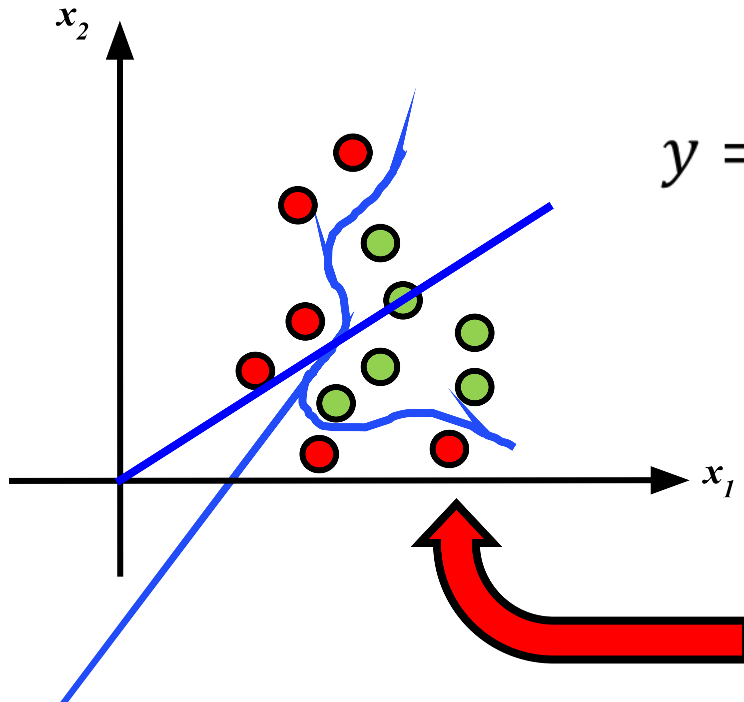
- A set of connecting links from inputs x_i (**synapses**) each of which is characterized by a **weight** w_i.
- A summing unit (**adder**).
- An activation function (**nonlinearity**)

ANN (cont'd)

- If the sum exceeds a certain threshold, the ANN (or the *perceptron*) fires an output value that is transmitted to the next unit(s)
- ANN uses nonlinear transfer function

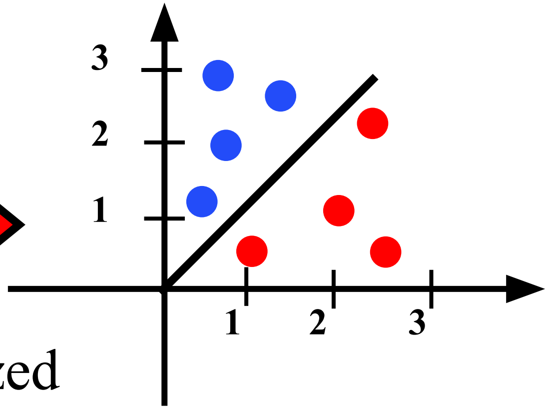


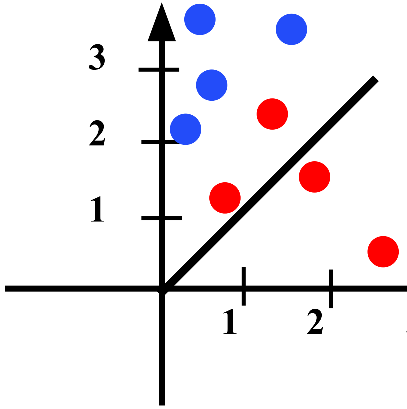
Why do we need nonlinearity?



$$y = f\left(b + \sum_{i=1}^{m} w_i x_i\right)$$

$$y = f\left(b + \mathbf{W}^{\mathrm{T}}\mathbf{X}\right)$$





Not good enough

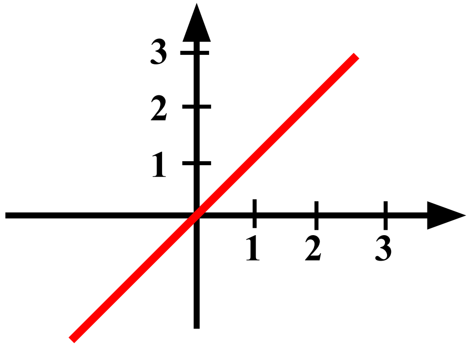
y is **linear**

- Can NOT be generalized
- LESS power to solve *complex nonlinear* problems

ANN Transfer Functions

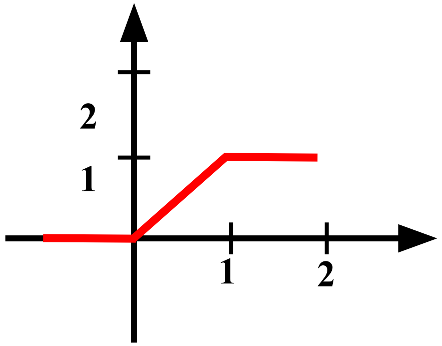
Linear

$$y_k = u_k$$



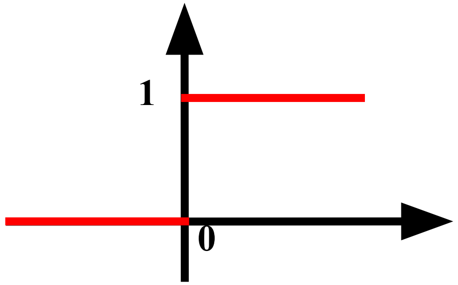
Saturating linear

$$y_k = \begin{cases} 1 & if\ u_k > 1 \\ u_k & if\ 0 \le u_k \le 1 \\ 0 & if\ u_k < 0 \end{cases}$$



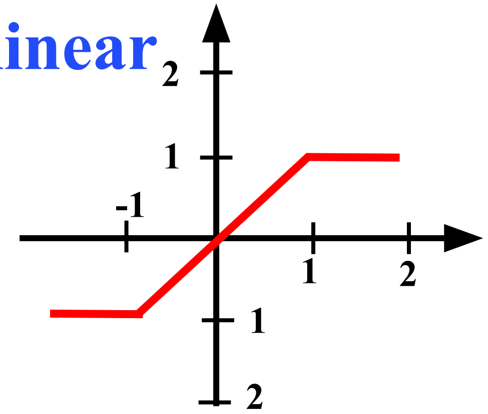
Hard Limit

$$y_k = \begin{cases} 1 & if\ u_k \ge 0 \\ 0 & if\ u_k < 0 \end{cases}$$



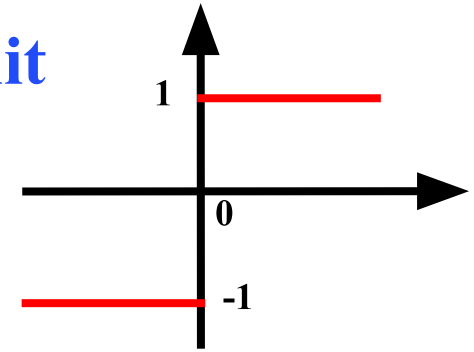
Symmetric Saturating linear

$$y_k = \begin{cases} 1 & if\ u_k > 1 \\ u_k & if\ 0 \le u_k \le 1 \\ -1 & if\ u_k < 0 \end{cases}$$



