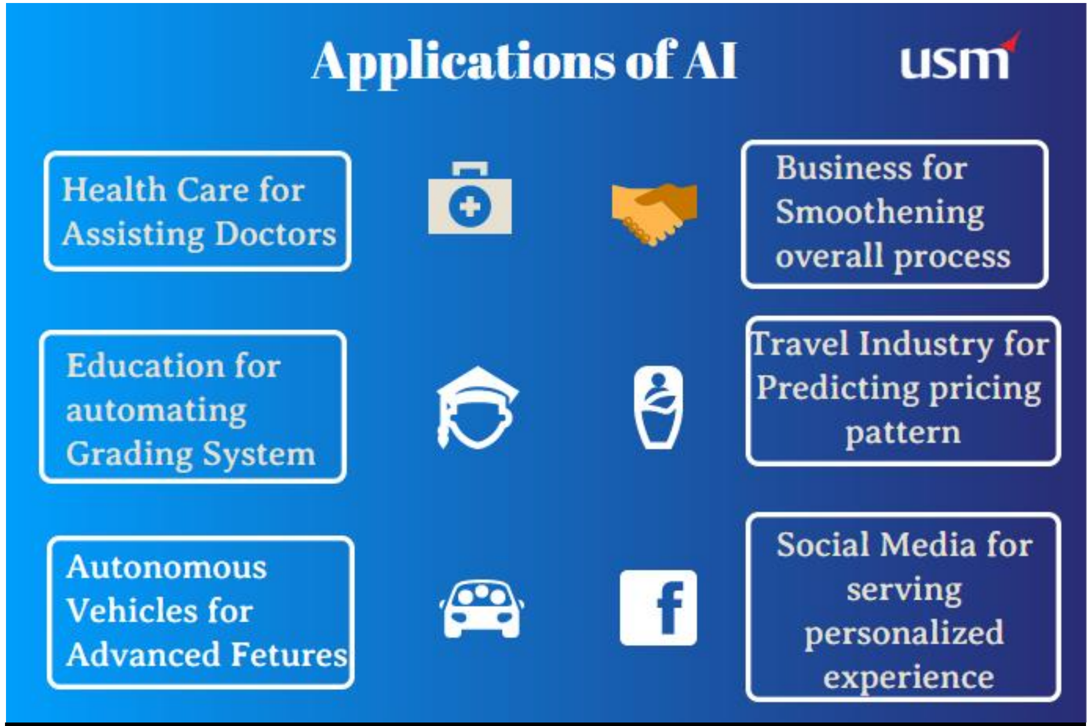
Outline

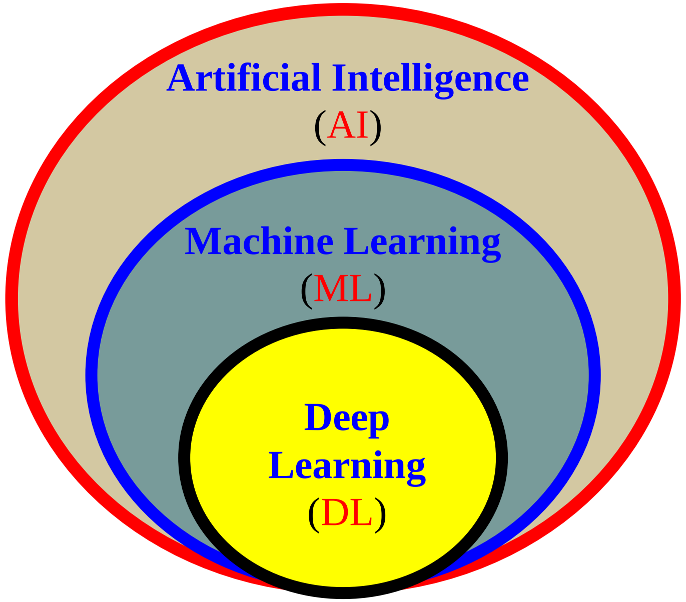
- Introduction
- Learning paradigms
- History of artificial neural networks (ANN)
- Modelling of ANNs
- Multilayer perceptron (MLP)
- Gradient Descent and Backpropagation
- ANN types, design and issues
- Validation techniques for efficient learning
- Assignment(s)
- Conclusion

Introduction

- The ever-increasing popularity of artificial intelligence (AI) and machine learning (ML) provides a groundbreaking impetus on many aspects of our life.
  - **Artificial Intelligence** (AI) are those set of human-designed tools (programs) to do things that is typically done by human
  - **Machine learning** (ML) is an AI field where *machine* can learn new things *through* **<u>experience</u>** without the involvement of a human.
  - **Deep learning** (DL) is a ML subset where machines adapt and learn from vast amount of data



https://pvvajradhar.medium.com/ai-applications-in-various-fields-748dde27516d



Categories of Machine Learning

Learning Paradigms

Supervised Learning

- Learning with a teacher
- Data with known output (label) is given
- Classification and Regression

Reinforcement Learning

- Interactive learning environment by trial and error using feedback from its own actions and experiences.

Unsupervised Learning

- Learning without a teacher
- No labels
- Machine understand the data
- Clustering

Support Vector Machine (SVM), K Nearest Neighbours (KNN), Decision Trees, Random Forest *Feedforward* Artificial Neural Network (ANN)

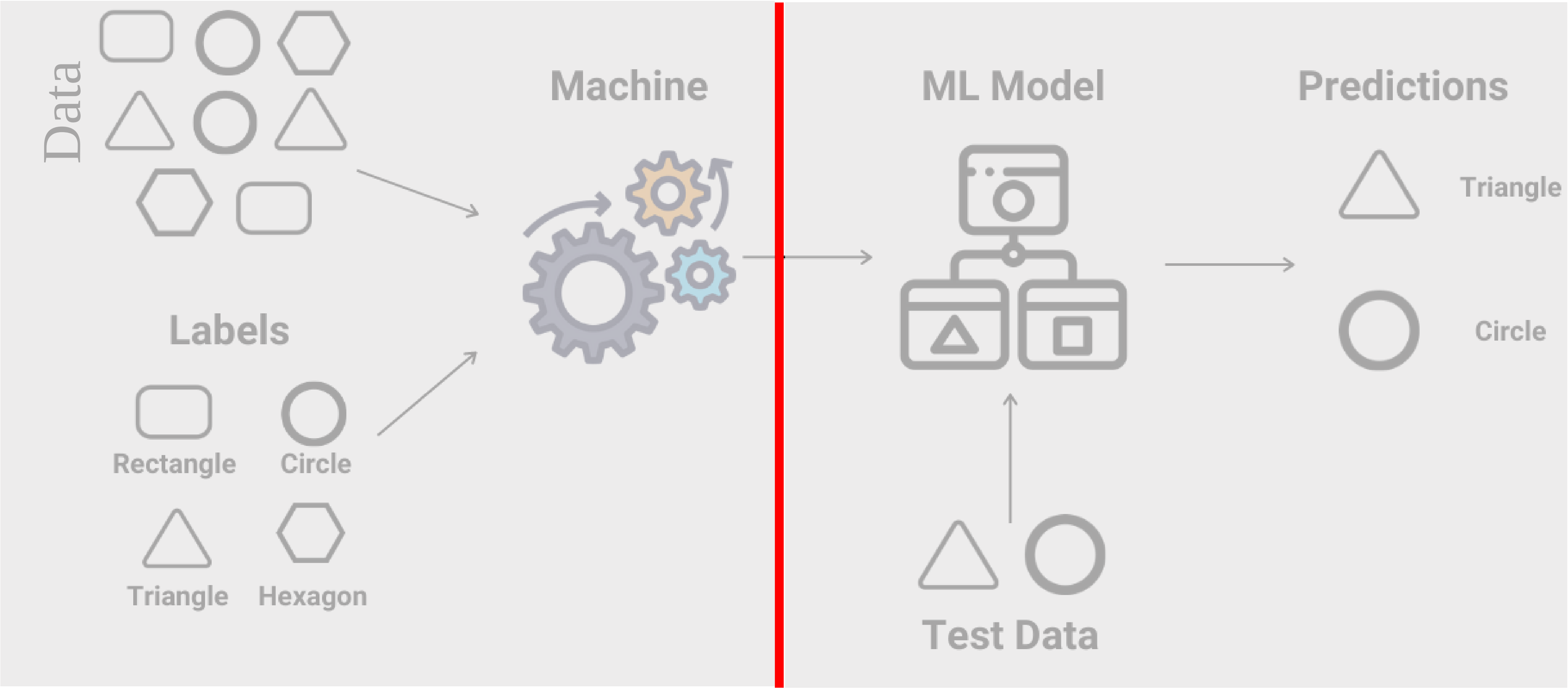
Q-learning,
Markov Decision Process

Gaussian Mixtures,
K-means, RNN,
Fuzzy *c*-means

Supervised Machine Learning

Image source: https://www.enjoyalgorithms.com/blog/classification-of-machine-learning-models

Supervised Machine Learning (cont’d)



Training

Testing (or *validation*)

Image source: https://www.enjoyalgorithms.com/blog/classification-of-machine-learning-models

Supervised Machine Learning (cont'd)

Learning Paradigms

Supervised Learning

- Learning with a teacher
- Data with known output (label) is given
- Classification and Regression

Support Vector Machine (SVM), K Nearest Neighbours (KNN), Decision Trees, Random Forest *Feedforward* Artificial Neural Network (ANN)

Reinforcement Learning

- Interactive learning environment by trial and error using feedback from its own actions and experiences.

Q-learning,
Markov Decision Process

Unsupervised Learning

- Learning without a teacher
- No labels
- Machine understand the data
- Clustering

Gaussian Mixtures,
K-means, RNN,
Fuzzy *c*-means

History of Neural Networks (NN)

- **1940**: McCulloch and Pitts: ***First*** mathematical model of a *neuron* (A verification model)
- **1957**: Rosenblatt's: The ***Perceptron*** model
- **1959**: Widrow and Hoff developed MADALINE was the first NN to be applied to a *real-world* problem

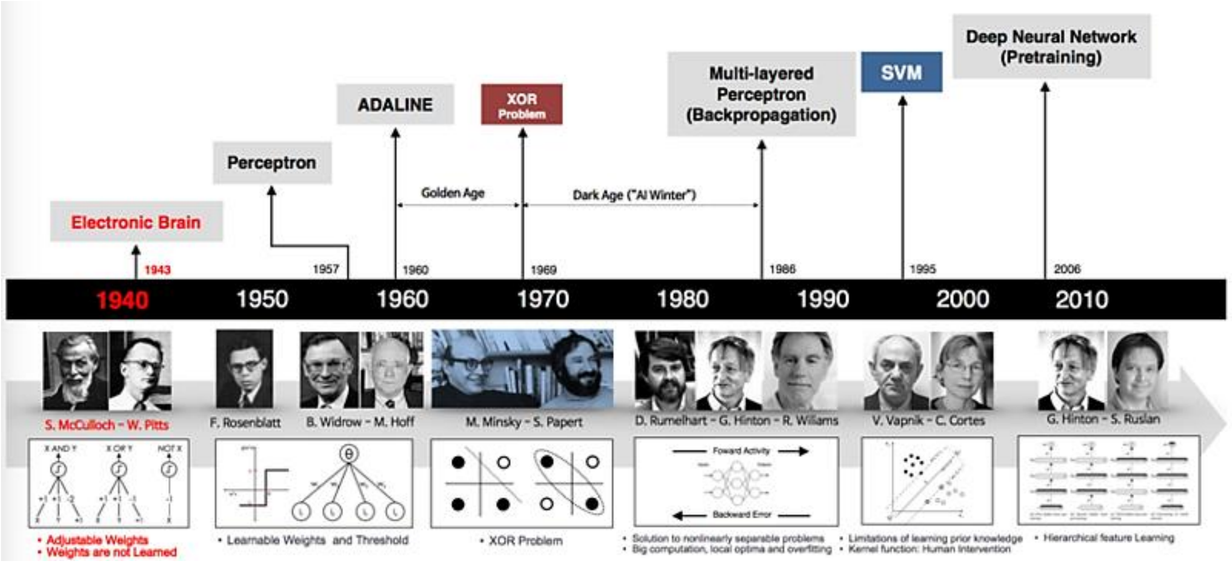


Image source: https://developpaper.com/take-you-into-the-past-life-and-this-life-of-neural-network/

Progress on NN research halted until 1981

- **1982**: Hopfield: Associative memory - Recurrent NN (or the RNNs)
- **1986**: Rumelhart: Backpropagation and the ***era*** of multilayer perceptron (MLP).
- **1990s**: Rise of support vector machine (SVM)
- **1997**: Schmidhuber & Hochreiter: An RNN, long short-term memory (LSTM) was proposed.
- **2006**: Hinton et al.: NN returned to the public's vision again though Deep belief nets (DBNs)
- **2016**: Boom of NN (Deep convolutional neural networks (CNNs): AlexNet, GoogLeNet, VGG, ResNet, etc.

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)

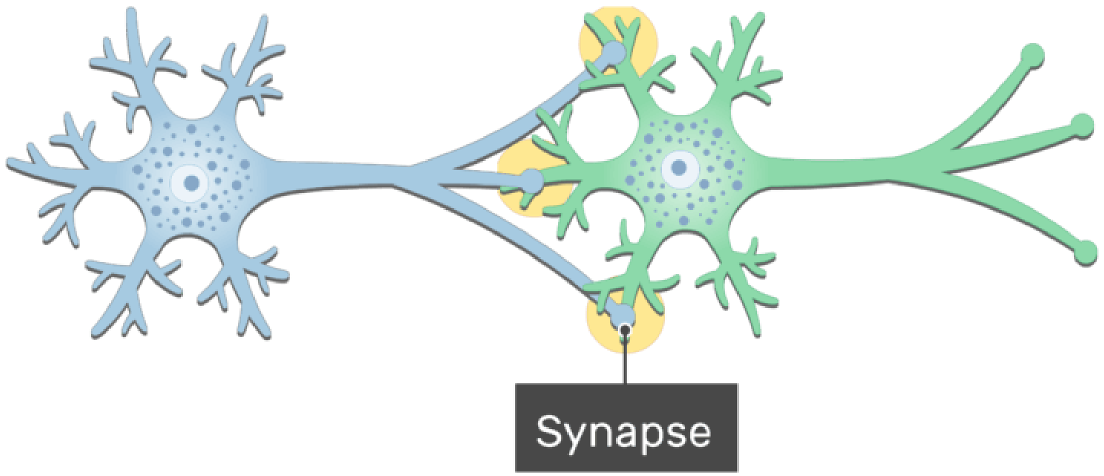
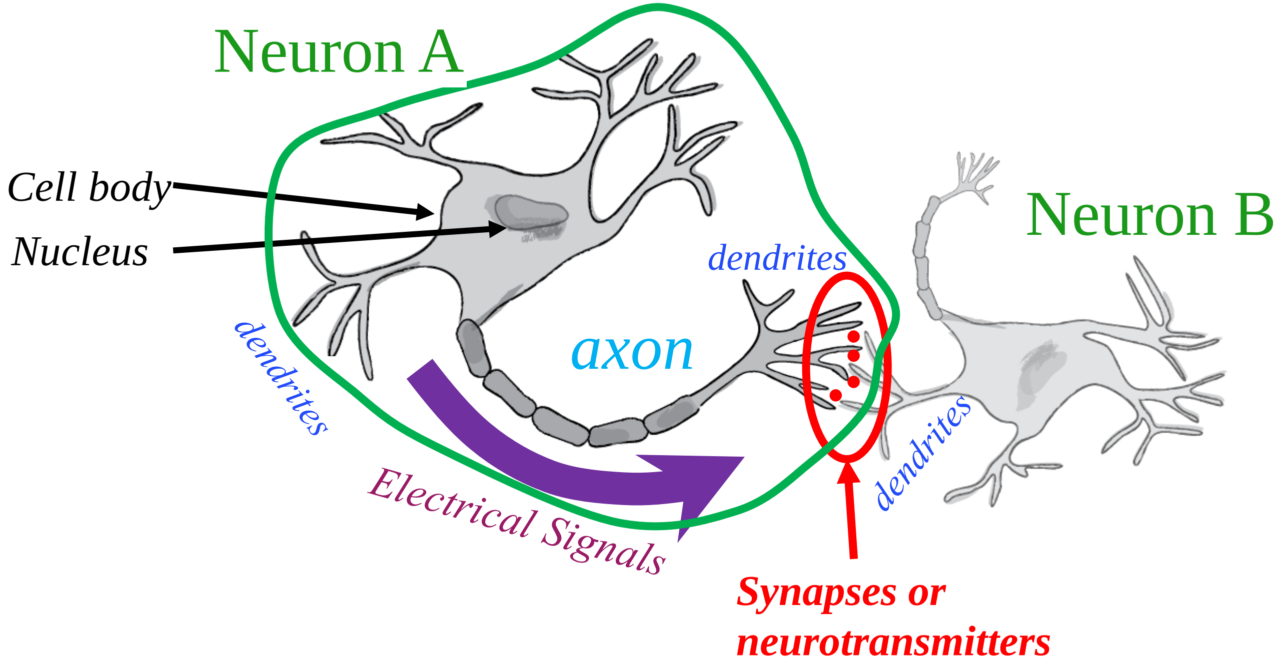


Image source: https://beautifulnow.is/discover/wellness/new-brain-flows-are-beautiful-now

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

Human Brain and Biological Neurons (cont'd)