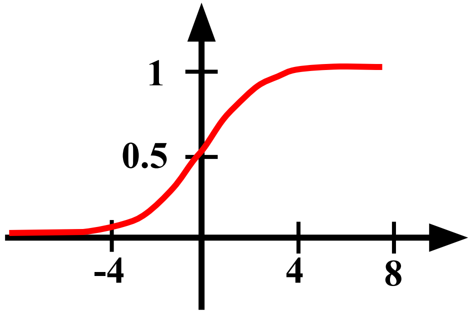
Symmetric Hard Limit

$$y_k = \begin{cases} 1 & if\ u_k \ge 0 \\ -1 & if\ u_k < 0 \end{cases}$$



Log Sigmoid

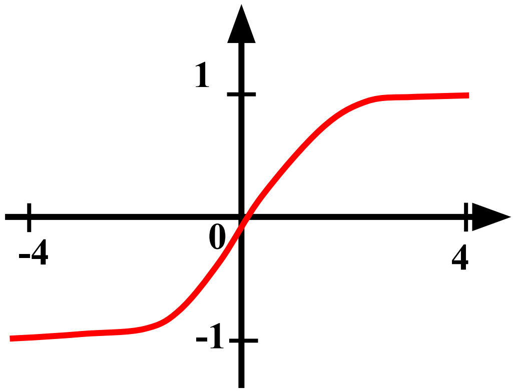
$$y_k = \frac{1}{1 + e^{-u_k}}$$



Artificial Neuron: Transfer Function

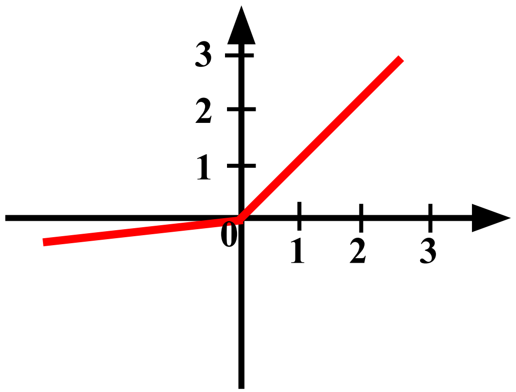
Hyperbolic Tangent Sigmoid

$$y_k = \frac{e^{u_k} - e^{-u_k}}{e^{u_k} - e^{-u_k}}$$



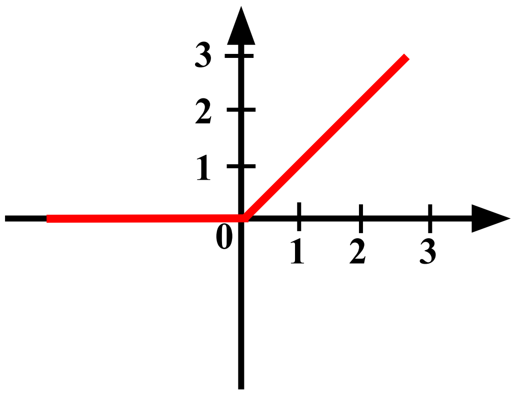
Leaky ReLU

$$y_k = \max(\epsilon u_k, u_k)$$
$$\epsilon \ll 1$$



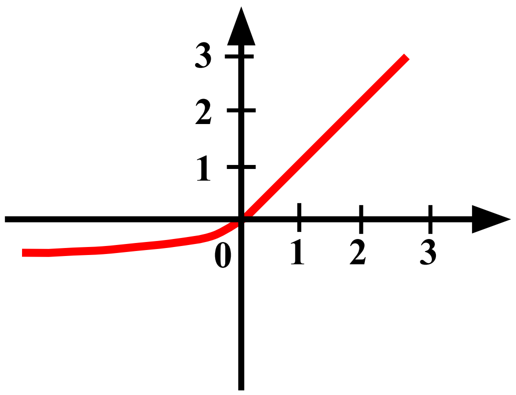
Rectified Linear Unit (ReLU)

$$y_k = \max(0, u_k)$$



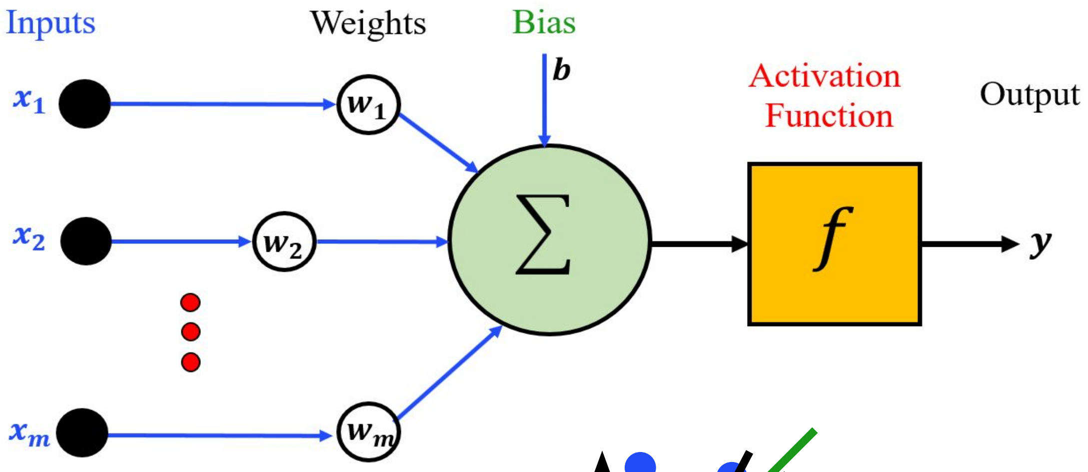
Exponential Linear Unit (ELU)

$$y_k = \begin{cases} S_k & if\ u_k \geq 0 \\ \alpha(e^{S_k} - 1) & if\ u_k < 0 \end{cases}$$



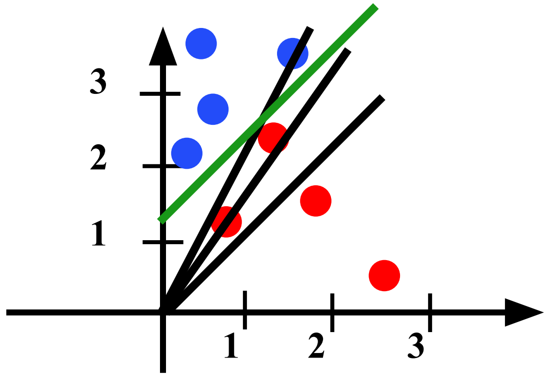
Artificial Neural Network (ANN)

- An artificial neural network (ANN) is a **massively parallel distributed processor** made up of **simple** processing units (neurons).



- ANN is capable of resolving paradigms that linear computing cannot resolve.

- ANNs are **adaptive systems**, i.e., parameters **can be changed** through a ***learning*** (**training**) process to suit the underlying problem.



- ANNs can be used in a *wide* variety of classification tasks, e.g., character recognition, speech recognition, fraud detection, medical diagnosis.

- "*neural networks are the second-best way of doing just about anything*" ***John Denker (AT&T Bell laboratories)***

Learning Process: Summary

- **Learning** is a *recursive* operation through which network **parameters** (***weights***) are *updated* in a way to reduce the **difference** (***error***) between network output and the *desired* (**target**) output

Set initial values of the weights (e.g., randomly)

Do

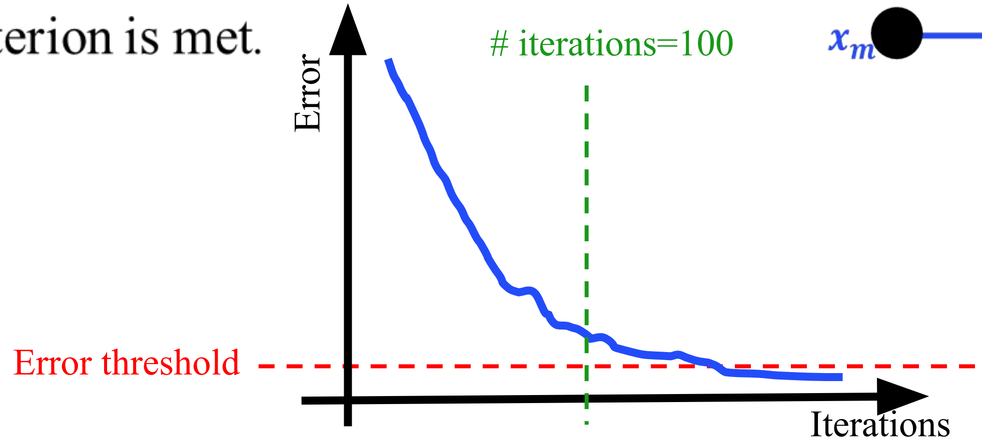
Compute the output function of a given input ($X(n)$)

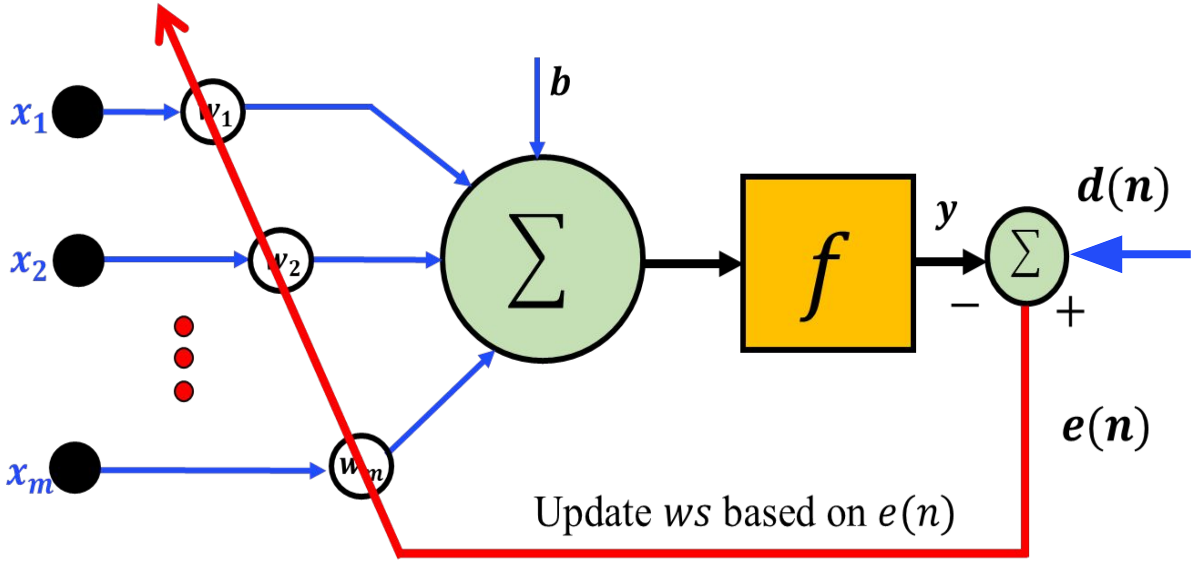
Evaluate the output by comparing $y(n)$ with $d(n)$.

Adjust the *weights*.

Loop until a criterion is met.

end





Criterion

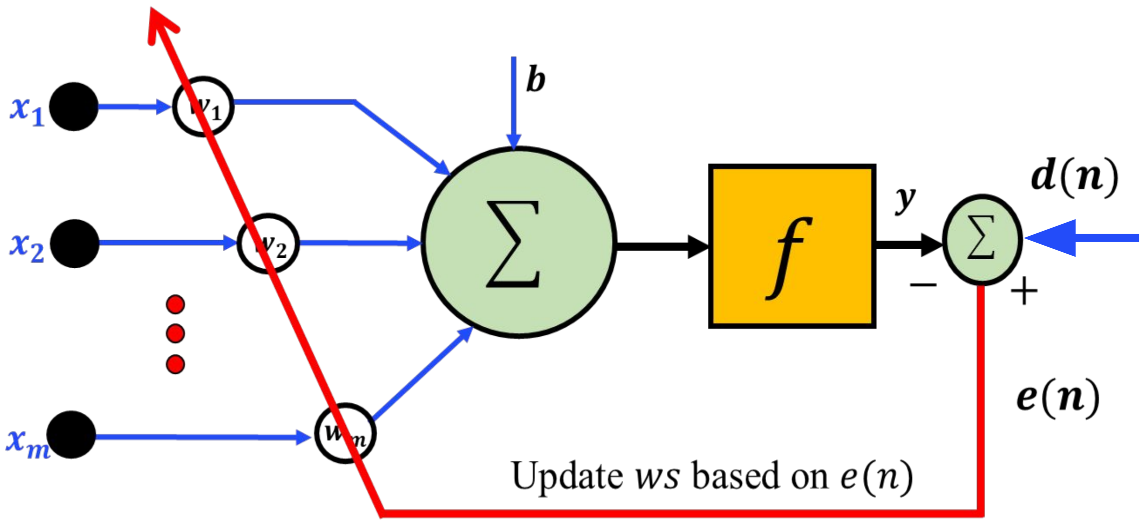
- Certain number of iterations
- Error threshold

Learning Process: Cost Function

- Our **objective** is to **reduce** the difference between the *actual* and *target* outputs (i.e., the error)

- This can be achieved by **minimizing** a **function** of the error (**error energy**)
  - This is called the ***cost function***.
  - Example is the **mean squared error**

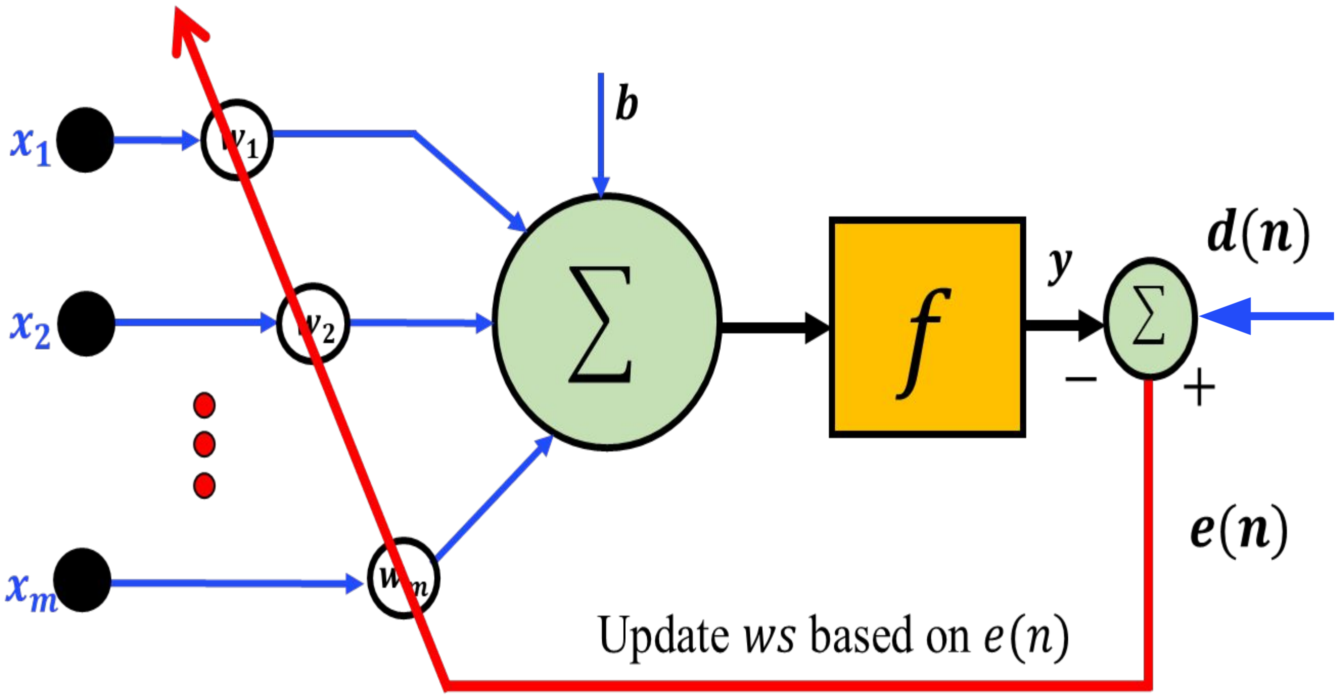
$$E(n) = \frac{1}{2}e^2(n) = \frac{1}{2}\left(d(n) - y(n)\right)^2$$



$$e(n) = d(n) - y(n)$$

Learning Process: Epoch

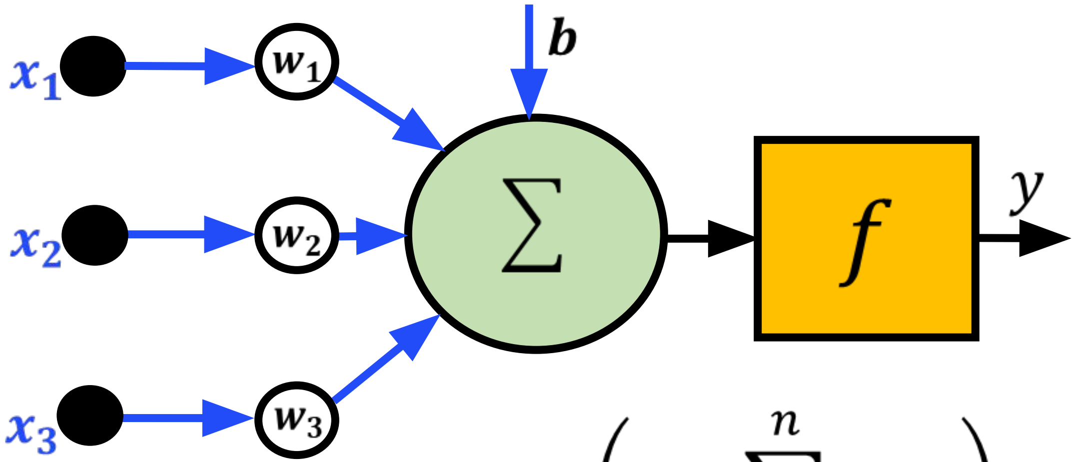
n	x_1	x_2	x_3	Output
1	10.33	56	0.56	0.7
2	8.97	48	0.61	0.9
3	11.01	49	0.49	0.8
4	9.32	53	0.89	0.8
5	10.51	50	0.71	0.7
6	12.10	59	0.90	0.8
	⋮	⋮	⋮	
1996	7.99	61	0.59	0.9
1997	11.36	52	0.63	0.9
1998	12.09	48	0.78	0.8
1999	10.81	55	0.87	0.7
2000	13.00	53	0.91	0.6



The training cycle at which **All** the training samples have been used by the network is called the ***epoch***

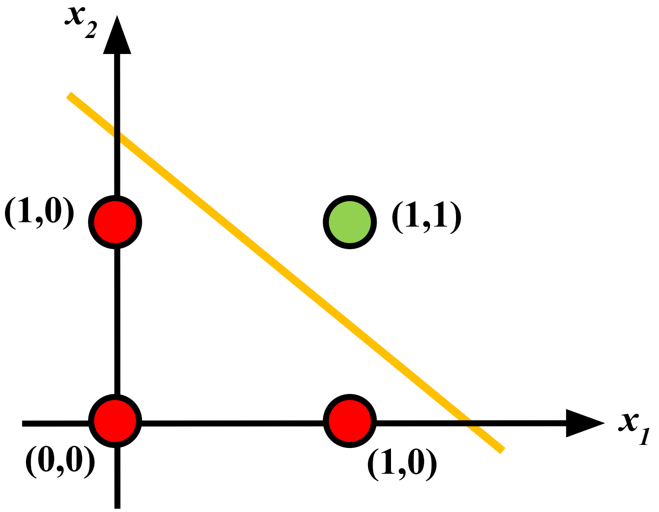
ANN Examples

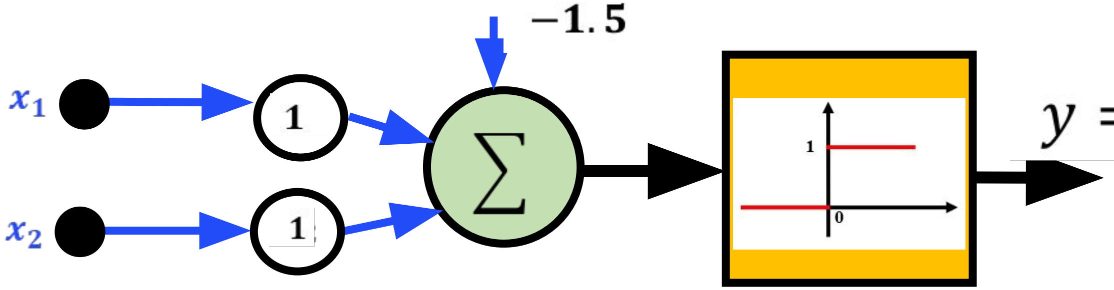
- One layer *feedforward* neural network is called the *perceptron*
- Can solve linear function, e.g., AND, OR, NOT