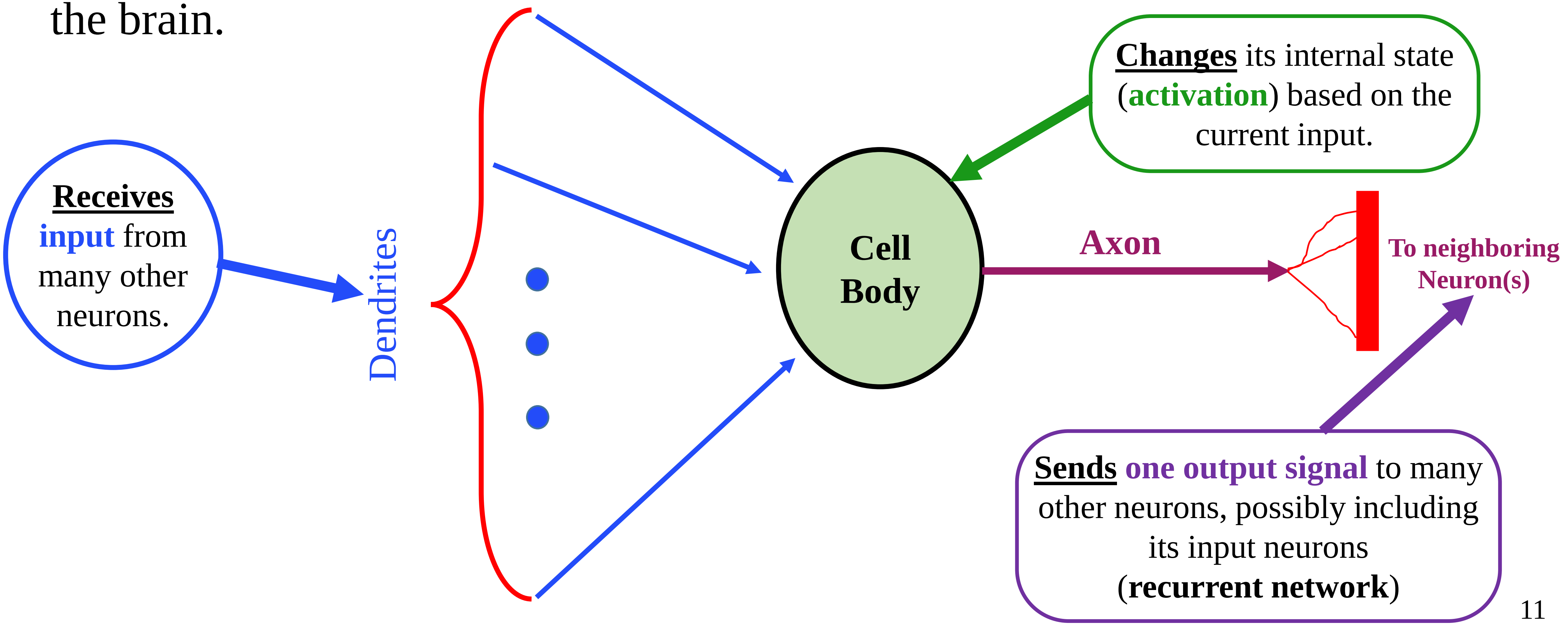
- A neuron accept (**and combine**) inputs through ***dendrites*** from other neurons
- If a given neuron *combined* input above a **<u>threshold</u>**, the neuron discharges a spike (**<u>electrical pulse</u>**) that travels from the body, down the **axon**, to the next **neuron(s)**
- The strength 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/

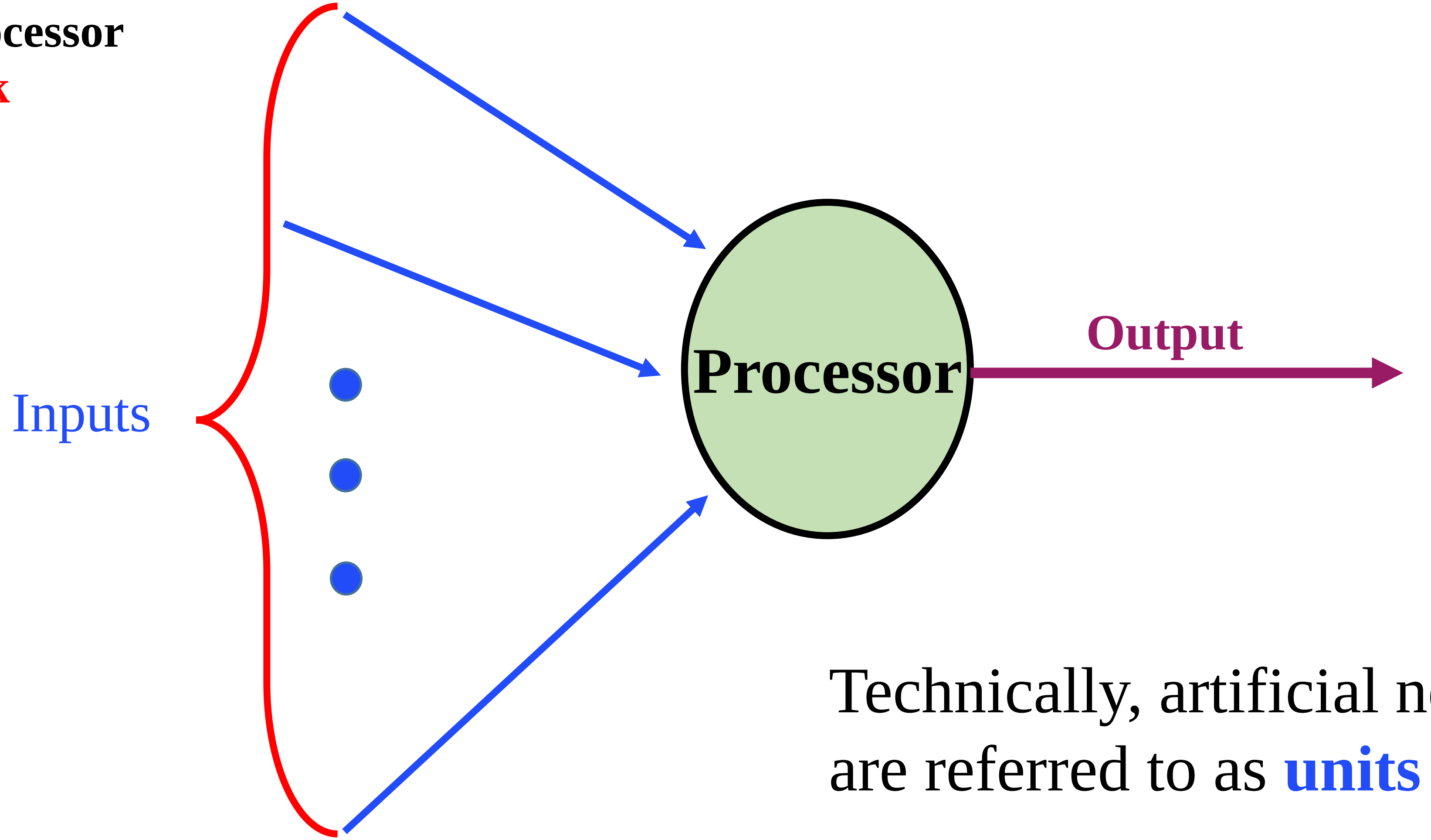
Modeling of a Biological Neuron

- 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

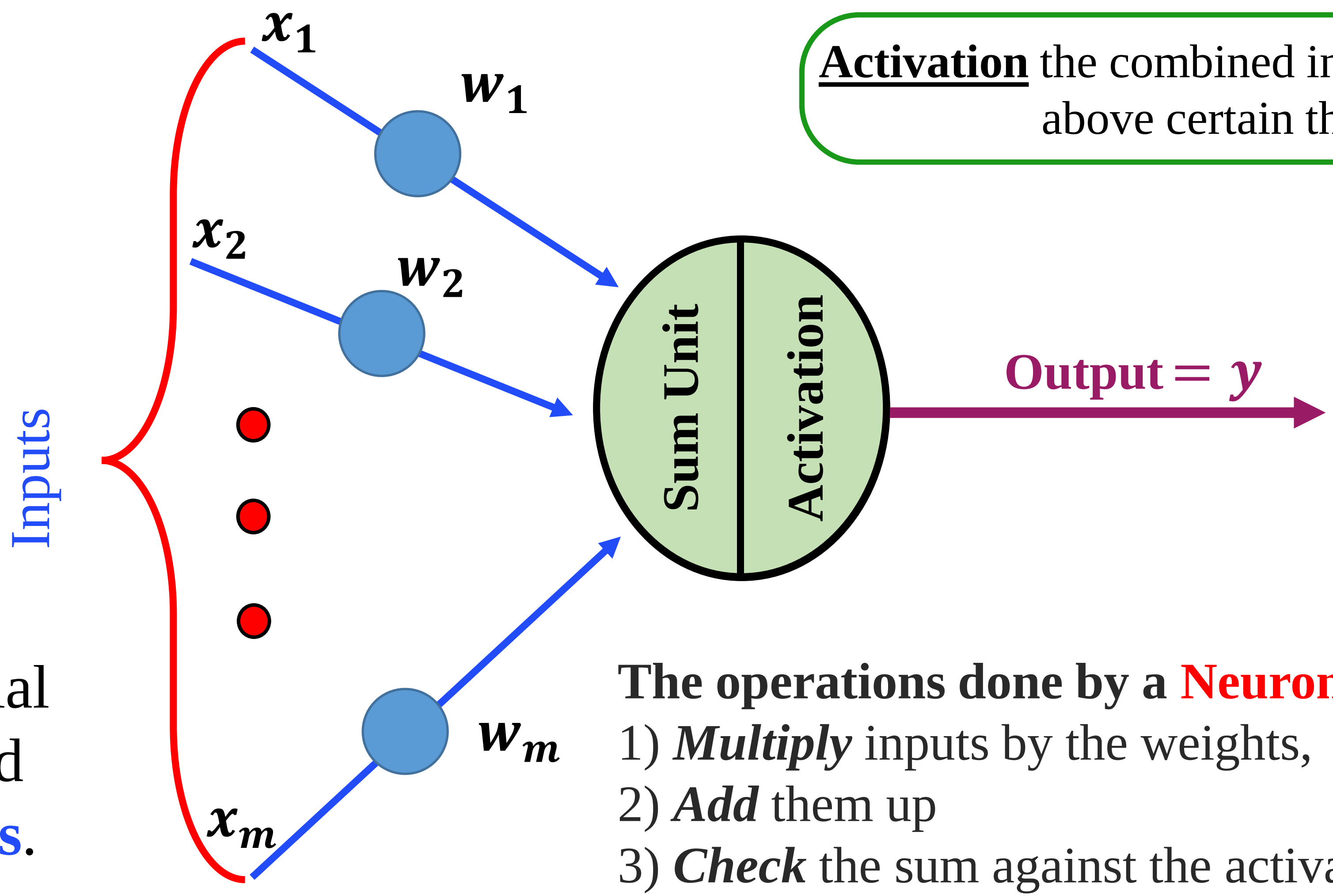
- An artificial neuron is an imitation of a human neuron
  - **Dendrites: Input**
  - **Cell body: Processor**
  - **Synaptic: Link**
  - **Axon: Output**



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**



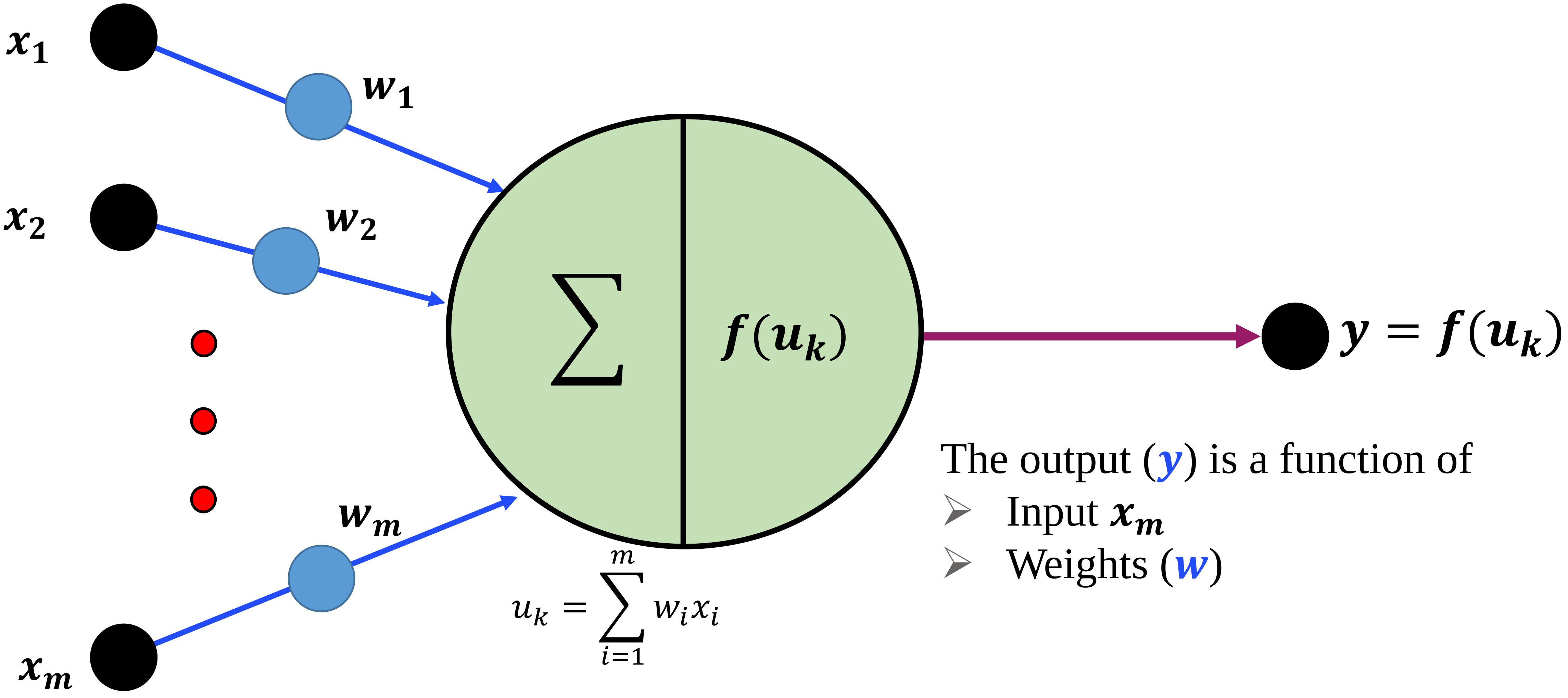
Activation the combined input must be above certain threshold

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

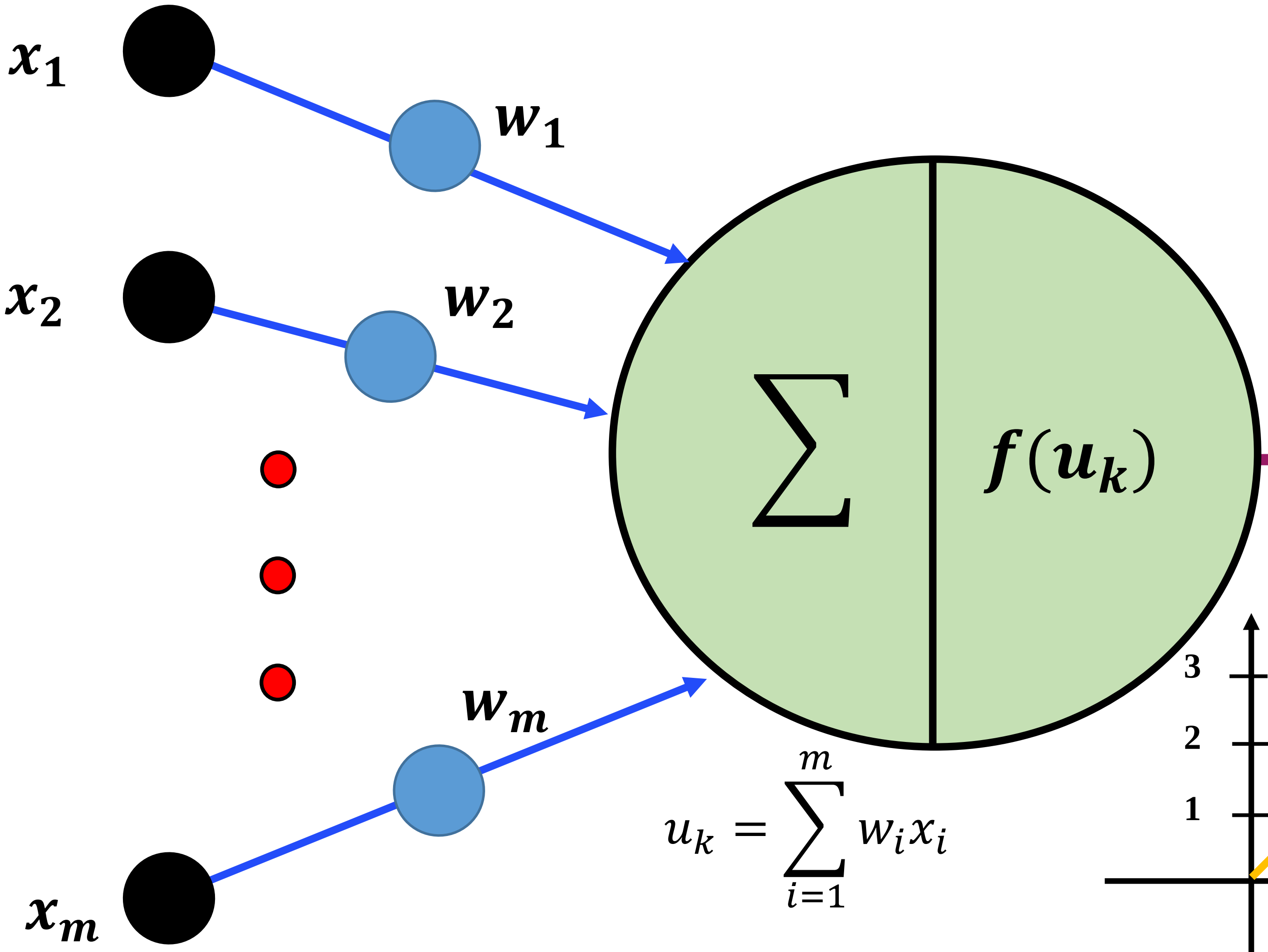
Artificial Neuron (cont'd)



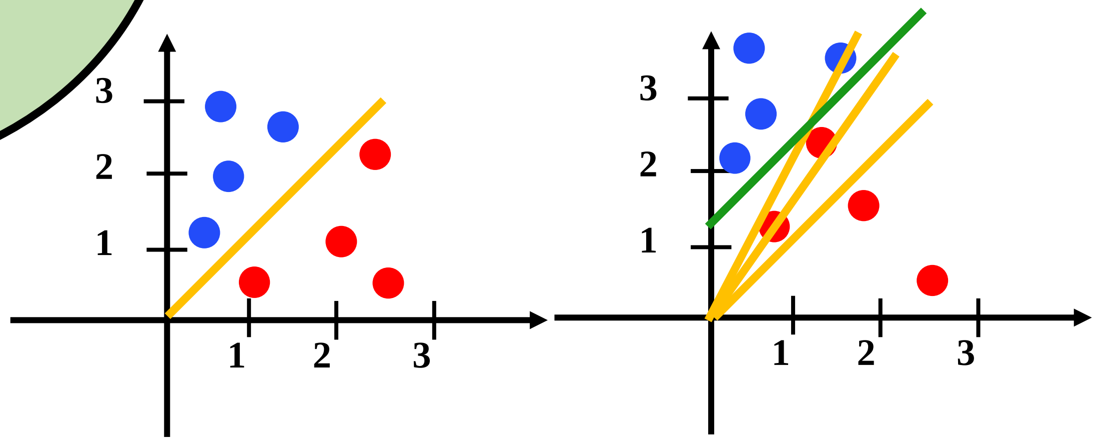
The output (y) is a function of

- Input x_m
- Weights (w)

Artificial Neuron (cont'd)