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

x_1	x_2	y
0	0	**0**
1	0	**0**
0	1	**0**
1	1	**1**

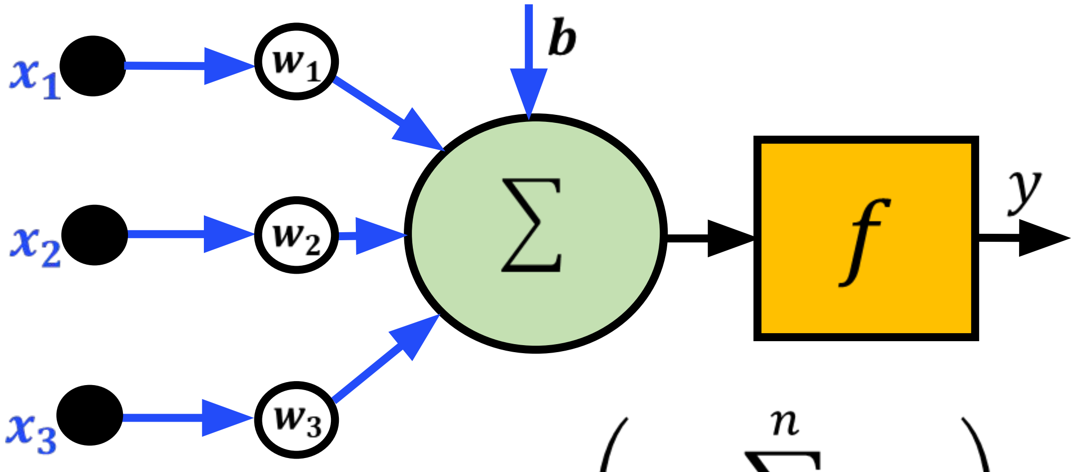




$$y = step(\underbrace{-1.5}_{b} + \underbrace{1}_{w_1}.x_1 + \underbrace{1}_{w_2}.x_2)$$

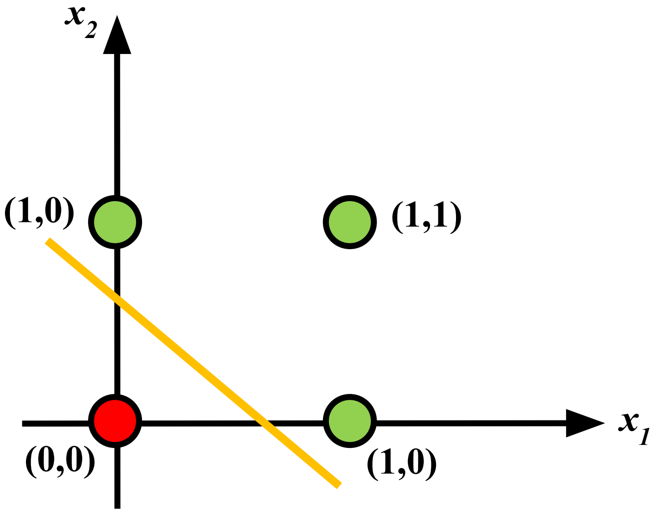
ANN Examples (cont'd)

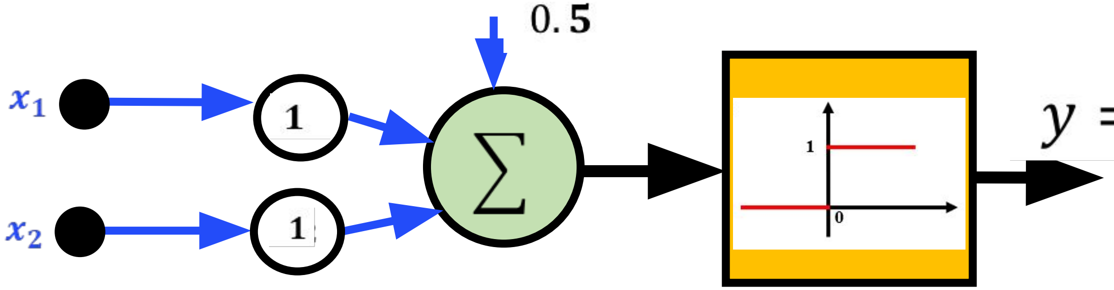
- One layer *feedforward* neural network called the *perceptron*
- Can solve linear function, e.g., AND, OR, NOT



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

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

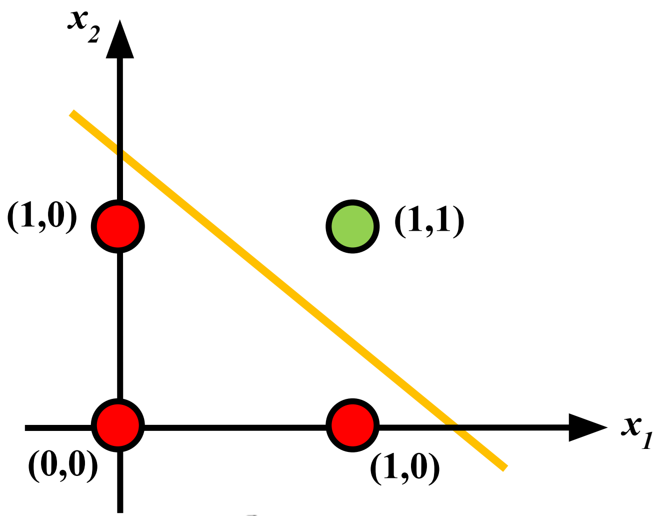




$$y = step(\underbrace{0.5}_{b} + \underbrace{1}_{w_1}.x_1 + \underbrace{1}_{w_2}.x_2)$$

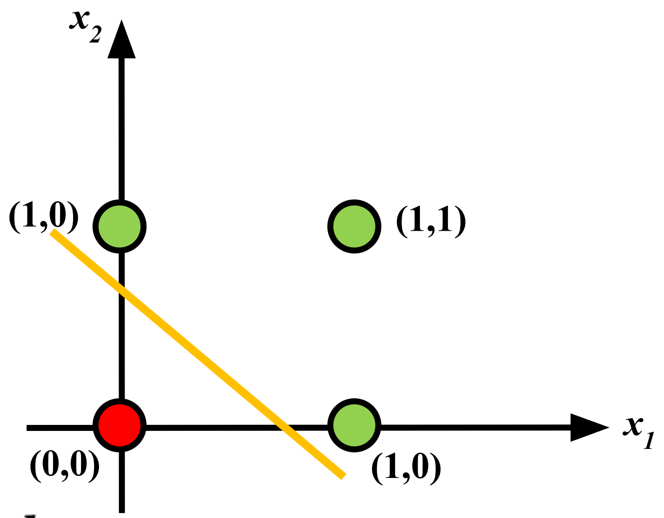
ANN Examples (cont'd)

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1



$$y = step(\overset{b}{-1.5} + 1.x_1 + 1.x_2)$$

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

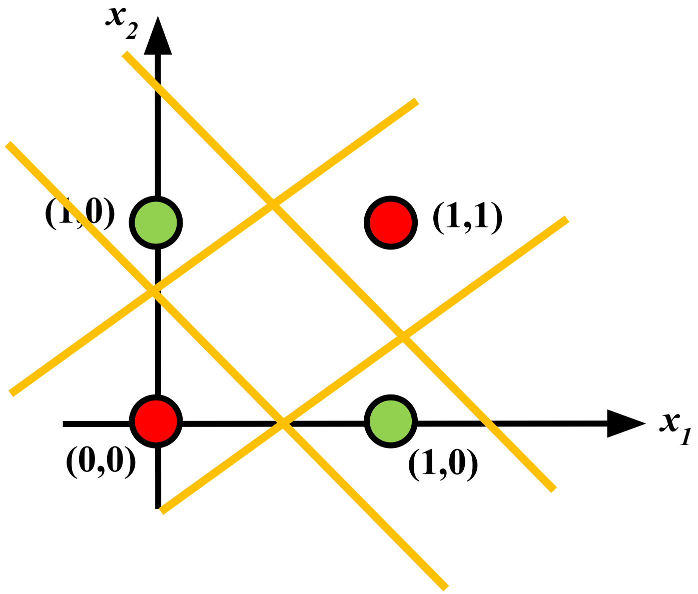


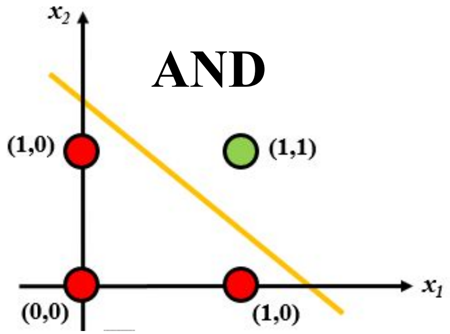
$$y = step(\overset{b}{0.5} + 1.x_1 + 1.x_2)$$