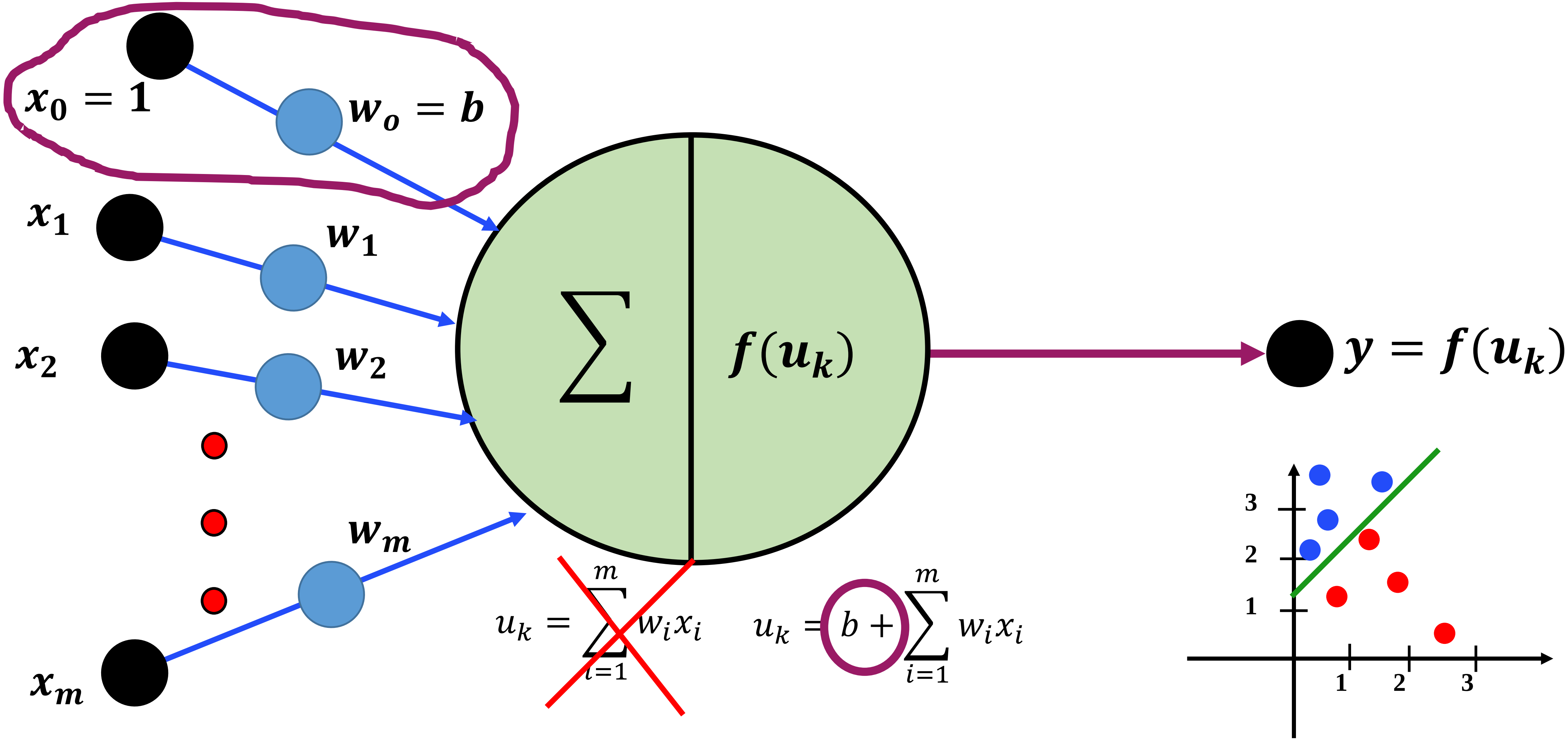


$f(.)$: How the combined xs and ws are used to produce y?



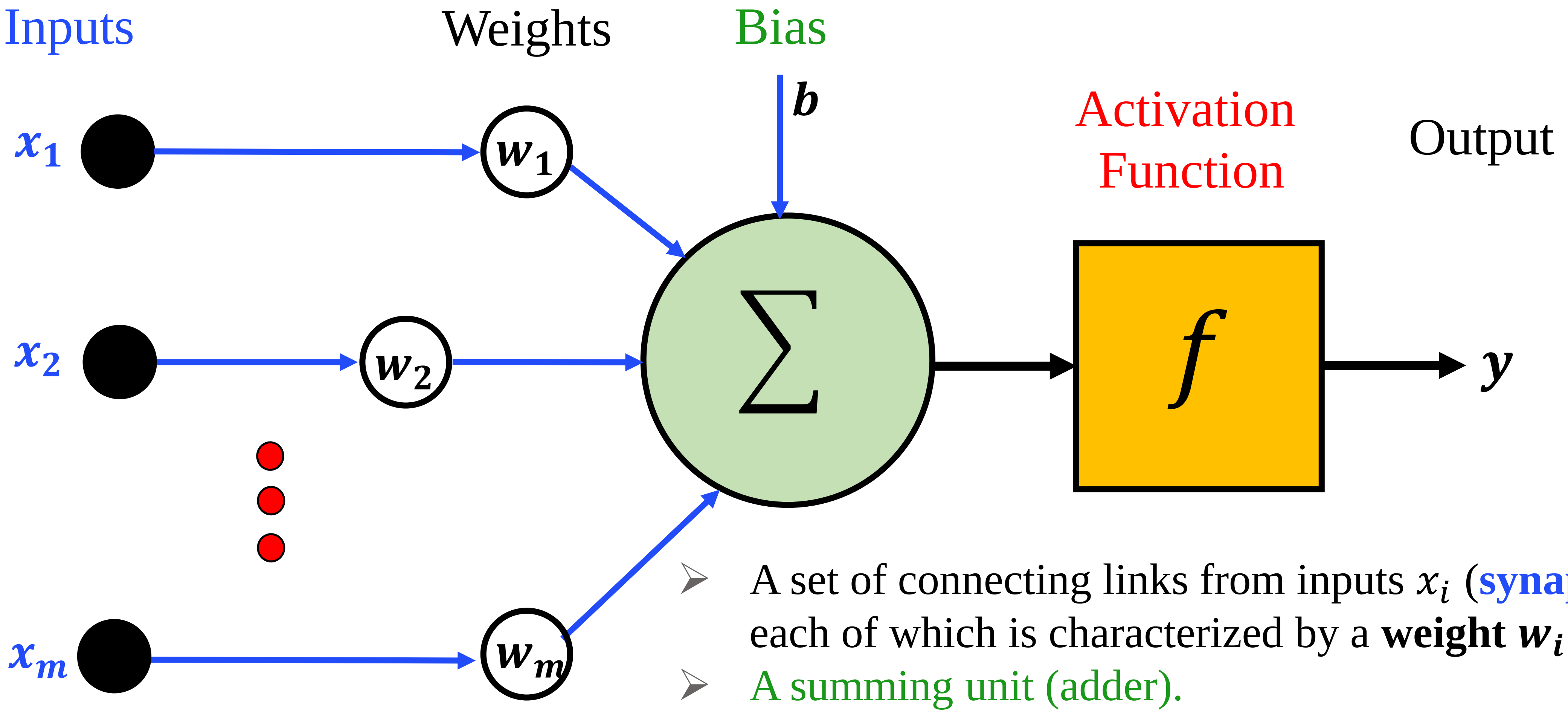
A **bias value (b)** is important to **full control** of the **activation function** (i.e., the output) for successful learning. This is a sort of **regularization**

Artificial Neuron (cont'd)



Artificial Neuron Network (ANN)

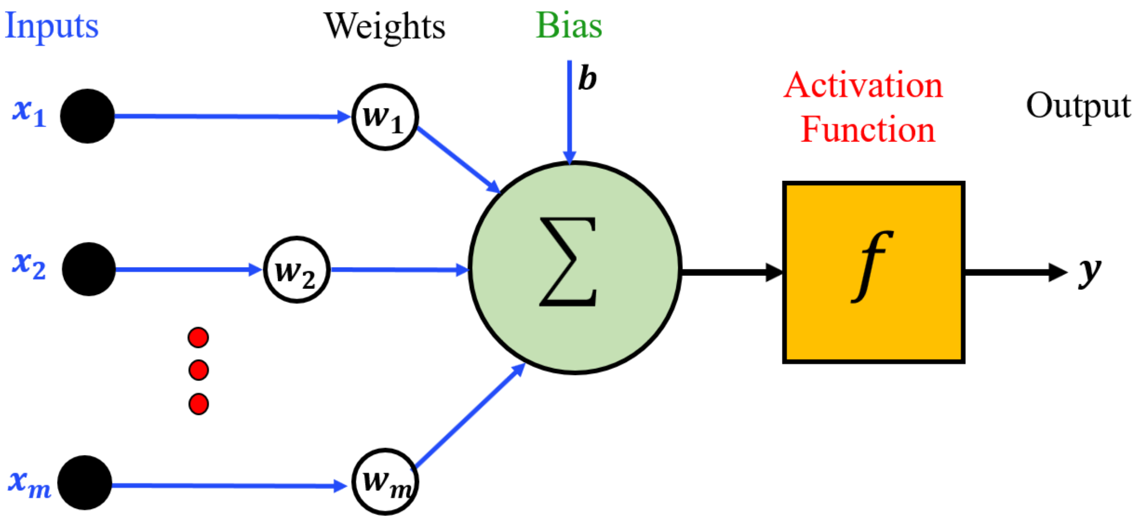
Basic Elements of any ANN:



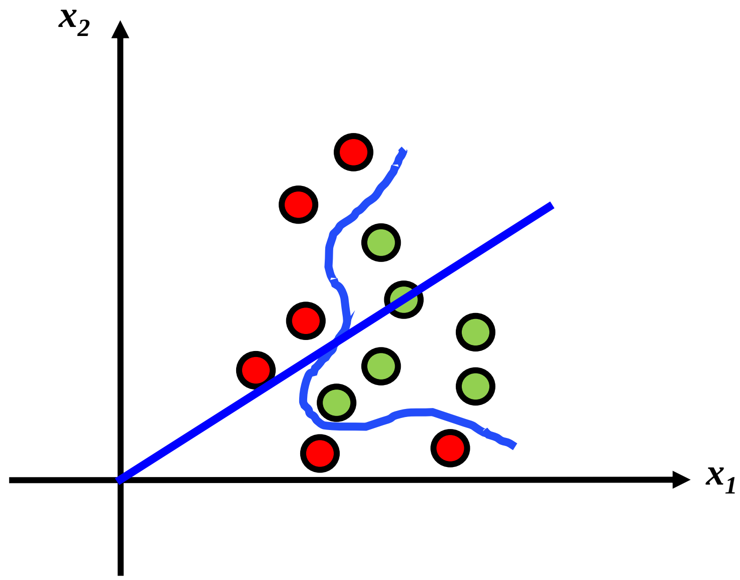
- A set of connecting links from inputs x_i (**synapses**) each of which is characterized by a **weight** $\boldsymbol{w_i}$.
- A summing unit (adder).
- An activation function (**nonlinearity**)

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?



$$y = f\left(b + \sum_{i=1}^{m} w_i x_i\right) \Rightarrow y = f(b + \mathbf{W}^{\mathrm{T}}\mathbf{X})$$

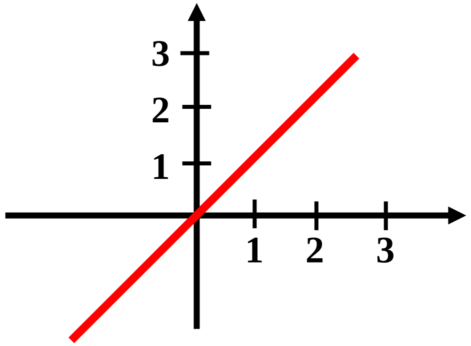
y is **linear** and **unbounded**

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

ANN Transfer Functions

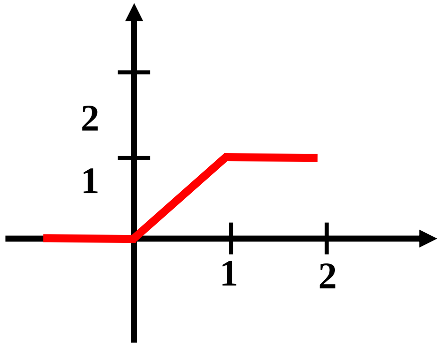
Linear

$$y_k = u_k$$



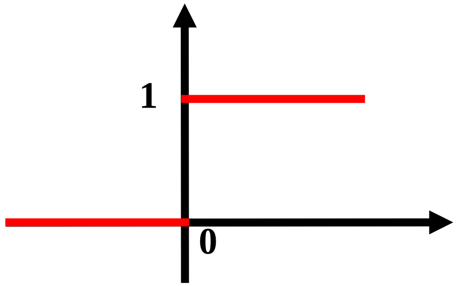
Saturating linear

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



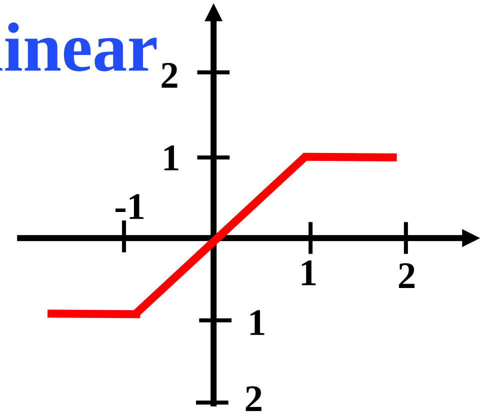
Hard Limit

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



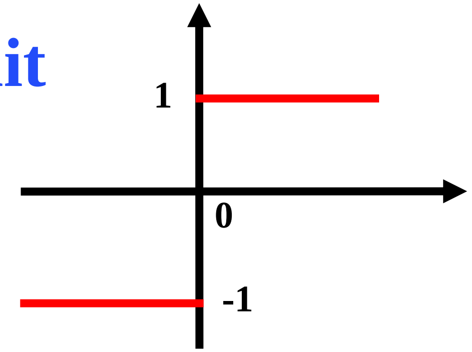
Symmetric Saturating linear

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



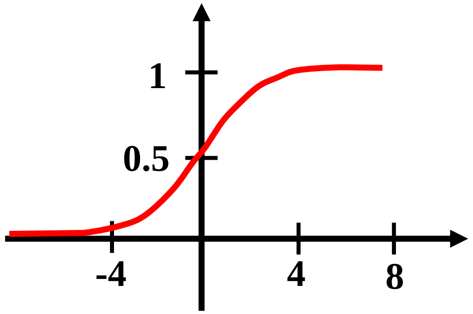
Symmetric Hard Limit

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



Log Sigmoid

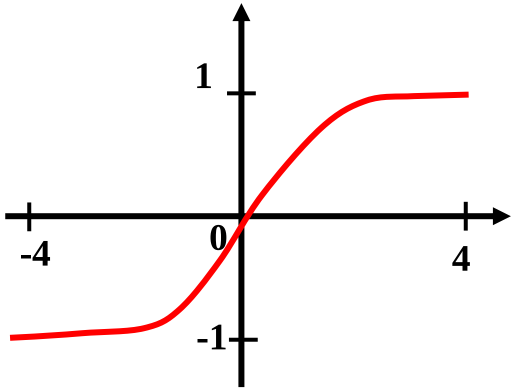
$$y_k = \frac{1}{1 + e^{-u_k}}$$



Artificial Neuron: Transfer Function

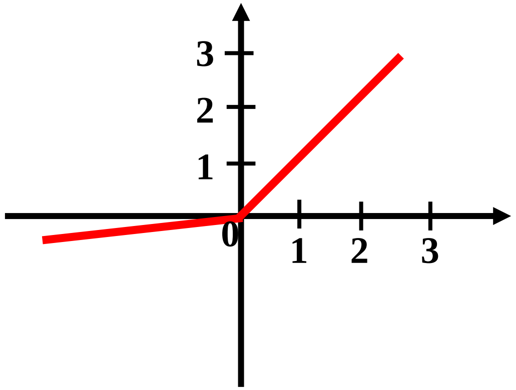
Hyperbolic Tangent Sigmoid

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



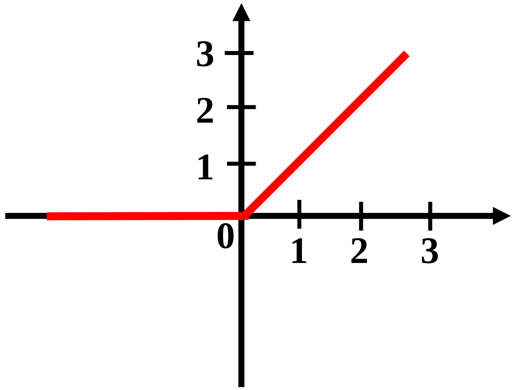
Leaky ReLU

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



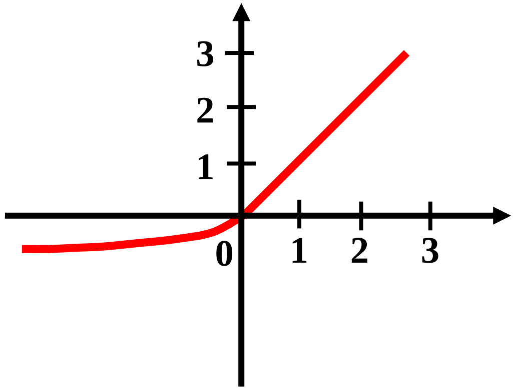
Rectified Linear Unit (ReLU)

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



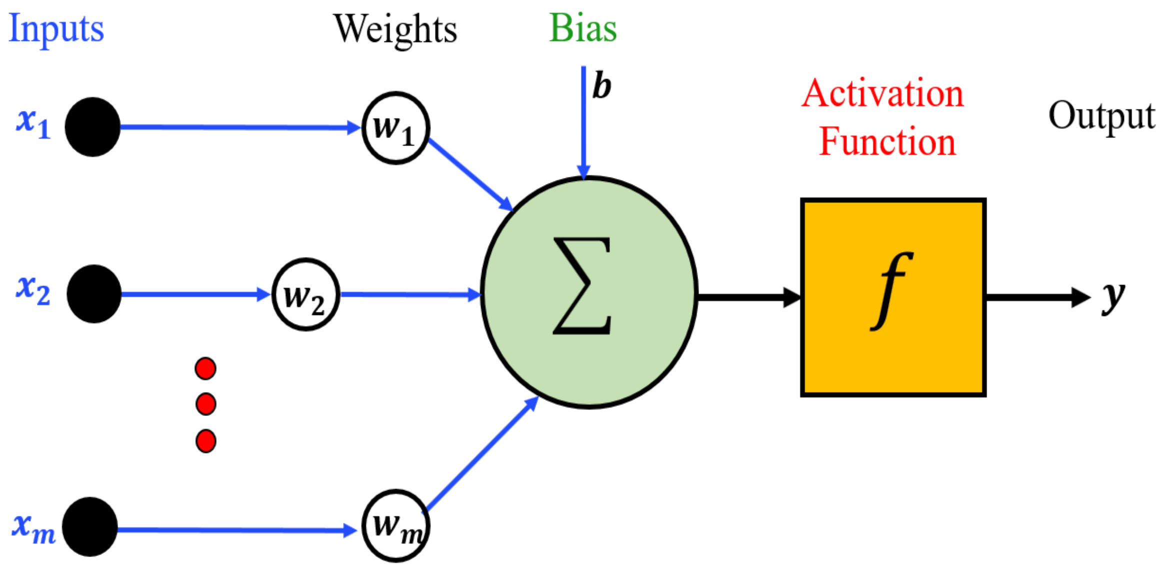
Exponential Linear Unit (ELU)

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



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*** process (**training**) 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

- **learning** is the process by which the *parameters* of an ANN, i.e., *w*, are **adapted** through a process of stimulation by the environment by which the network is embedded.

Learning ≡ Training

- ***Selection*** of the network topology
- ***Adapt*** weights values.
- ***Learn*** by trial-and-error (experience!)

X(n)
Sample Features

Sample number

d
Target

batch

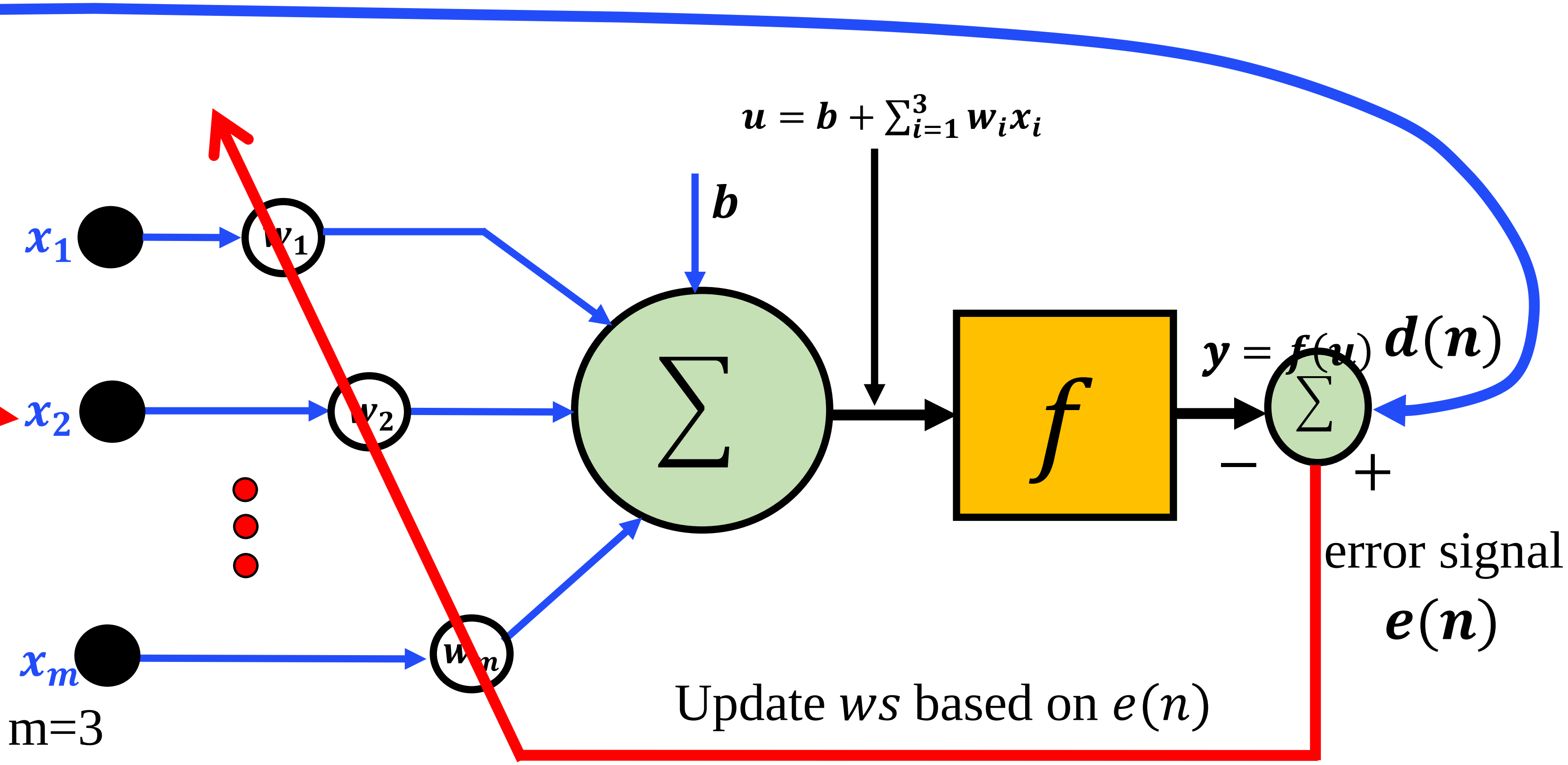
n	x_1	x_2	x_3	Output
1	10.33	56	0.56	0.8
2	8.97	48	0.61	0.1
3	11.01	49	0.49	0.3
4	9.32	53	0.89	0.7
5	10.51	50	0.71	0.4
6	12.10	59	0.90	0.8
⋮				
1996	7.99	61	0.59	0.9
1997	11.36	52	0.63	0.5
1998	12.09	48	0.78	0.2
1999	10.81	55	0.87	0.7
2000	13.00	53	0.91	0.6

- **Every data sample** for an ANN ***training*** consists of a vector **X(n)** and the corresponding (***desired*** or ***target***) output **d**

- A **batch** is a group of input samples with their *desired* outputs

Learning Process (cont'd)

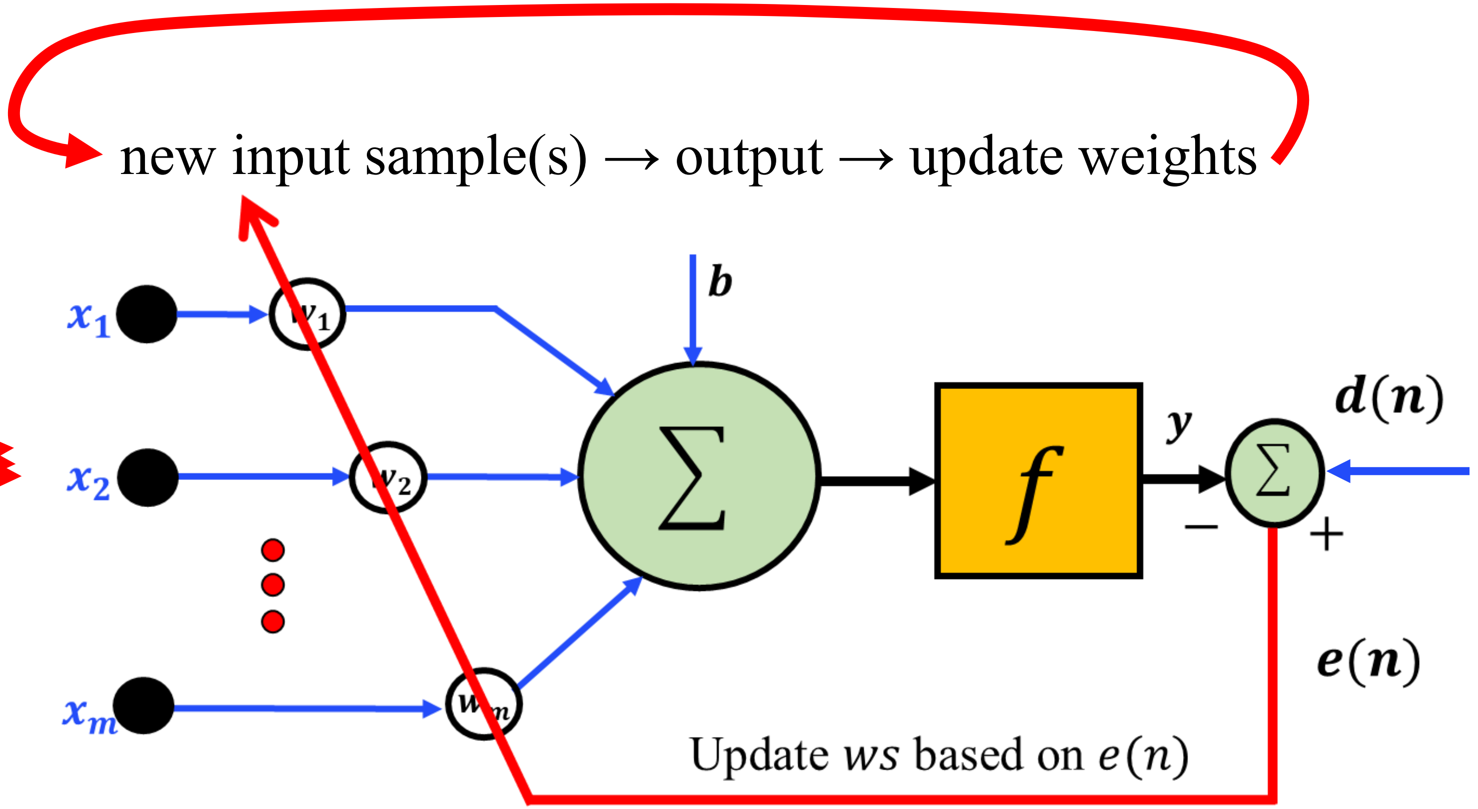
n	x_1	x_2	x_3	Output
1	10.33	56	0.56	0.8
2	8.97	48	0.61	0.1
3	11.01	49	0.49	0.3
4	9.32	53	0.89	0.7
5	10.51	50	0.71	0.4
6	12.10	59	0.90	0.8
⋮				
1996	7.99	61	0.59	0.9
1997	11.36	52	0.63	0.5
1998	12.09	48	0.78	0.2
1999	10.81	55	0.87	0.7
2000	13.00	53	0.91	0.6



Weight Adjustment= function ($error$,input)

Learning Process (cont'd)

n	x_1	x_2	x_3	Output
1	10.33	56	0.56	0.8
2	8.97	48	0.61	0.1
3	11.01	49	0.49	0.3
4	9.32	53	0.89	[illegible]
5	10.51	50	0.71	[illegible]
6	12.10	59	0.90	[illegible]
⋮				
1996	7.99	61	0.59	[illegible]
1997	11.36	52	0.63	[illegible]
1998	12.09	48	0.78	[illegible]
1999	10.81	55	0.87	0.7
2000	13.00	53	0.91	0.6



Weight Adjustment= function (*error*,input)

General rule for neuron learning

$$w_{new} = w_{old} + \eta * e * x$$

η is the learning constant or the ***learning rate***

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

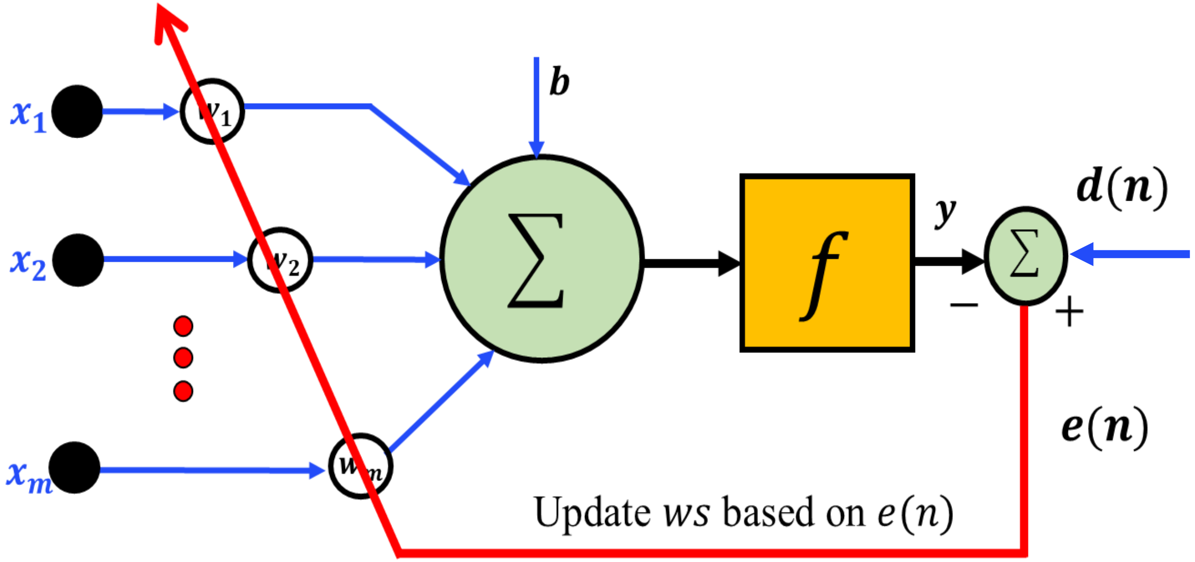
Compute the output function of a given input ($X(n)$)

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

Adjust the *weights*.

Loop until a criterion is met.

end

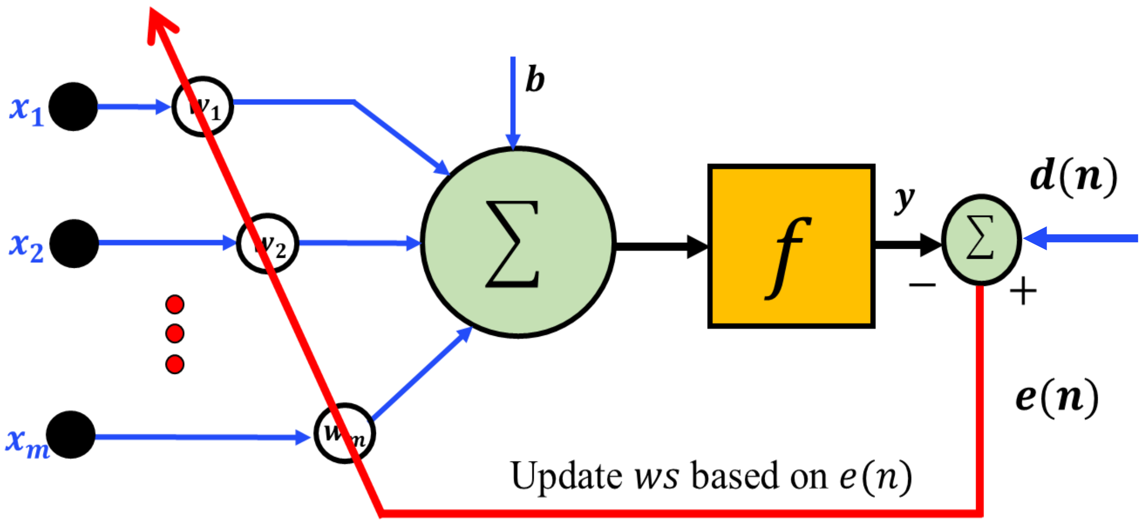


Criterion

- 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)



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

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

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

- This learning is called ***error-correction learning*** or ***delta*** rule or ***Widrow-Hoff*** rule

$$\Delta w_{kj}(n) = \eta . e_k(n) . x_j(n)$$

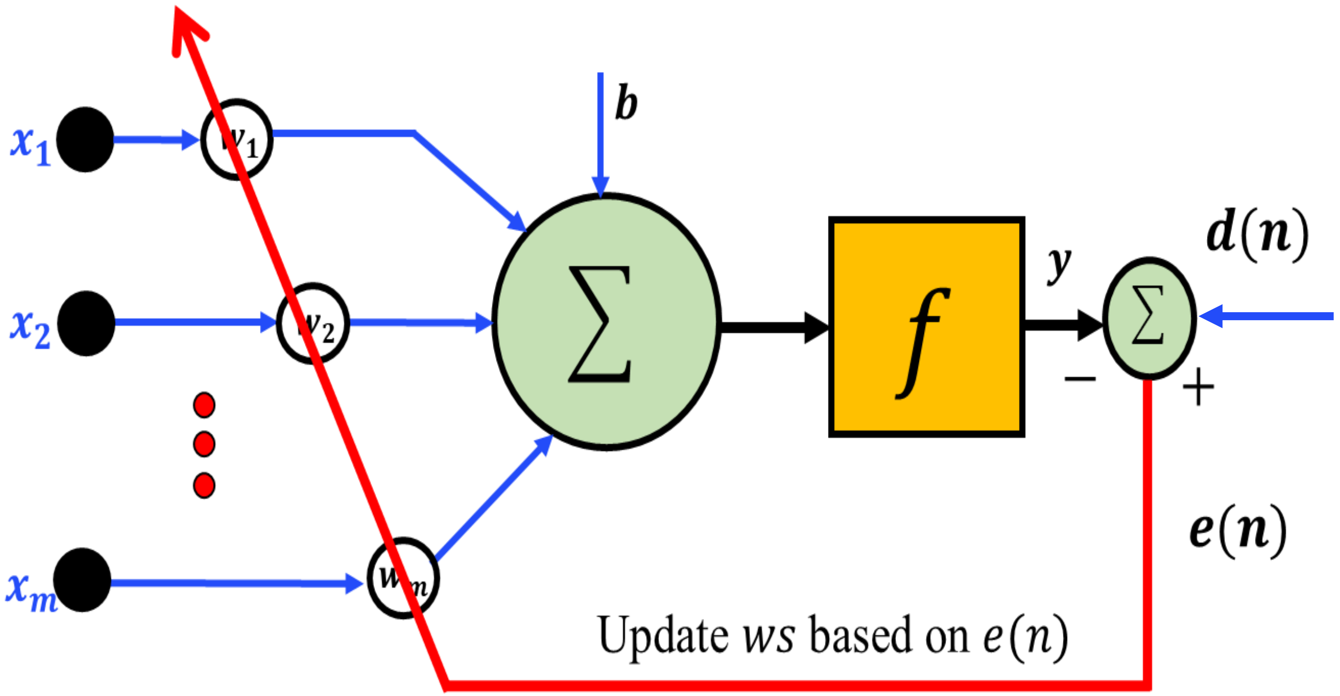
$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n)$$

n is the current sample
k index for the current neuron
j: $1 \rightarrow$ m

- The adjustment of a weight vector of **n input neuron connection** is proportional to the *product* of the **error signal** and the **input value** of the connection in question.