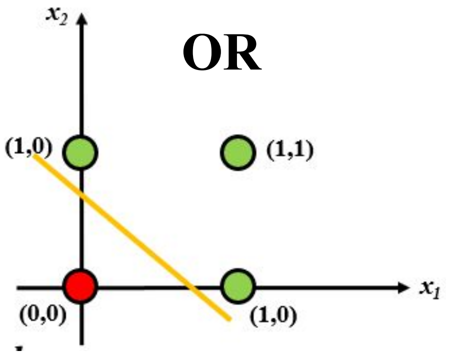
- Solving linearly, means the **decision boundary** is linear (straight line in 2D and a plane in 3D)
- The bias term (***b***) alters the **position**, but not the **orientation**, of the decision boundary
- The weights (w_1, w_2, ...w_m) determine the gradient

ANN Examples: XOR function

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0







- The XOR function is said to be **not linearly separable**
- If **<u>one neuron</u>** defines **<u>one line</u>** through input space, **<u>what do we need to have two lines?</u>**
- We need to have two neurons working in ***parallel*** (*rather than in different layers*).
- We would need a **multilayer neural network** to model (or to separate the two classes) the XOR function.

Multilayer Perceptron (MLP)

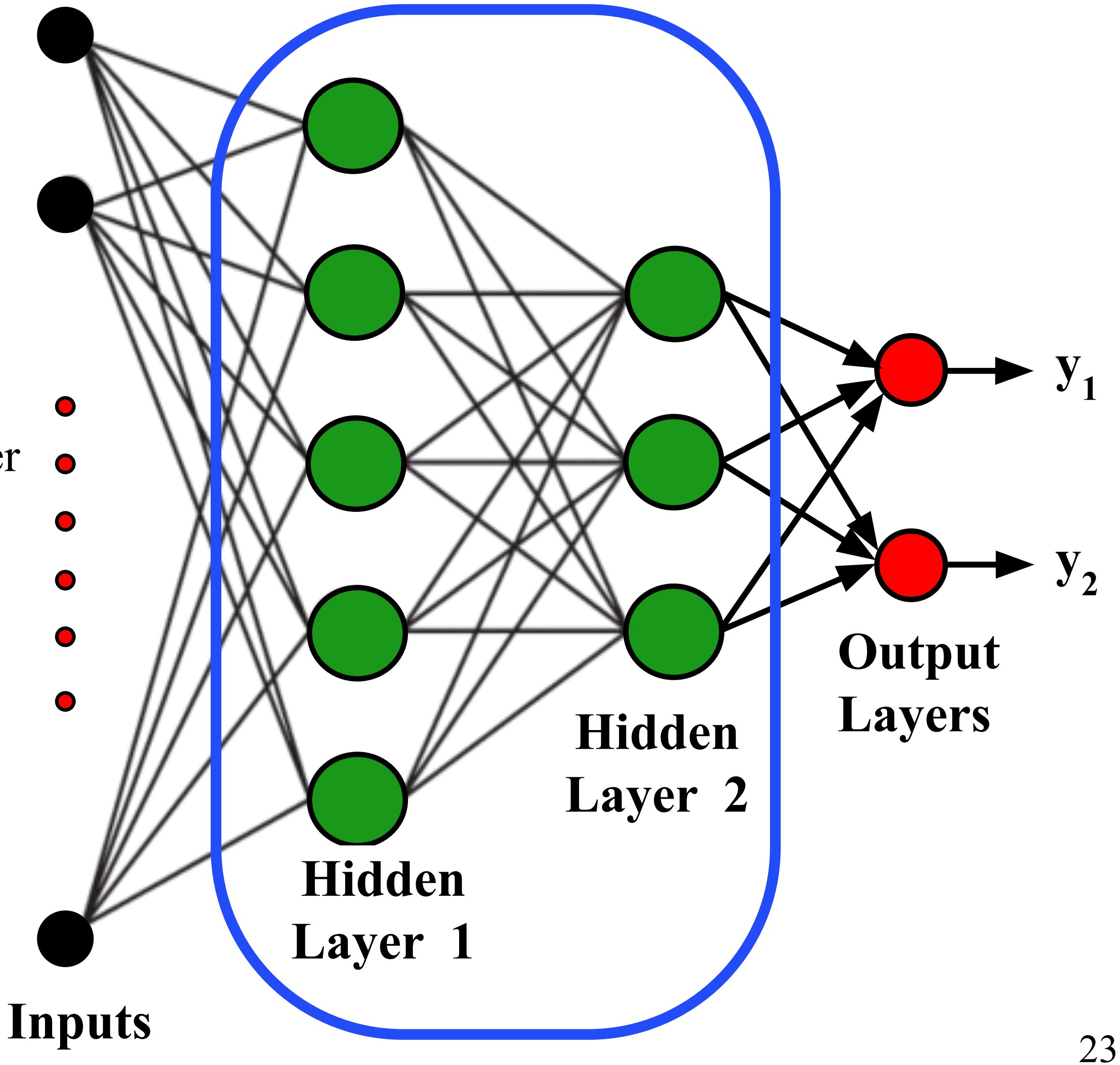
- More layers between the ***input*** the ***output*** layers
- Fully connected layers
- Multiple neurons at the output layers

  $\boldsymbol{y_j}$,**j** ∈ $\boldsymbol{C}$ C is set of all neurons at the output layer

- Error **backpropagation** is used for learning

$$e(n) = d(n) - y(n)$$

- Weight adjustments are applied so as to minimize $e(n)$ in a statistical sense.



Gradient Descent

The **delta rule** is a gradient descent learning rule for updating the weights of an artificial neuron inputs in a single-layer NN

$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n)$$

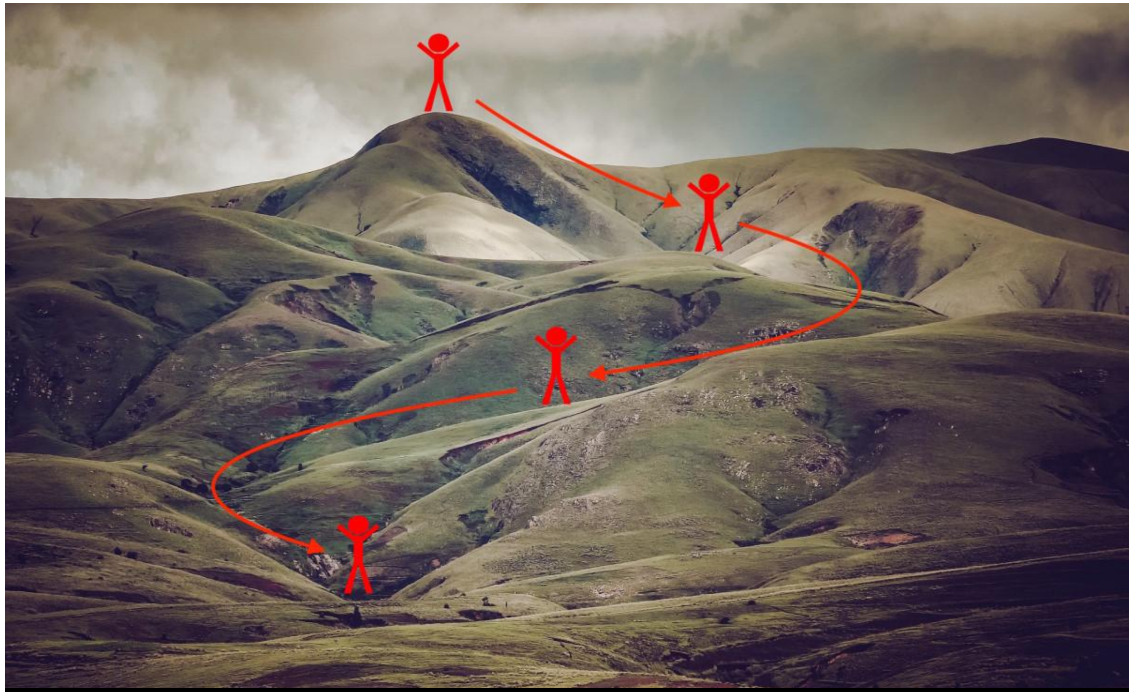
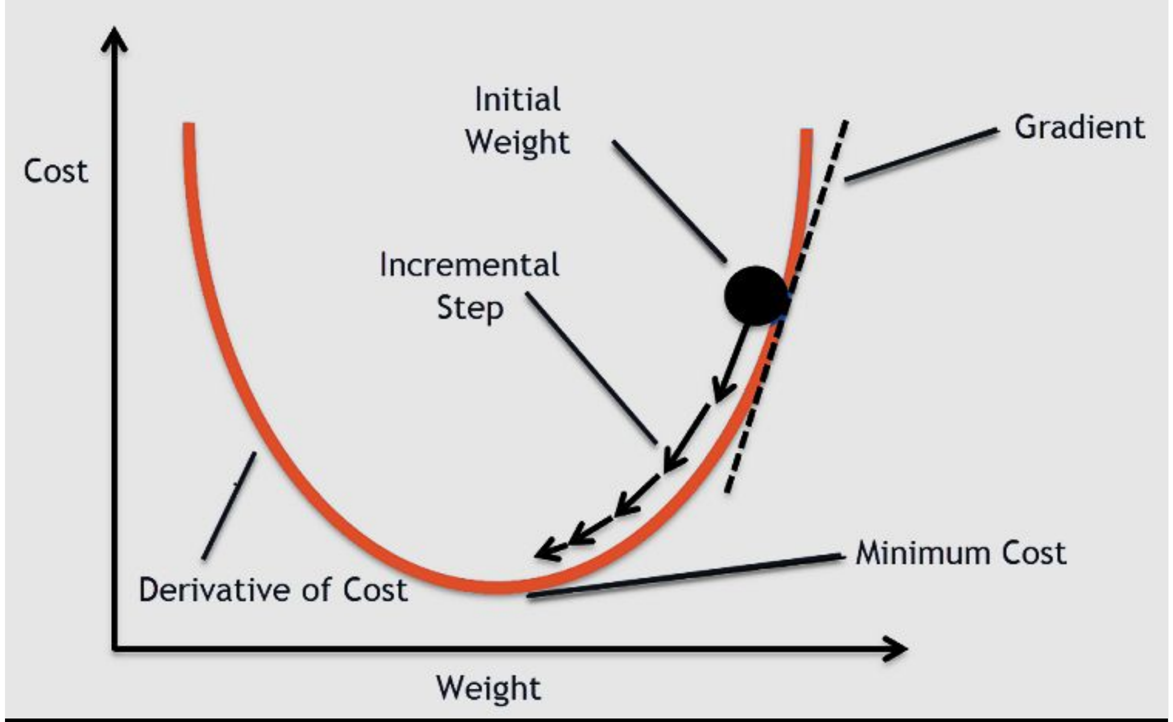