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***

Learning Process: Example

Example

n	x_1	x_2	x_3	d
1	1	1	0.5	0.7
2	-1	0.7	-0.5	0.2
3	0.3	0.3	-0.3	0.3

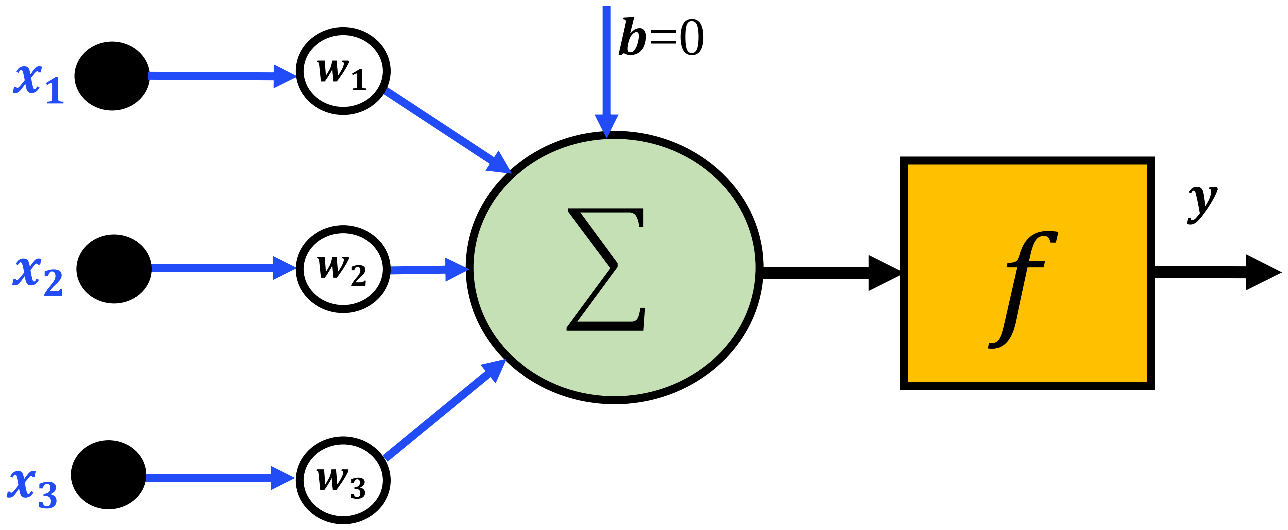
Assume

- initial weights are 0.5, -0.3, 0.8,
- b=0;
- η=0.1 and
- linear activation function

Learning Process Example: Solution

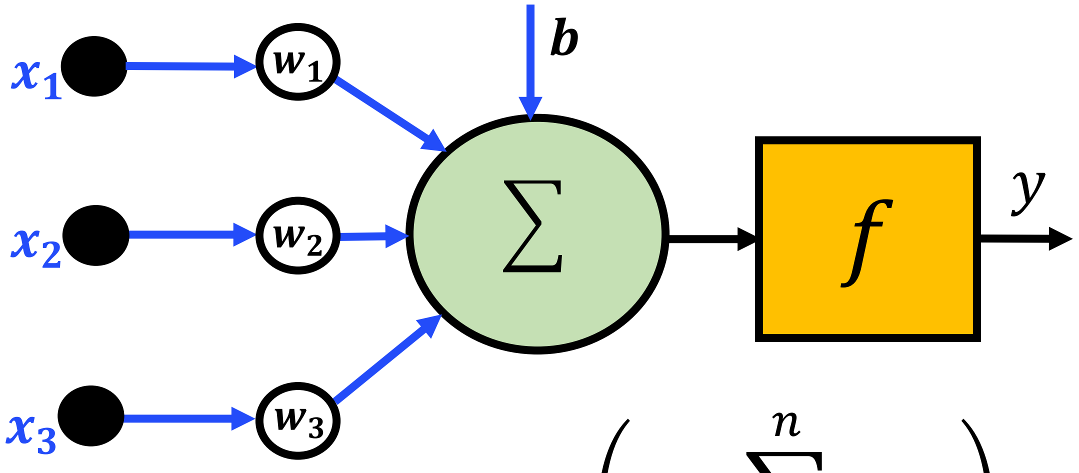
Solution

n	x_1	x_2	x_3	d
1	1	1	0.5	0.7
2	-1	0.7	-0.5	0.2
3	0.3	0.3	-0.3	0.3



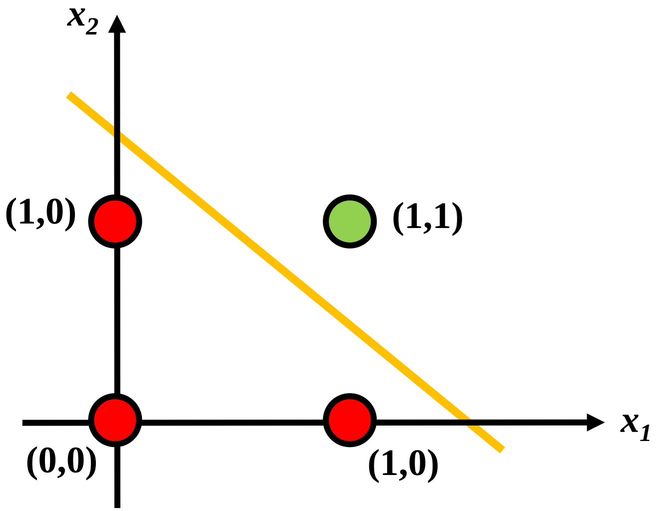
ANN Examples

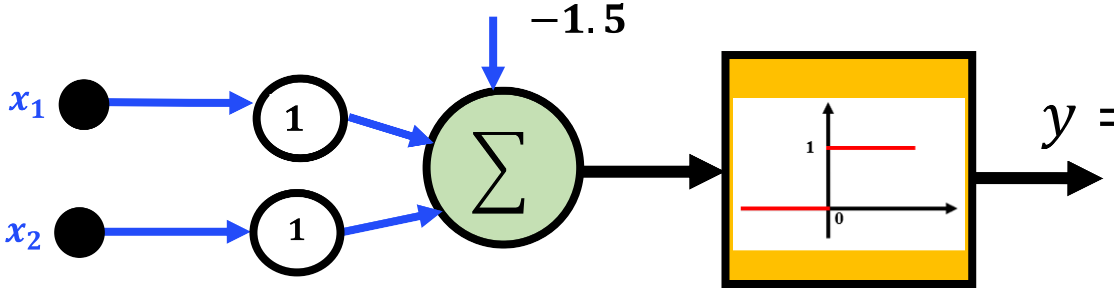
- One layer *feedforward* neural network called the *perceptron*
- Can solve linear function, e.g., AND, OR, NOT



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

$\mathbf{x_1}$	$\mathbf{x_2}$	**y**
0	0	**0**
1	0	**0**
0	1	**0**
1	1	**1**

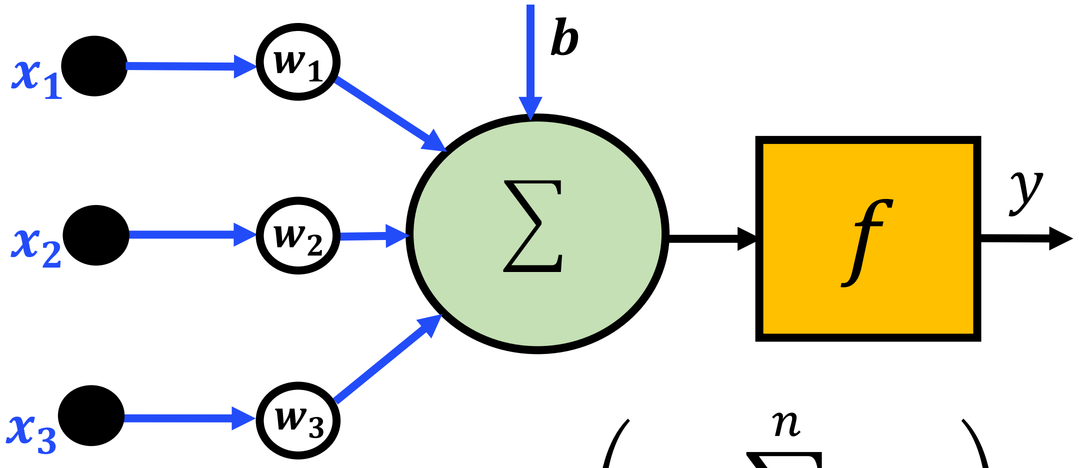




$$y = step(\underset{b}{-1.5} + \underset{w_1}{1}.x_1 + \underset{w_2}{1}.x_2)$$

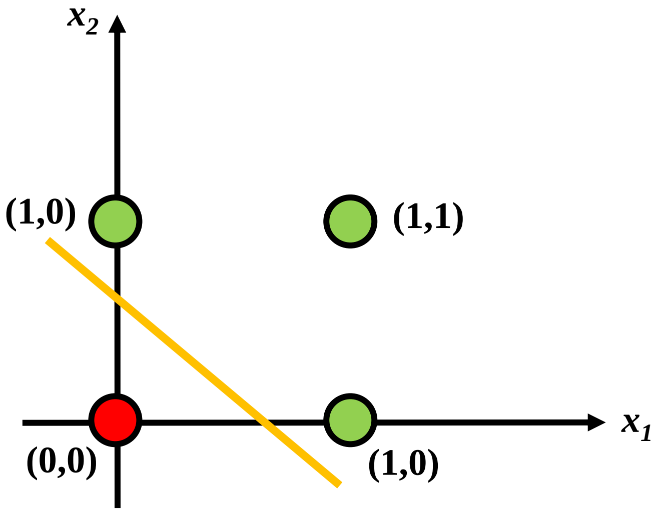
ANN Examples (cont’d)

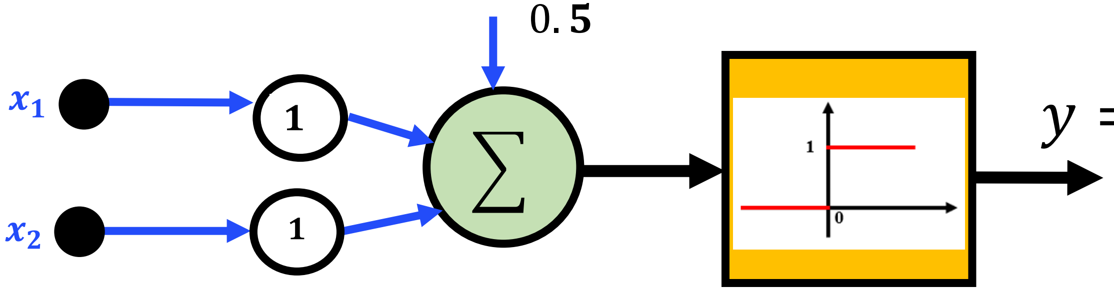
- One layer *feedforward* neural network called the *perceptron*
- Can solve linear function, e.g., AND, OR, NOT



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

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

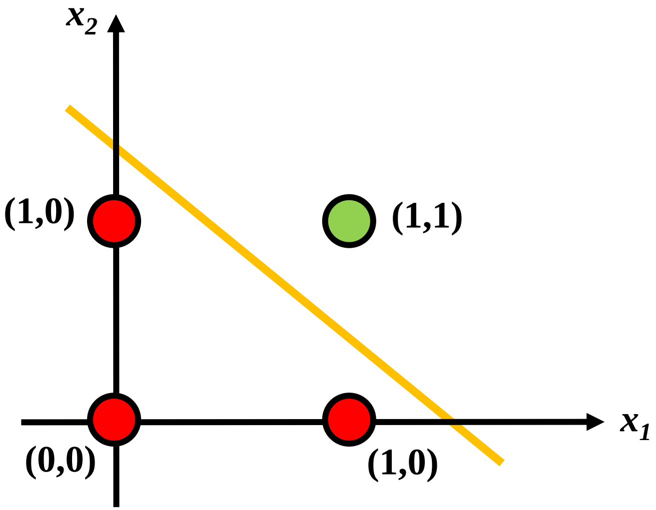




$$y = step(\underbrace{0.5}_{b} + \underbrace{1.}_{w_1}x_1 + \underbrace{1.}_{w_2}x_2)$$

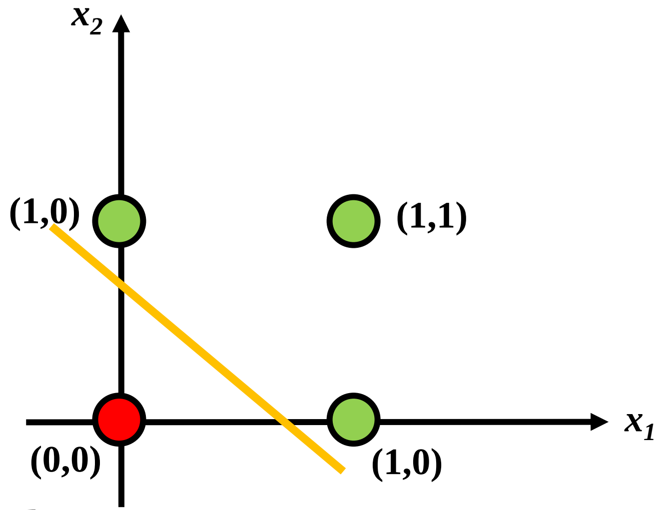
ANN Examples (cont'd)

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**

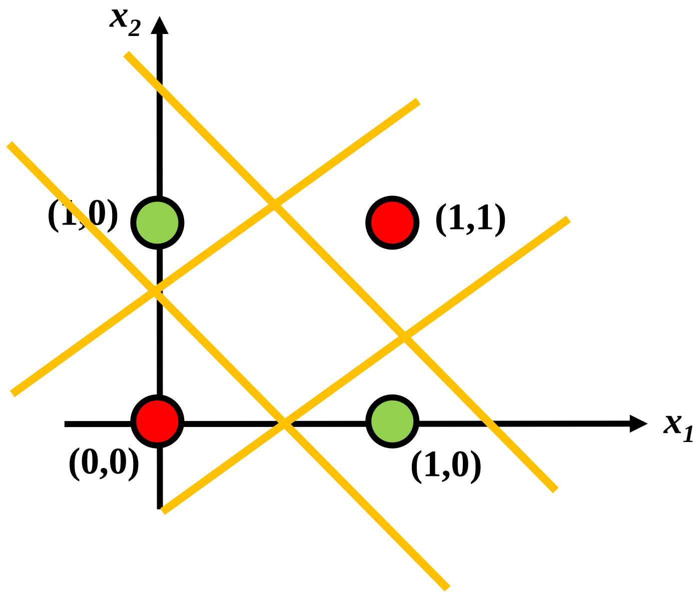


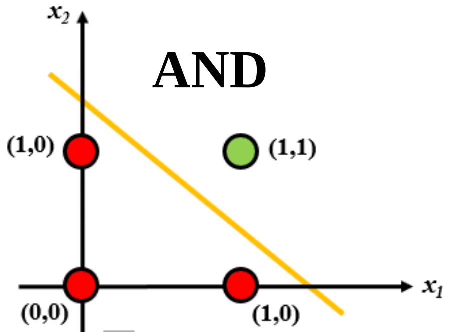
$$y = step(\overset{b}{0.5} + 1.x_1 + 1.x_2)$$

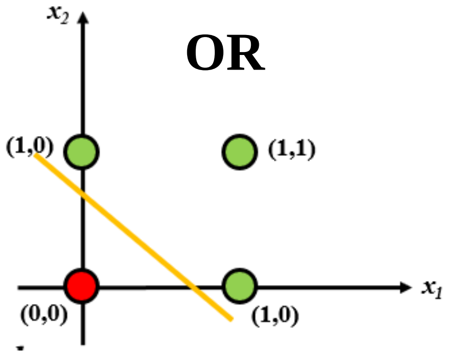
- Solving linearly, means the **decision boundary** is linear (straight line in 2D and a plane in 3D)
- The bias term (***b***) alters the **position**, but not the **orientation**, of the decision boundary
- The weights (w_1, w_2, ...w_m) determine the gradient

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 **one neuron** defines **one line** through input space, **what do we need to have two lines?**
- We need to have two neurons working in ***parallel*** (*next to each other rather than in different layers*).
- We would need a **multilayer neural network** to model (or to separate the two classes using) the XOR function.

Multilayer Perceptron (MLP)

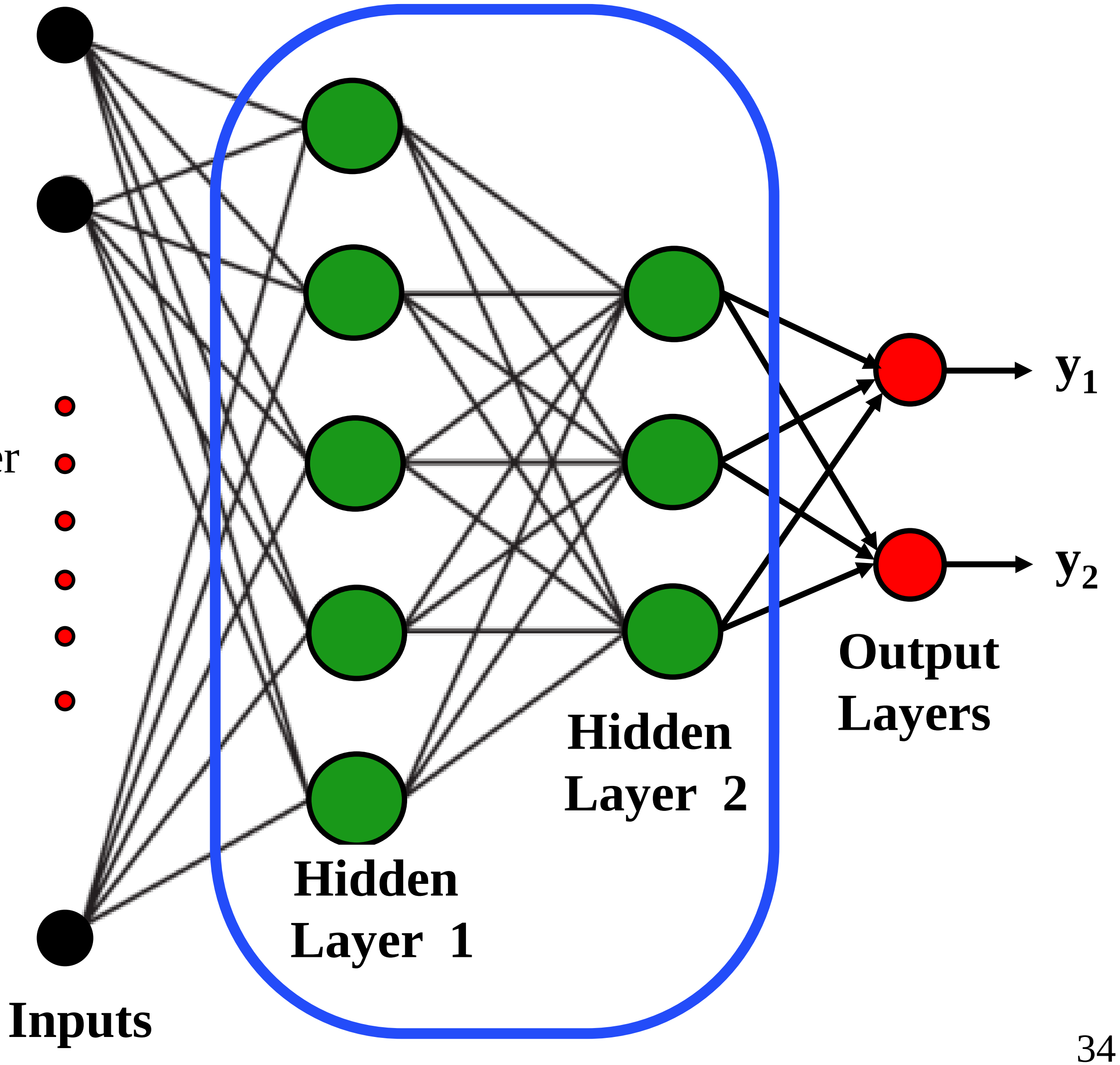
- More layers between the ***input*** the ***output*** layers
- Fully connected layers
- Multiple neurons at the output layers

  $\boldsymbol{y_j}$, **j** $\in$ $\boldsymbol{C}$ 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

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

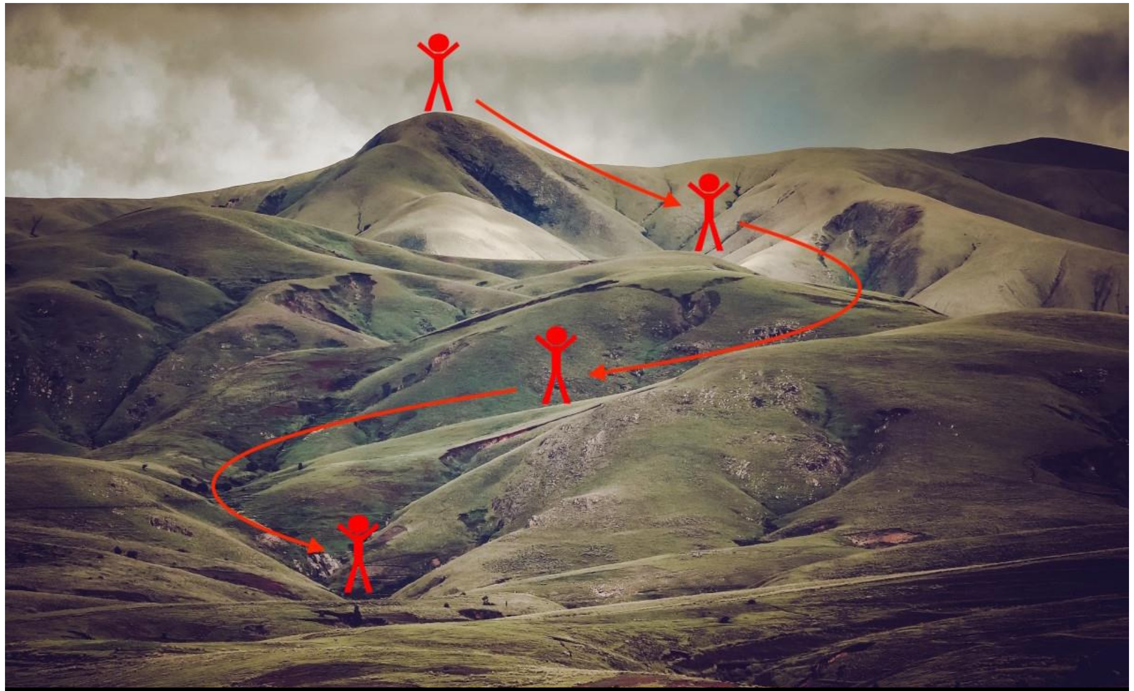
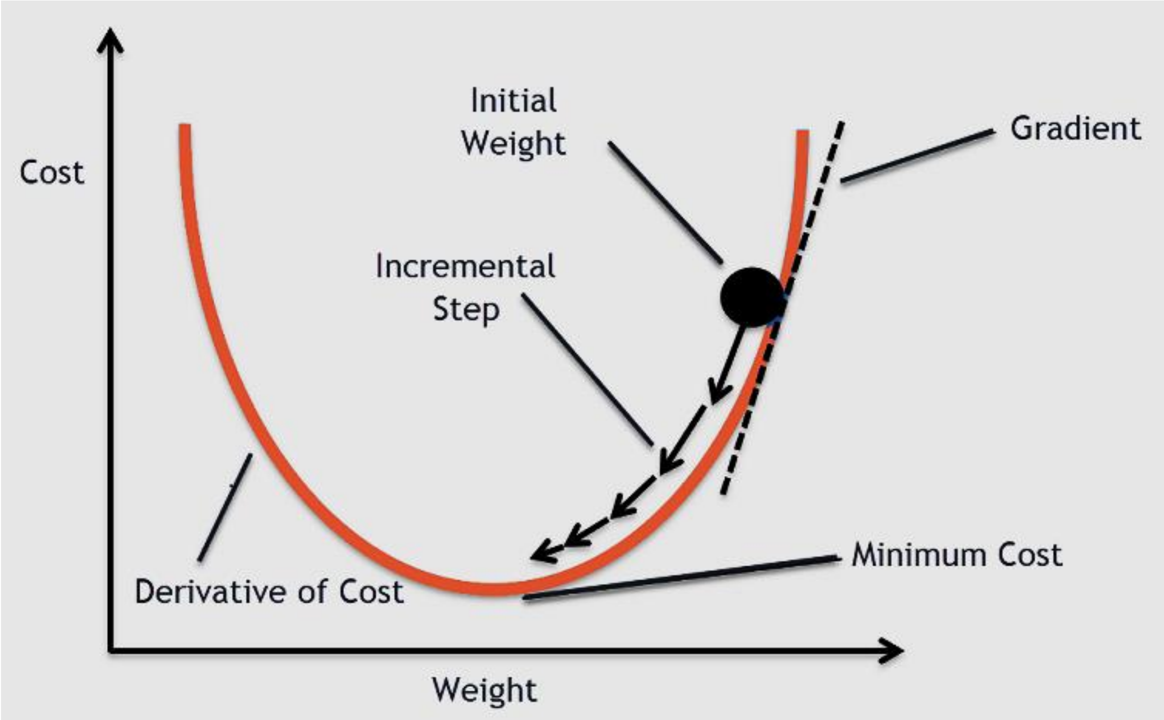


Image Source: https://datascience-enthusiast.com/figures/cost.jpg



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

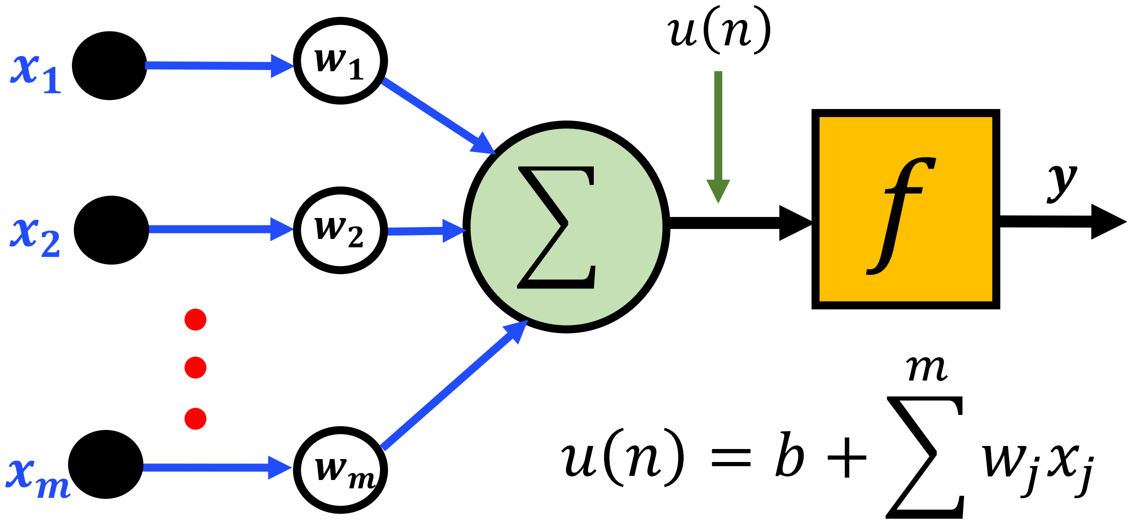
Gradient Descent (cont'd)

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

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

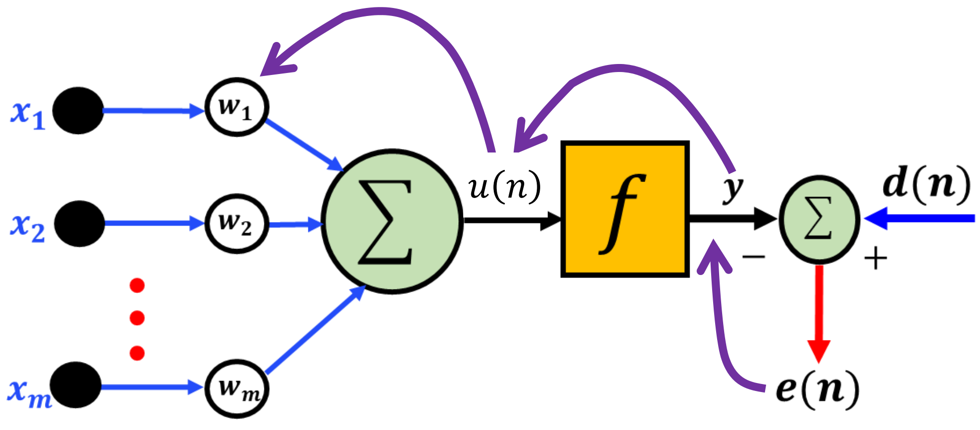
For **any activation** function f:

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



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

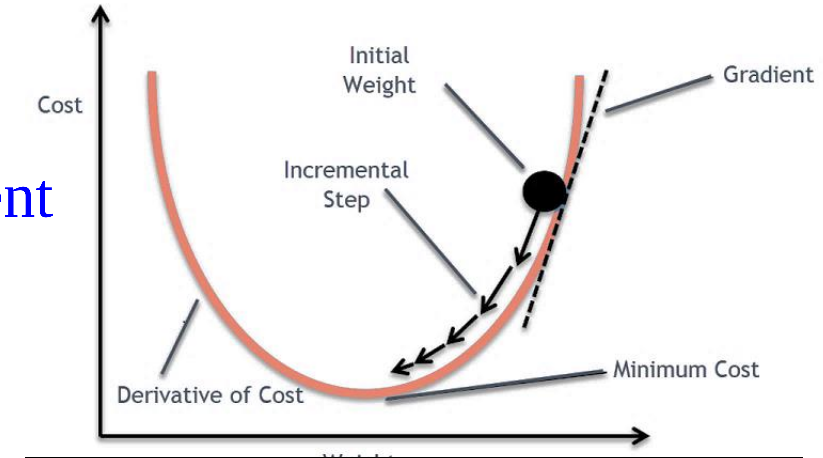
Gradient Descent (cont’d)



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

minimization

gradient



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

By applying the chain rule

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

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

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

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

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

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

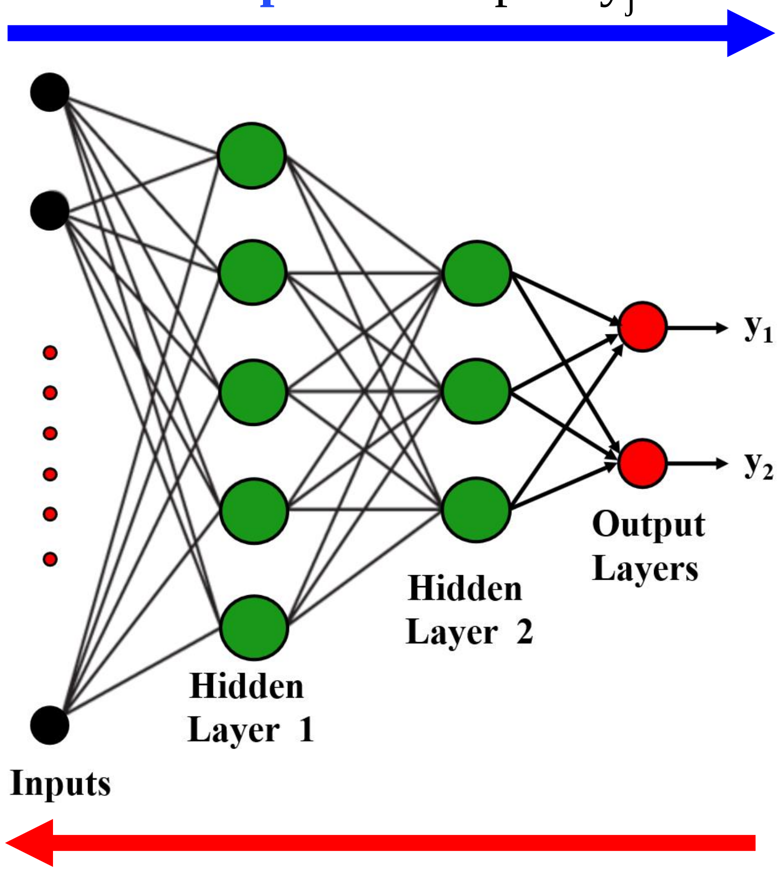
$$u(n) = \sum_{j=1}^{m} w_j x_j \Rightarrow \frac{\partial u}{\partial w_j} = x_j$$

Backpropagation

- Backpropagation is *supervised* algorithm that is a generalization for the least mean square (LMS) algorithm
- It is based on the ***gradient search*** technique to minimize the **cost function** ≡ 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/

propagate activation from input to output ≡ compute y_j



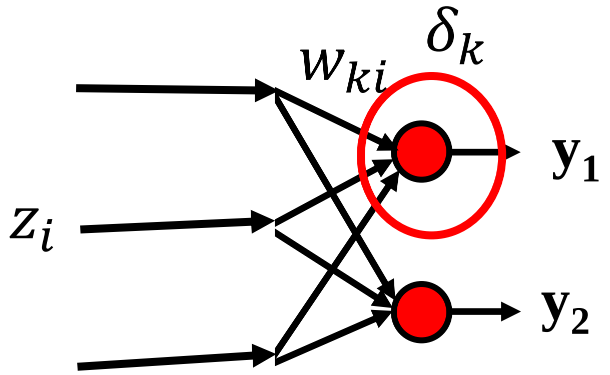
propagate error from output to hidden layers ≡ adjust all weights

Backpropagation (cont'd)

- The weights of each **output neuron** can be determined directly using the *delta* learning rule.

$$\Delta w_{ki} = \eta * \boxed{e * f'(\cdot)} * z_i \qquad \delta_k = e * f'(\cdot)$$

local gradient or error signal

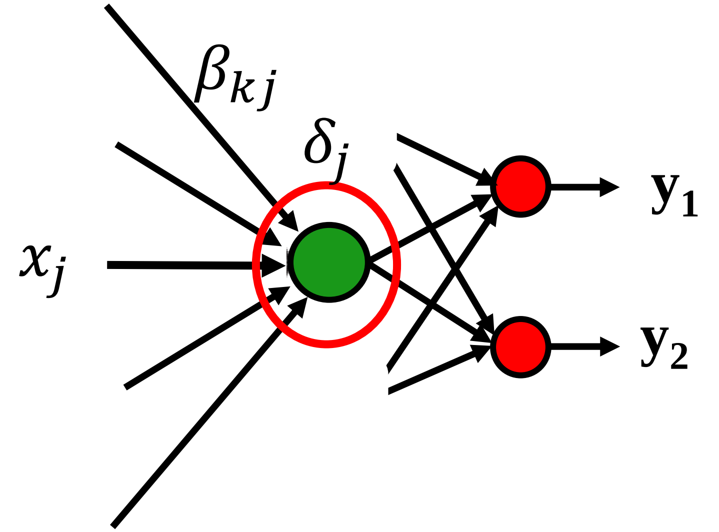


Backpropagation (cont'd)

- The weights of each **output neuron** can be determined directly using the *delta* learning rule.

$$\Delta w_{ki} = \eta * \boxed{e * f'(\cdot)} * z_i \qquad \delta_k = e * f'(\cdot)$$

local gradient or error signal



- If the neuron is a *hidden* node

$$\delta_j = f'(\cdot) * \sum_{k=1}^{K} \delta_k * w_k$$

K is the set of all <u>nodes</u> on a ***next layer*** connected to the **current neuron**

[local gradient] x [upstream gradient]

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

Backpropagation Example

- Assume **one** input layer, **one** hidden layer, and **one** output neuron

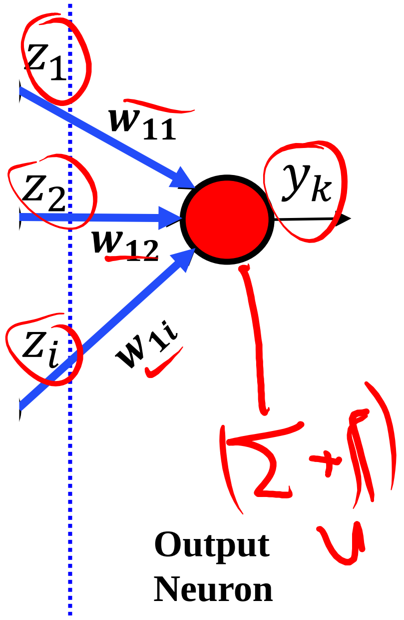
x_j :is the j^{th} **input**

z_i :is the output of the i^{th} **hidden neuron**

y_k :is the output of the k^{th} **output neuron**

β_{ij} :is the weight from input node x_j to hidden node z_i

w_{ki} :is the weight from **hidden node z_i** to **output neuron y_k**



- The weights of the output neuron can be adjusted using the ***delta learning*** rule and the error signal:

$$\delta_{y_k} = e_k * f'(u_k) = (d_k - y_k)\, f'(u_k) \qquad u_k = \sum_{i=1}^{I} w_{ki} z_i$$

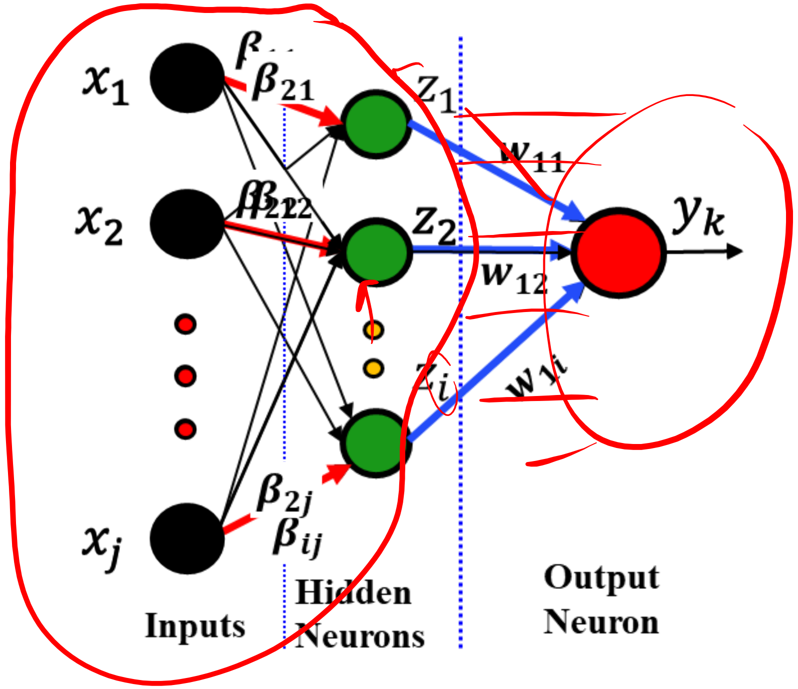
- Update the weights as follows:

$$w_{ki}(n+1) = w_{ki}(n) + \eta * \delta_{yk} * z_i$$

Backpropagation (cont'd)

- The weights of a i^{th} hidden neuron can be adjusted using its error signal:

$$\delta_{z_i} = f'(u_i) * \sum_{k=1}^{K} \delta_{y_k} * w_{ki} \qquad u_i = \sum_{j=1}^{J} \beta_{ij} x_j$$