Image Source: https://datascience-enthusiast.com/figures/cost.jpg



https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1

The goal of gradient descent is to *iteratively* take steps towards **lower** regions (minima) of the loss function

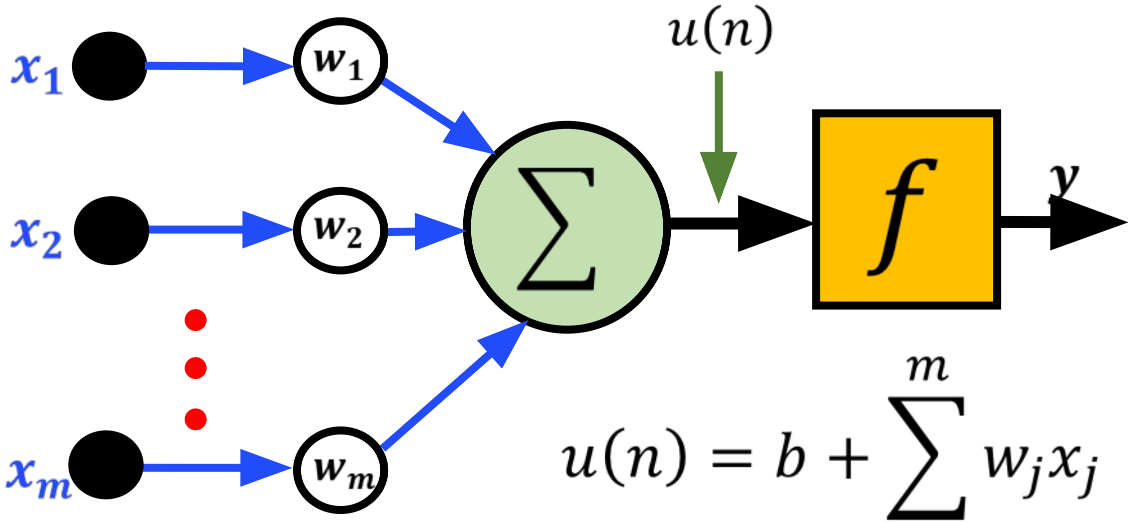
Gradient Descent (cont'd)

For *linear activation function*, the weight adjustment for a **neuron *k*** is given by

$$\Delta w_{kj}(n) = \eta * e_k(n) * x_j(n) \qquad j = 1,2, \ldots\ldots m$$

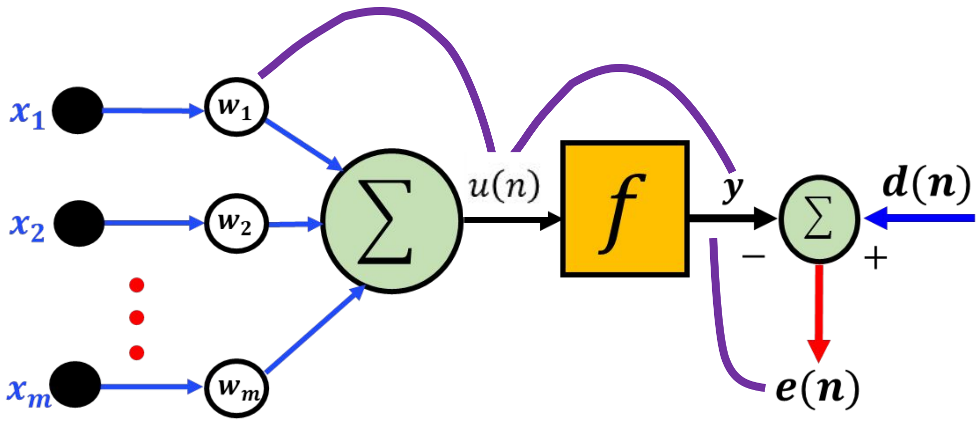
For **any activation** function f:

$$\Delta w_{kj}(n) = \eta * e_k(n) * f'(u(n)) * x_j(n)$$



$$u(n) = b + \sum_{j=1}^{m} w_j x_j$$

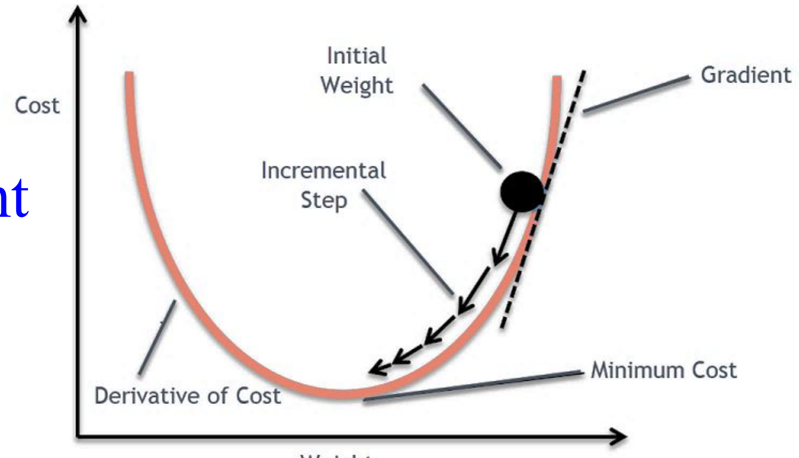
Gradient Descent (cont’d)



$$\Delta w_{kj} = \boxed{-}\eta * \frac{\partial E}{\partial w_j}$$

minimization

gradient



https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1

By applying the chain rule

$$\frac{\partial E}{\partial w_j} = \left(\frac{\partial E}{\partial e}\right)\left(\frac{\partial e}{\partial y}\right)\left(\frac{\partial y}{\partial u}\right)\left(\frac{\partial u}{\partial w_j}\right)$$