- Using the error signals, the weights of the i^{th} hidden neuron can be updated

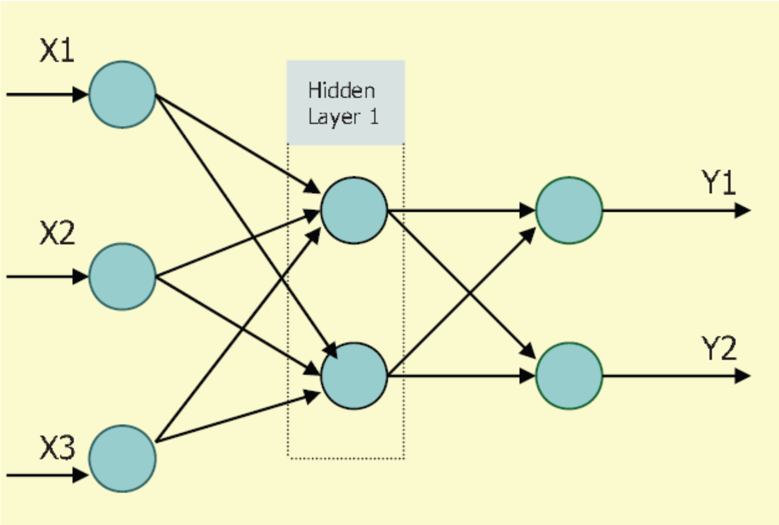
$$\beta_{ij}(n+1) = \beta_{ij}(n) + \eta * \delta_{zi} * x_j$$

- For a sigmoid activation with zero bias

$$f'(u_k) = y_k(1 - y_k)$$

Types of Neural Networks

Feedforward neural network

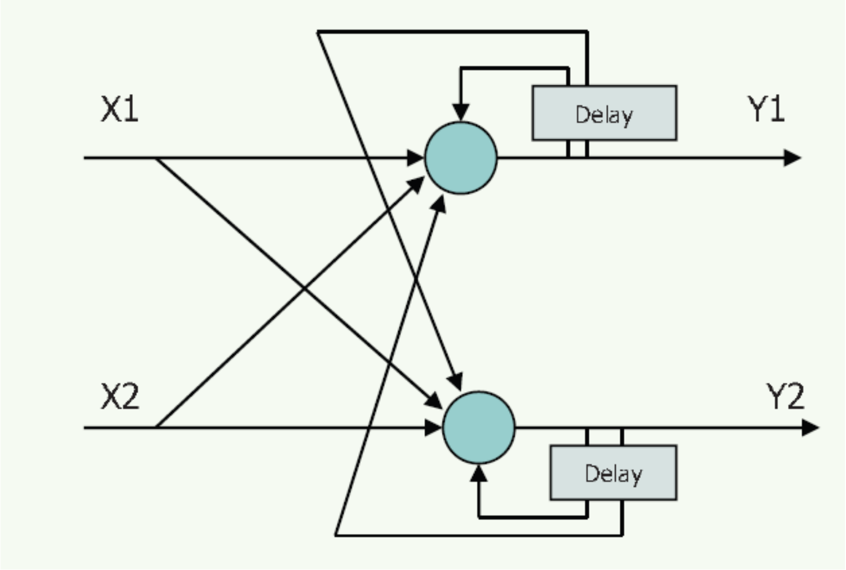


Signals to travel one way only (input to output)

Learning **with** a teacher

Supervised Learning

Recurrent neural network (RNN)



Output from previous step is fed as input in the current step

Learning **without** a teacher

Unsupervised Learning

Self-organizing maps (SOM)

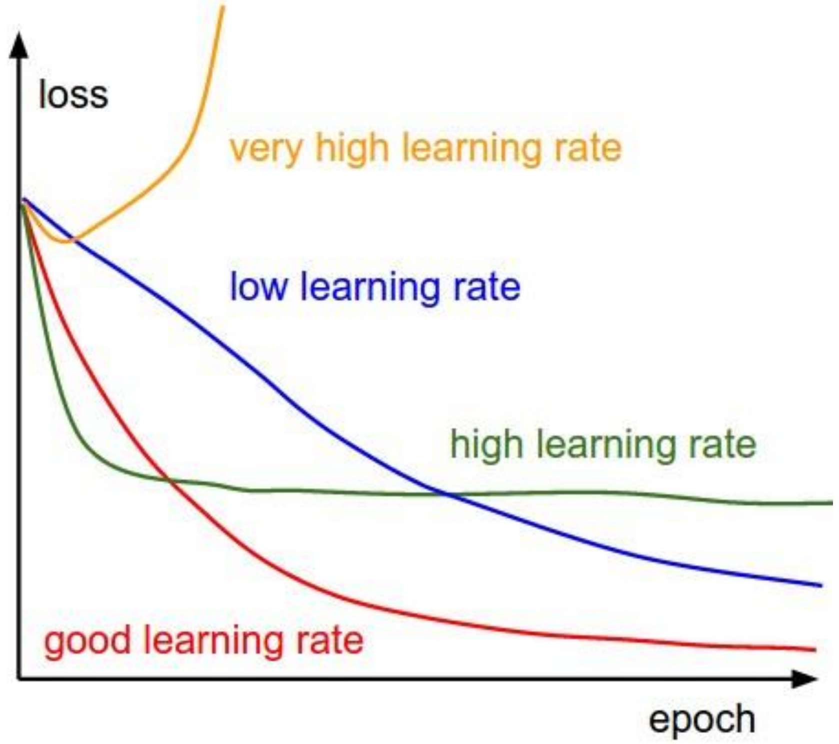
ANN Design and Issues

- Number of neurons, and hidden layers
- Initial weights (small random values $\in[-1,1]$)
- Choice of the transfer function
- Learning rate
- Weights adjusting
- Data representation, pre-processing, and splitting

Learning Rate

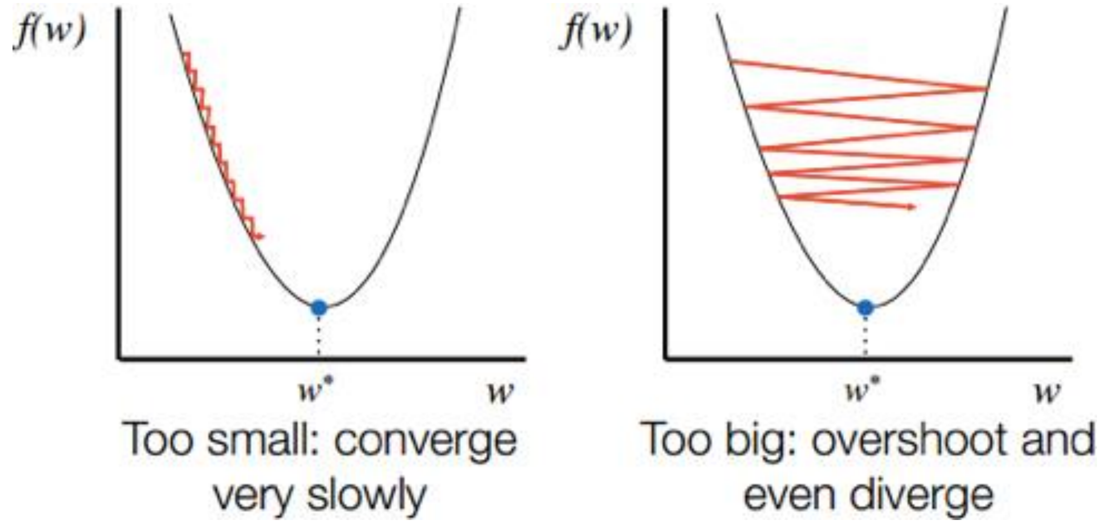
- The learning rate, η, is a configurable (**hyper)parameter** used in ANNs training
- η controls how *quickly* the model is adapted to the problem
- Practical value $0 < \eta < 1$.

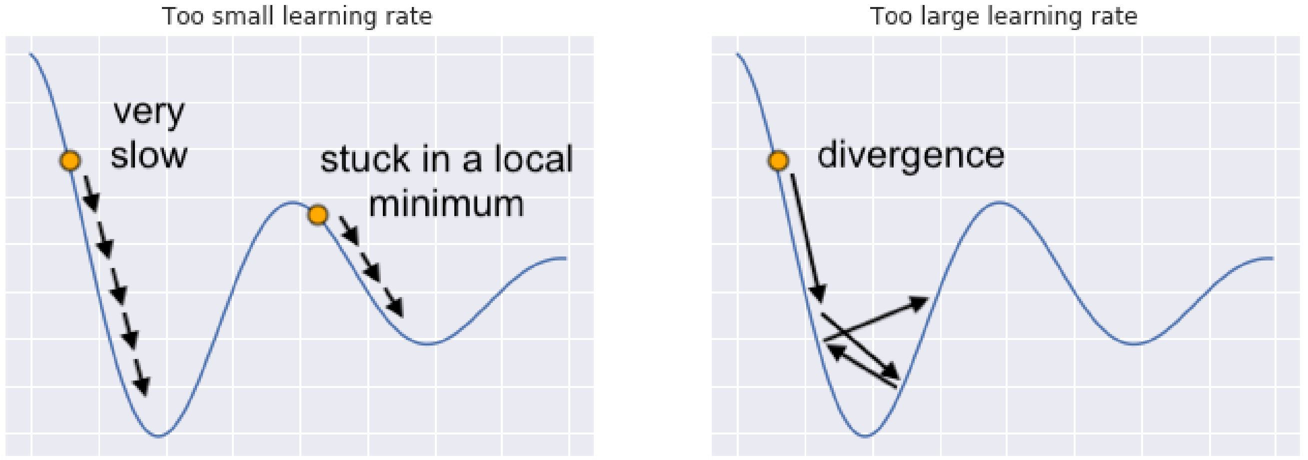
  - **Smaller** η $\rightarrow$ smaller changes to w $\rightarrow$ more training epochs
    - Can cause the local minima stuck.
  - Larger η $\rightarrow$ larger changes to w $\rightarrow$ fewer training epochs.
    - May results in divergence.



Graph Source: https://cs231n.github.io/neural-networks-3/

Learning Rate (cont'd)





Graph Source: https://towardsdatascience.com/the-learning-rate-finder-6618dfcb2025

Graph Source: https://srdas.github.io/DLBook/GradientDescentTechniques.html

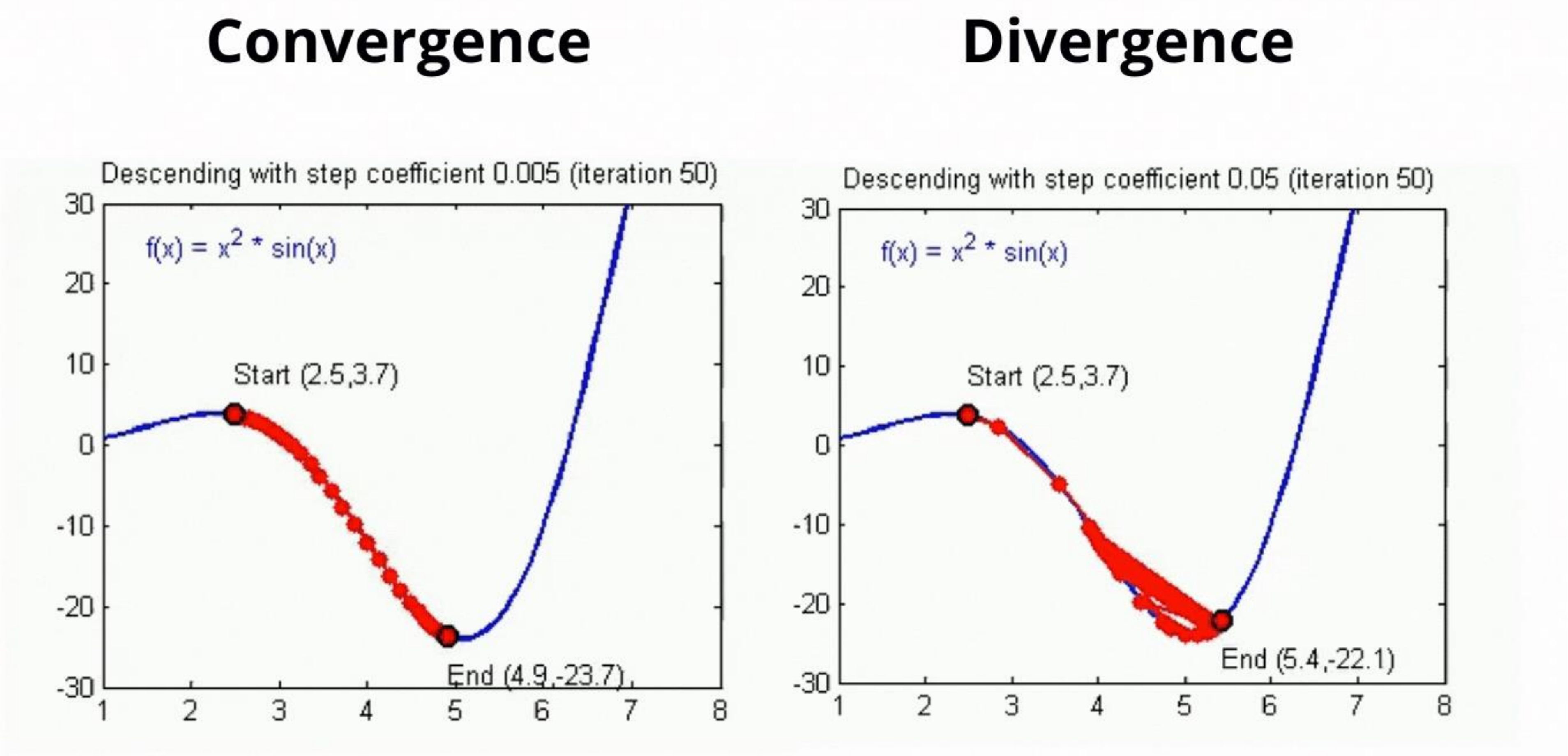
One technique that can help the network out of local minima is the use of a **momentum** term.

$$\Delta w_{kj}(n) = \eta * \delta_k(n) * x_j(n) + \boxed{\alpha \Delta w_{kj}(n-1)}$$

Momentum factor

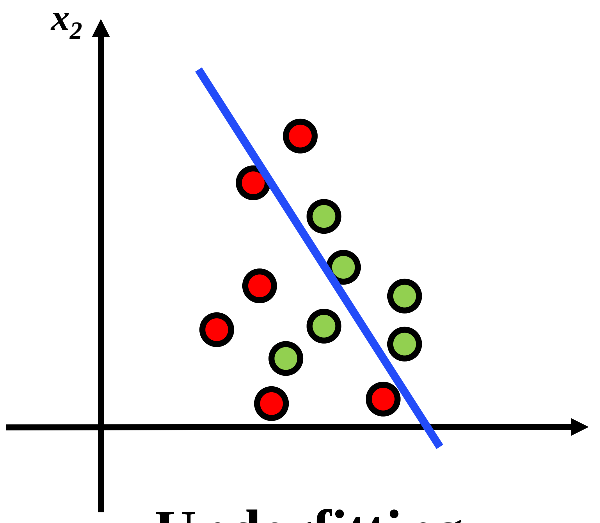
Weight increment from previous iteration

Learning Rate (cont'd)

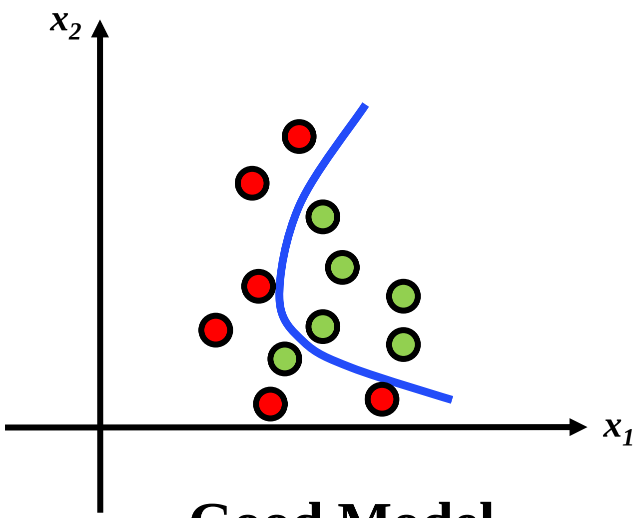


Graph Source: https://towardsai.net/p/machine-learning/analysis-of-learning-rate-in-gradient-descent-algorithm-using-python

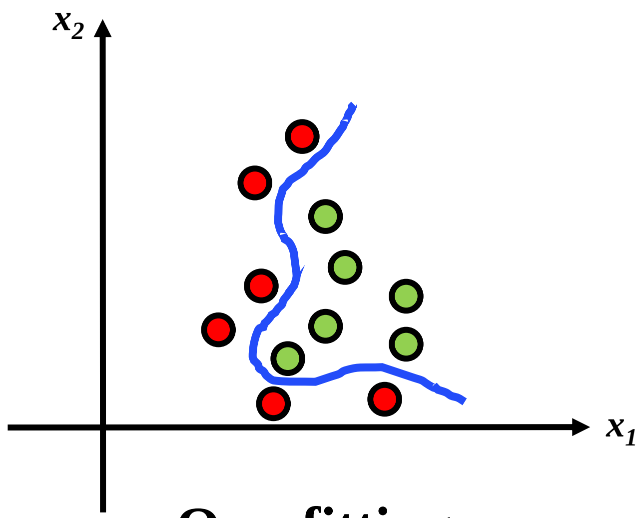
Overfitting



Underfitting



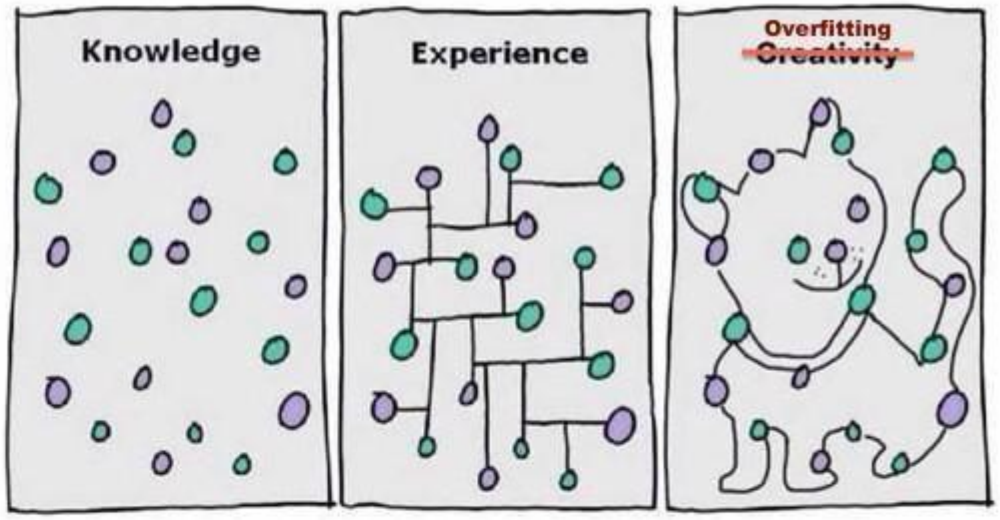
Good Model

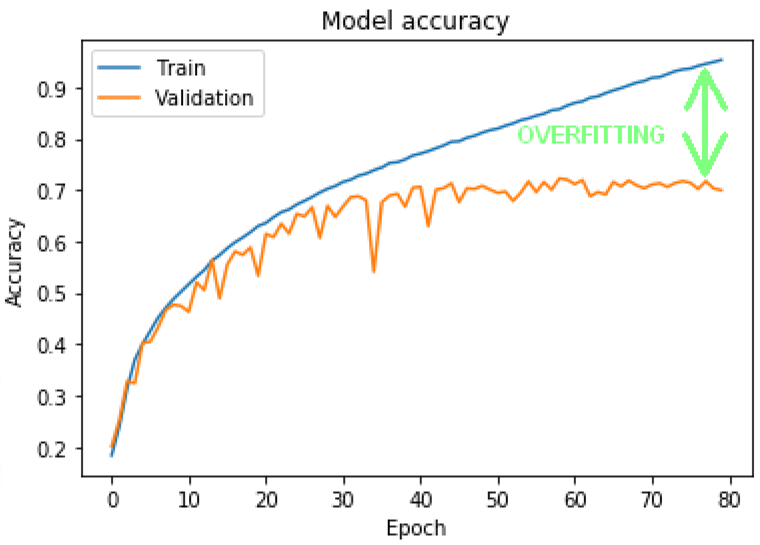


Overfitting

Can NOT be generalized

Correctly classify test patterns it has never seen (learned) before when tested in real-world problem





Solution

- Early stopping
- Regularization (Dropout)

Image Source: https://www.pinterest.com/pin/462604192955327068/

Vanishing Gradient

- Deeper neural networks (i.e., with multiple hidden layers) are *difficult* to train (difficulty increases geometrically).

$$\delta_j = f'(\cdot) * \sum_{k=1}^{D} \delta_k * w_k \quad \text{[local gradient] x [upstream gradient]}$$

  - The gradients get **smaller** and smaller when *backpropagating* the error.
  - After few layers of propagation, the gradient disappears (***vanishes***)
  - The parameters in the deep layer will be **almost static**

- Solution
  - **Modify** the activation function
  - **Use** batch normalization (sort of regularization)

ANN Advantages and Disadvantages

- Advantages
  - Very **simple** principles
  - Highly parallel: ***information processing is much more like the brain than a serial computer***
  - Adapt to unknown situations, can model *complex* functions
  - Ease of use, *learns by example*, and very little user domain-specific expertise needed.