$$E(n) = \frac{1}{2}e^2(n) \Rightarrow \frac{\partial E}{\partial e} = e$$

$$e(n) = d(n) - y(n) \Rightarrow \frac{\partial e}{\partial y} = -1$$

$$y(n) = f(u(n)) \Rightarrow \frac{\partial y}{\partial u} = f'(u(n))$$

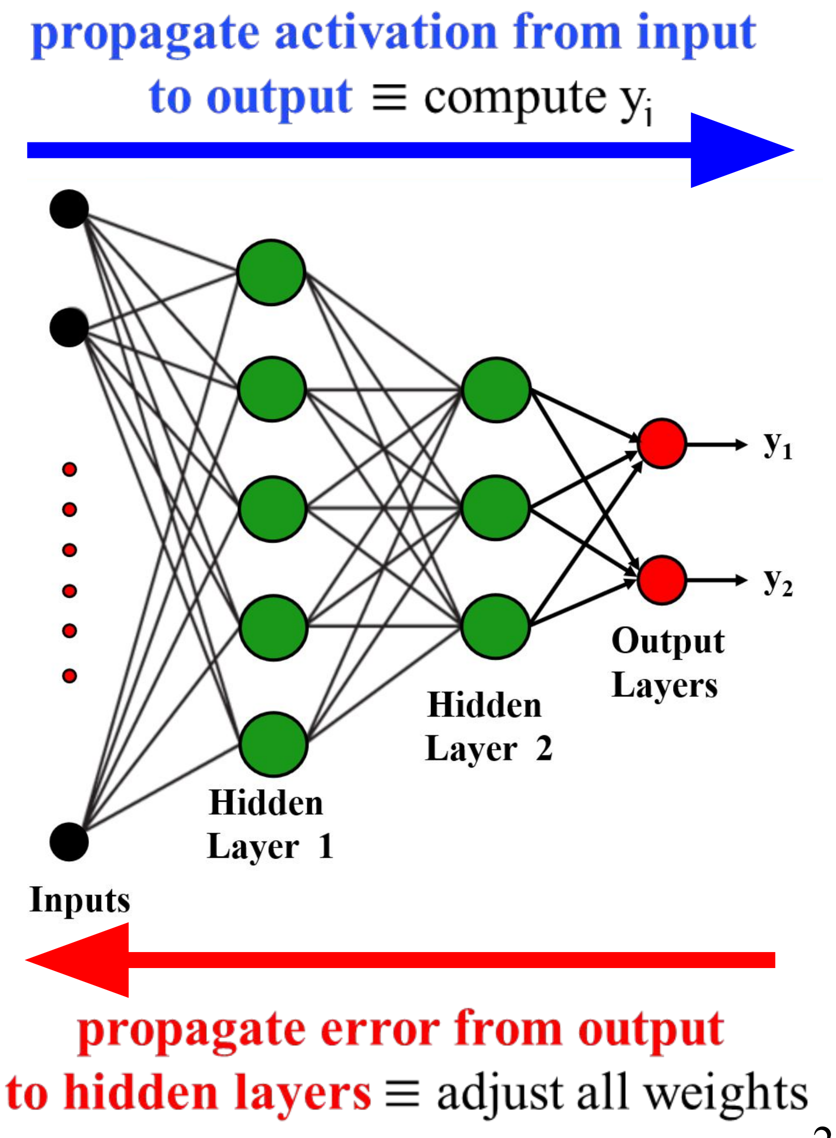
$$u(n) = \sum_{j=1}^{m} w_j x_j \Rightarrow \frac{\partial u}{\partial w_j} = x_j$$

$$\Delta w_{kj} = -\eta * (e)(-1)\left(f'(u(n))\right)(x_j)$$

$$\Delta w_{kj} = \eta * e * f'(u(n))x_j$$

Backpropagation

- It is based on the ***gradient search*** technique to minimize the **<u>cost function</u>** ≡ squared error between the network output and the *target* output
- It is **recursive** application of the *chain rule* to compute the *gradients*

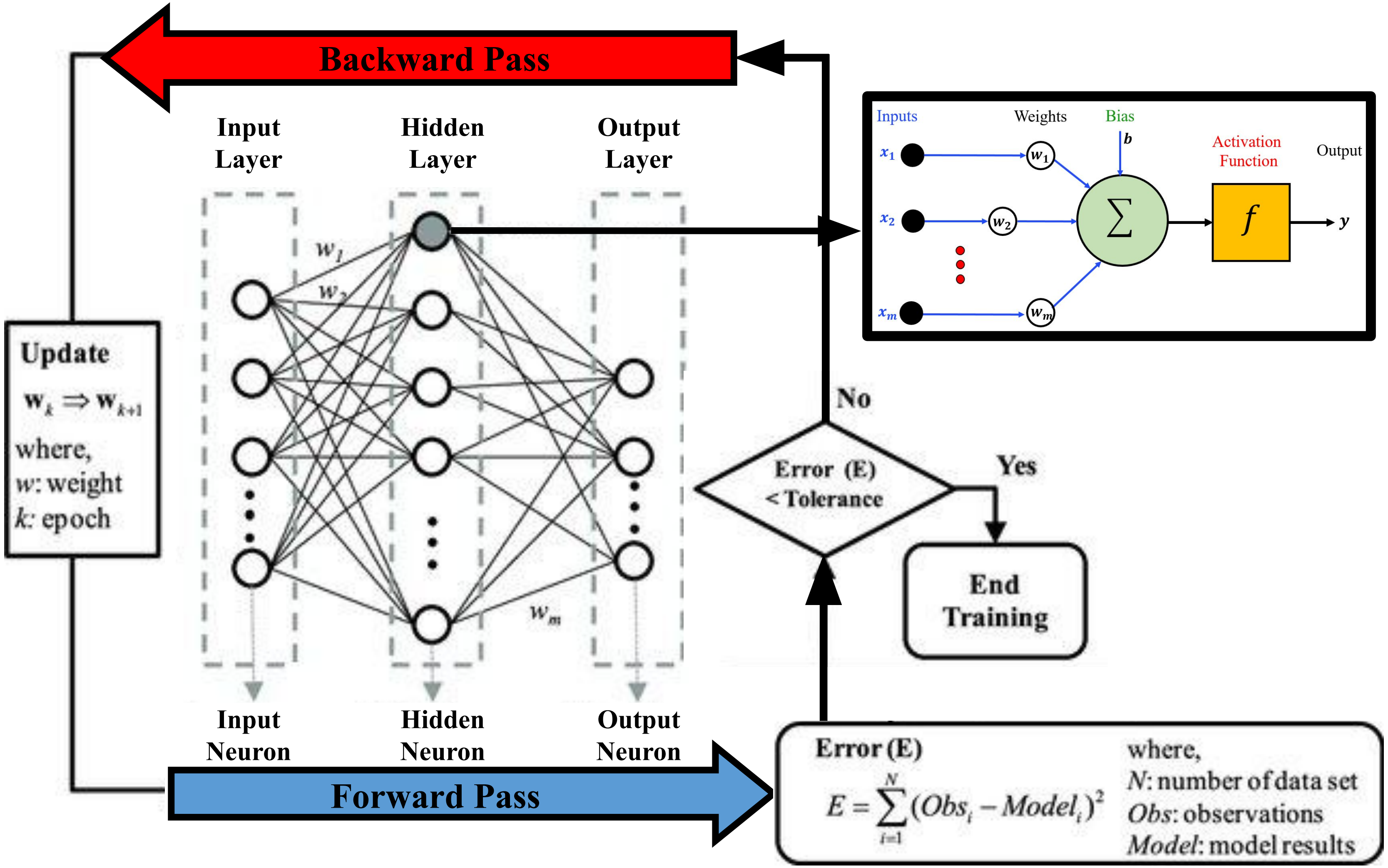


Please see the following for all details about mathematical derivation:
https://www.jeremyjordan.me/neural-networks-training/

Fully Connected ANN: Backpropagation

- Backpropagation algorithm is the heart of ANN training

  - **Forward pass**:
    - $(x, b, f, w_{initial})$→compute (y) and the cost function (L)
    - Check terminating criteria
  - **Backward pass**:
    - **Compute** the gradients of L w.r.t each parameter (***weights & biases***)
    - **Take** a gradient ***descent*** step towards the **minimum**
    - **Update** the parameters



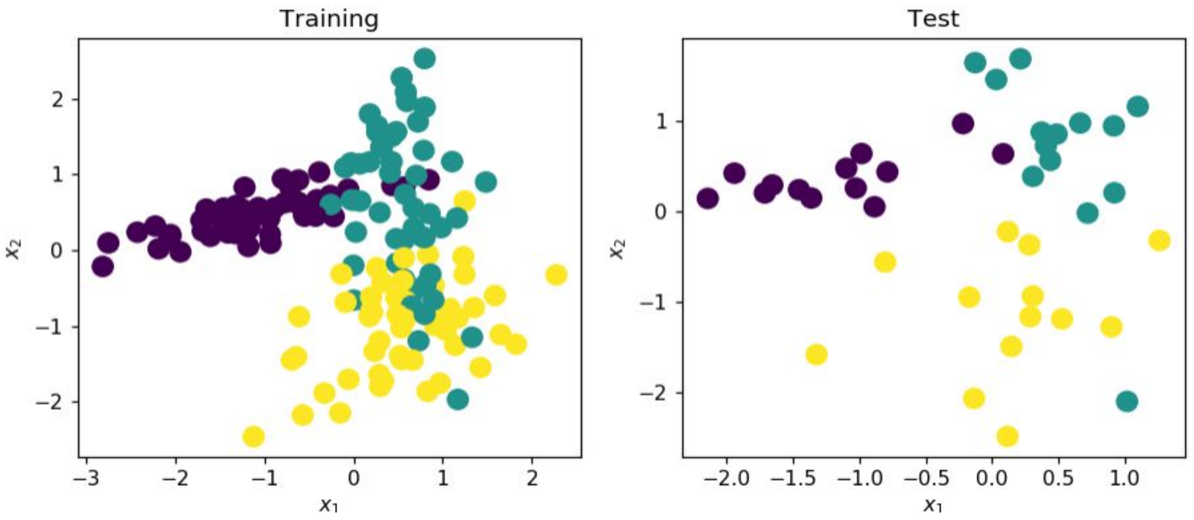
Modified from: https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

Example

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
```

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples = 200, n_features=2,
                           n_redundant=0, n_informative=2,
                           n_clusters_per_class=1,
                           n_classes=3, random_state = 0)
```

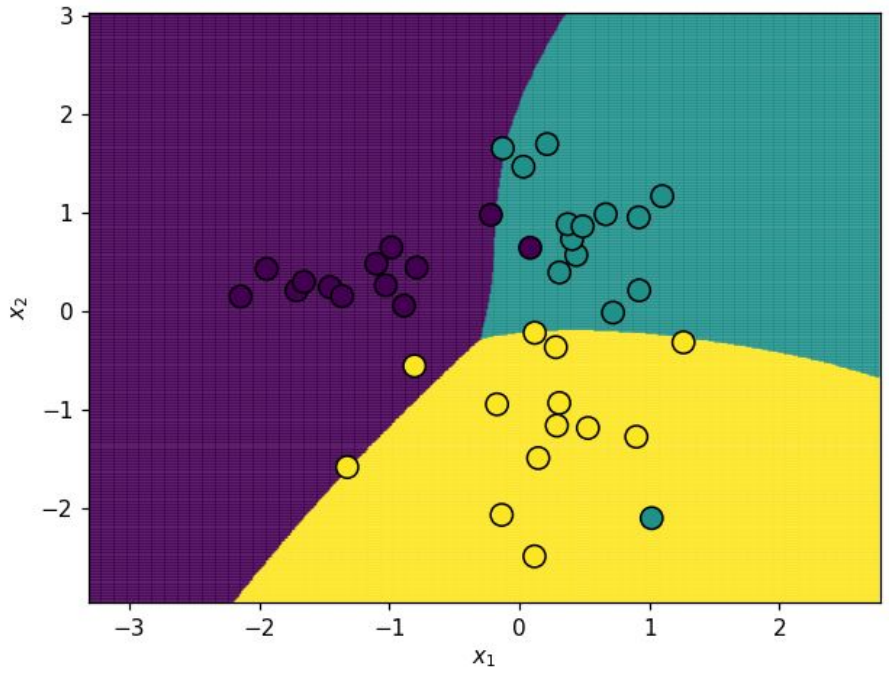
```
X_train, X_test, y_train,  y_test = train_test_split(X, y,
                                                     test_size = 0.2,
                                                     random_state = 0)
scaler_1 = StandardScaler().fit(X_train)
X_train = scaler_1.transform(X_train)
X_test = scaler_1.transform(X_test)
```



Example

```python
C = MLPClassifier(activation='logistic', hidden_layer_sizes=5,
                  random_state=0,max_iter=3000)
C.fit(X_train, y_train)
#prediction on the grid
grid_sample_dist = 0.05 # sampling period
x1_min, x1_max = X_train[:, 0].min() - .5, X_train[:, 0].max() + .5
x2_min, x2_max = X_train[:, 1].min() - .5, X_train[:, 1].max() + .5
x1grid, x2grid = np.meshgrid(np.arange(x1_min, x1_max, grid_sample_dist),
                             np.arange(x2_min, x2_max, grid_sample_dist))
Z = C.predict(np.c_[x1grid.ravel(), x2grid.ravel()])
Z = Z.reshape(x1grid.shape)
yp = C.predict(X_test)
```

```python
plt.figure(2)
plt.pcolormesh(x1grid, x2grid, Z, alpha = 0.4)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', edgecolor = 'k',
            c=y_test,s=100)
plt.show()
```



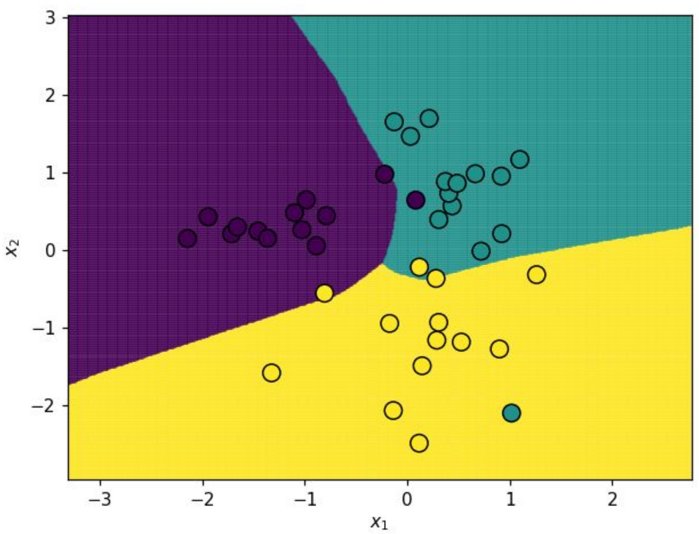

```python
print(accuracy_score(y_test,yp))
print("loss: ", str(C.loss_))
```

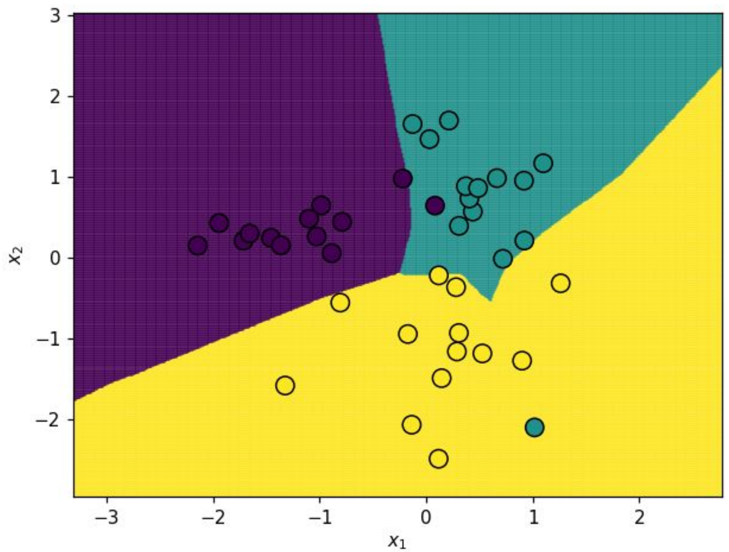
```
0.875
loss:  0.44192572086792625
```

Example

```
C2 = MLPClassifier(activation='relu', hidden_layer_sizes=(15,5),
                   random_state=0,max_iter=3000)
```

```
C3 = MLPClassifier(activation='relu', hidden_layer_sizes=(15,5,3),
                   random_state=0,max_iter=3000)
```






```
print(accuracy_score(y_test,yp2))
print("loss: ", str(C2.loss_))
```

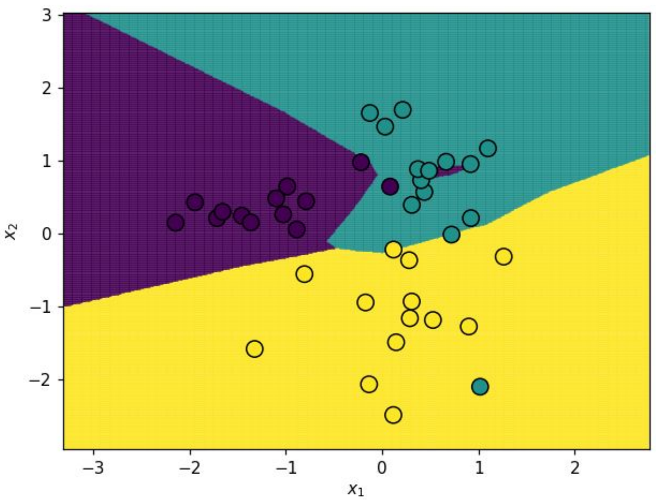
```
0.9
loss:  0.3208181670054528
```

```
print(accuracy_score(y_test,yp3))
print("loss: ", str(C3.loss_))
```

```
0.95
loss:  0.32993736221822
```

Example

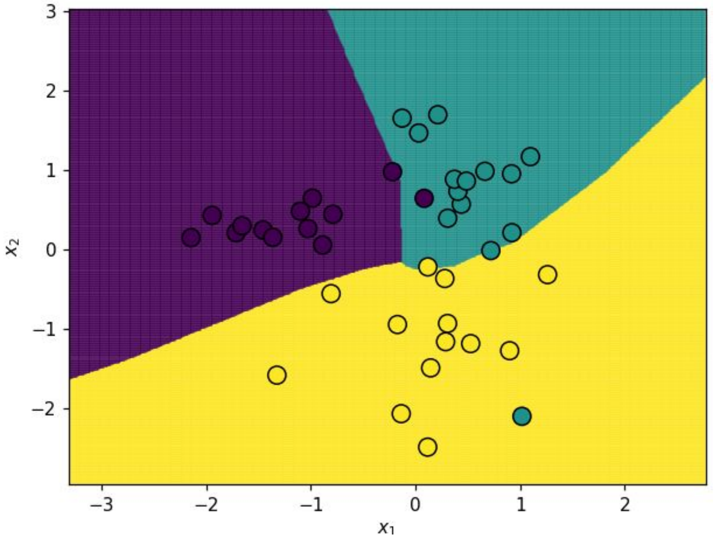
```
C3 = MLPClassifier(activation='relu', hidden_layer_sizes=(15,5,3),
                   random_state=0,max_iter=3000, alpha = 0)
```




```
print(accuracy_score(y_test,yp4))
print("loss: ", str(C4.loss_))
```

```
0.875
loss:  0.283498912076085
```

```
C3 = MLPClassifier(activation='relu', hidden_layer_sizes=(15,5,3),
                   random_state=0,max_iter=3000, alpha = 0.1)
```




```
print(accuracy_score(y_test,yp4))
print("loss: ", str(C4.loss_))
```

```
0.925
loss:  0.33669512966470094
```

Thank You
&
Questions