- Disadvantages
  - Very **complex** behaviors
  - Not exact.
  - Needs training.

ANN Terminology

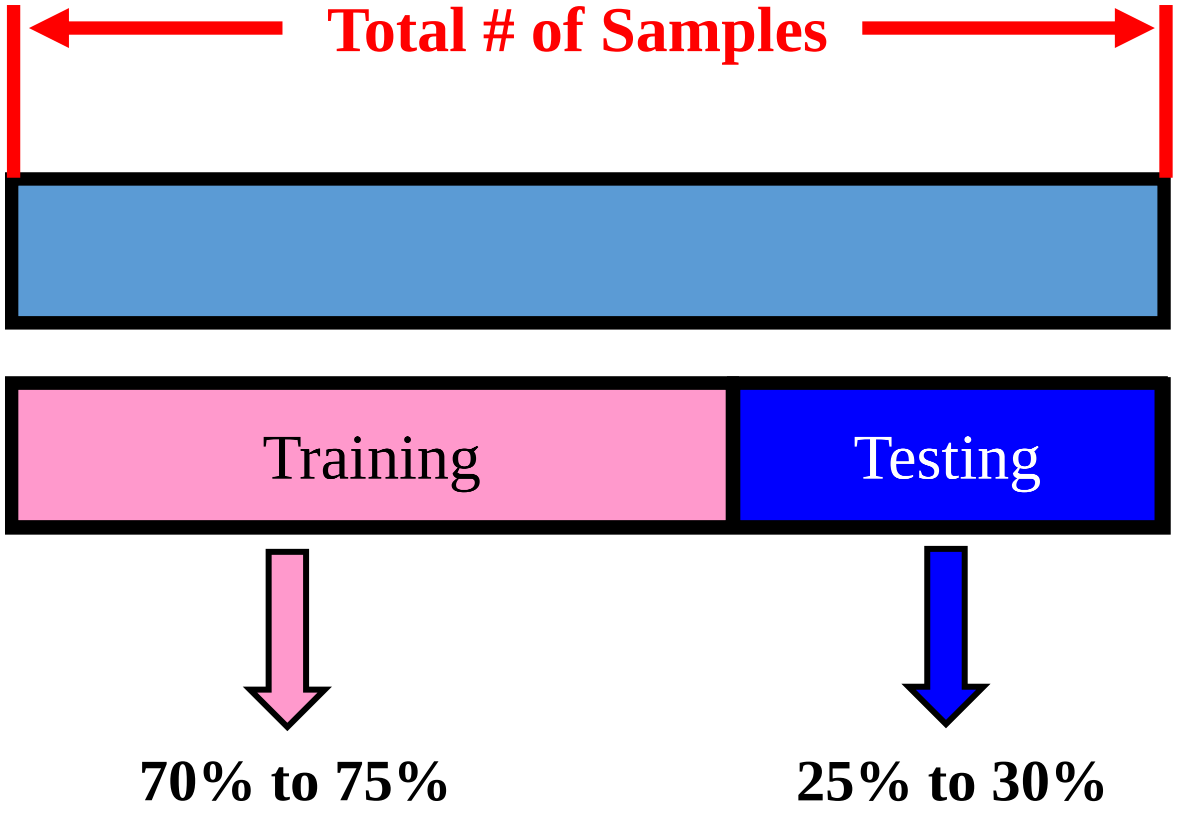
- Neuron, unit (node)
- Weight and bias
- Transfer function (linear, sigmoid, ReLU, etc)
- Loss function (*mean squared error*, cross entropy, etc.)
- Learning rate, epoch, batch
- Backpropagation (***error*** propagation)
- Optimization (gradient descent (GD), stochastic GD, Adam,….etc.)
- Overfitting
- Dropout, Batch normalization

Each ANN aspect is considered a standalone research venue

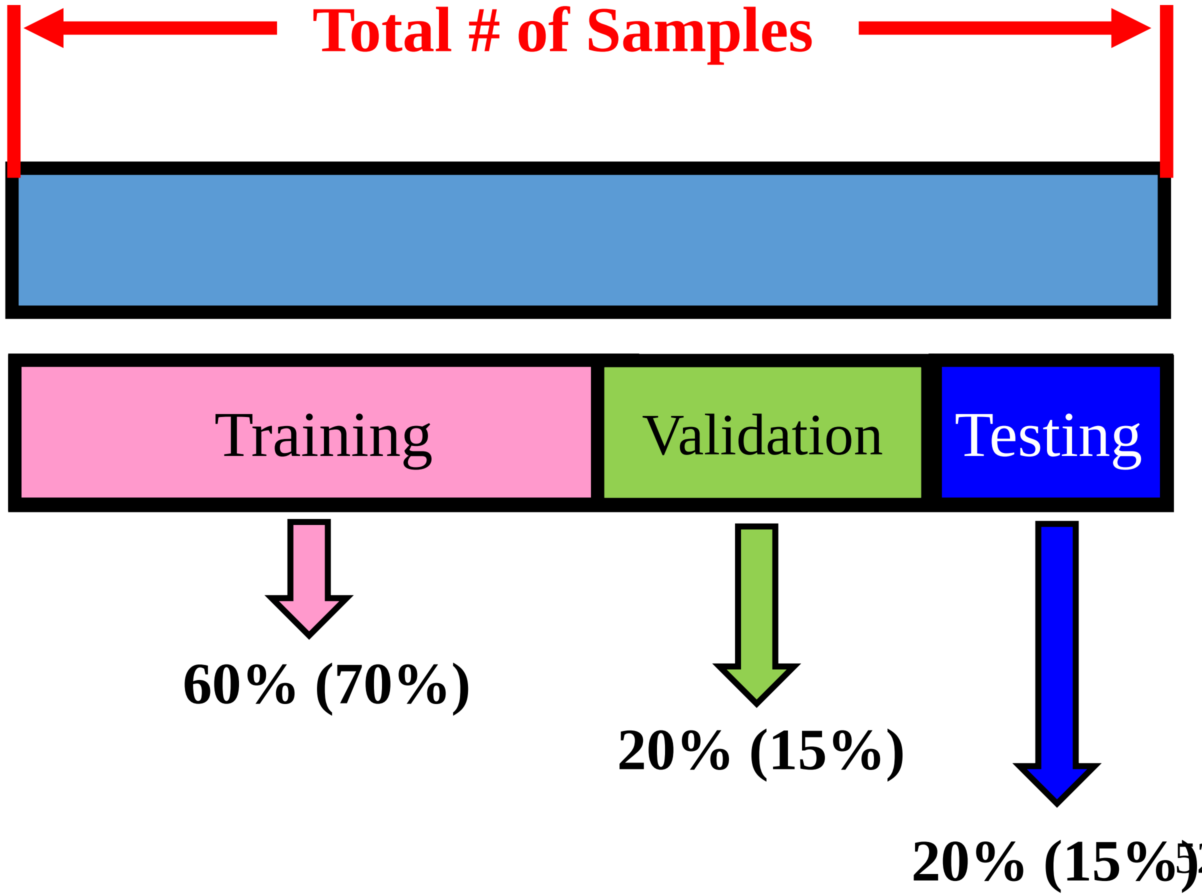
Validation Techniques

Data Splitting

Training/Testing

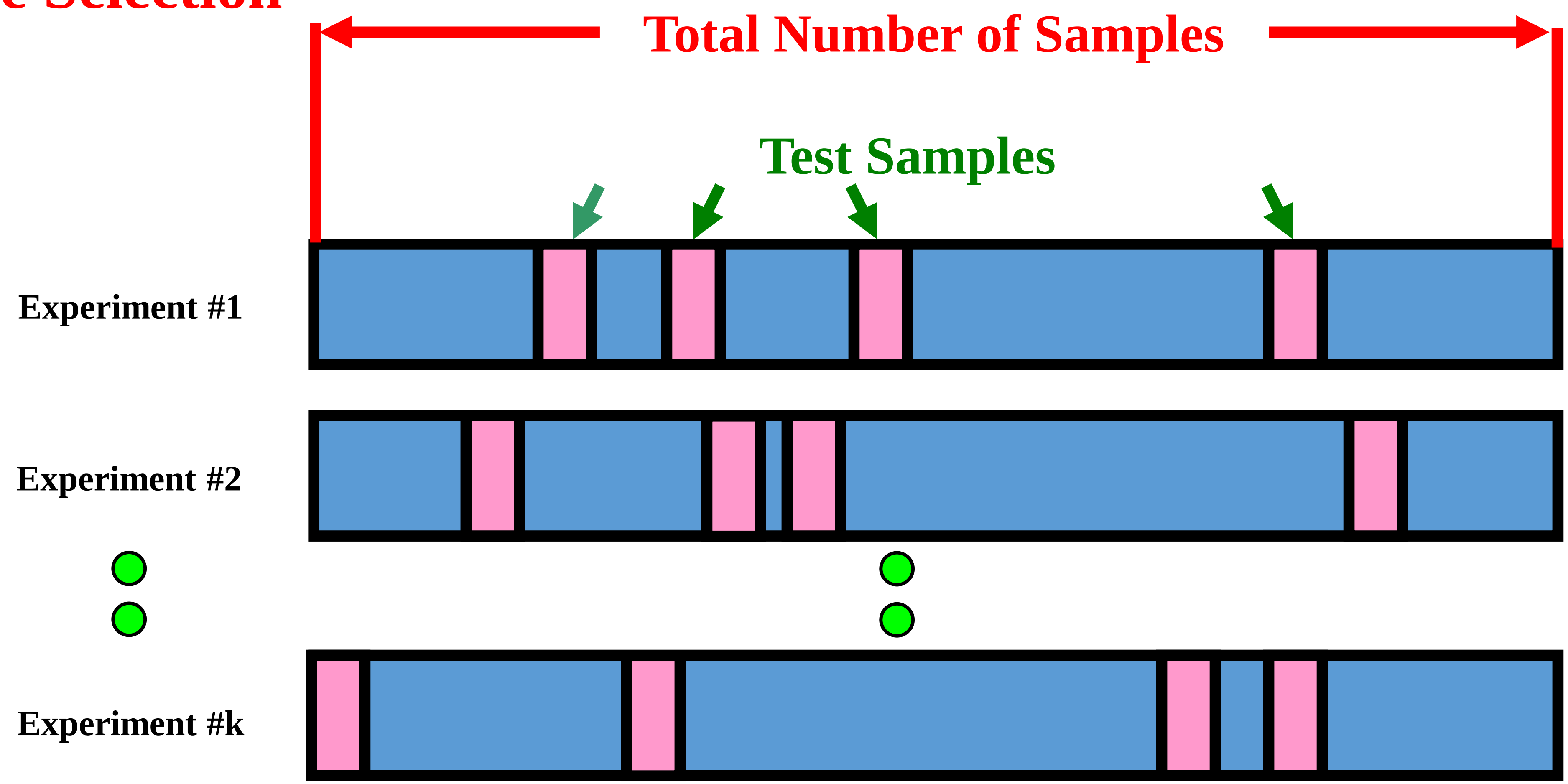


Training/Validation/Testing



Validation Techniques

Random Sample Selection



$$E = \frac{1}{k}\sum_{i=1}^{k} E_i$$

→k is the number of experiment

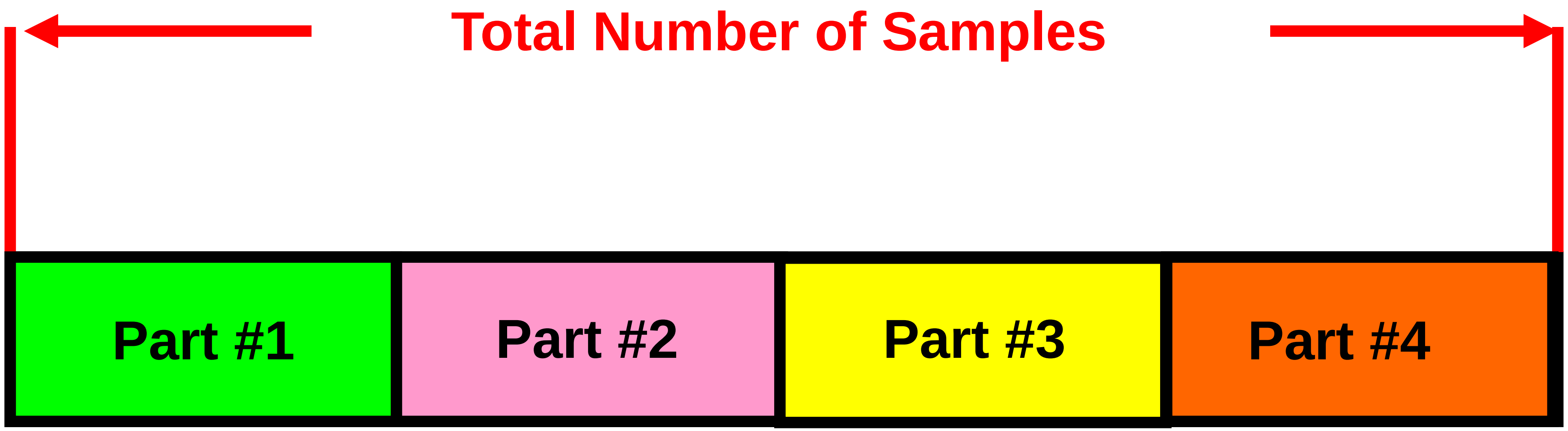
→ E_i is the average error for each experiment using only testing data

Validation Techniques

Cross Validation

Divide data into mutually exclusive and equal-sized subsets, folds, and this number is called K

Here K=4



$$E = \frac{1}{k}\sum_{i=1}^{k} E_i$$

→k is the number of folds

→ E_i is the average error for each fold

Validation Techniques

Testing	Training	Training	Training
Training	Testing	Training	Training
Training	Training	Testing	Training
Training	Training	Training	Testing

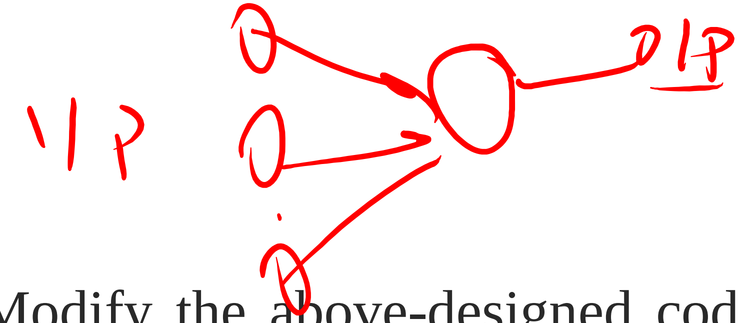
75, 25

Assignments

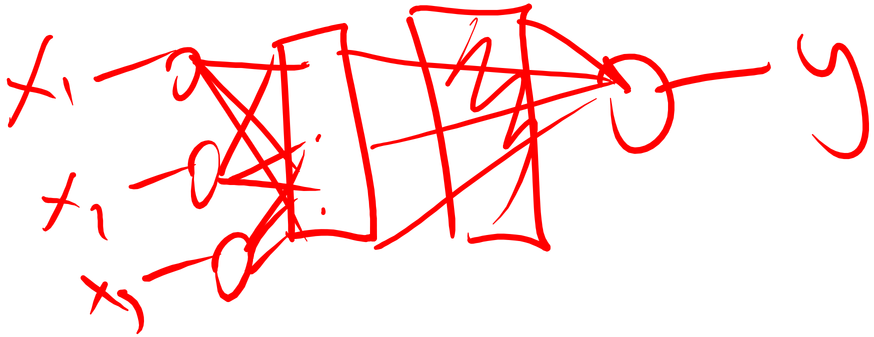
- **Assignment 1**: Design your own simple ANN, (one perceptron with one input layer and one output neuron). Use the data points listed in the adjacent Table as your training data. Assume the activation function is sigmoid and assume there is no bias for simplicity (b=0). Test your design using different iteration numbers.

y

X_1	X_2	X_2	d
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0



- **Assignment 2**: Modify the above-designed code to implement a multi-layer perceptron, MLP (an ANN with one input layer, one hidden layer and one output layer) for the same data points above. Assume sigmoid activation function and there is no bias for simplicity (b=0). Test your approach using different iteration numbers and different number of nodes for the hidden layer (e.g., 4, 8, and 16).



Assignments (cont'd)

- **Assignment 3**: Use the Keras library *(tensorflow.keras)* to build different ANNs using different numbers of hidden layers (**shallow**: 1 hidden, output layer, **deeper**: two hidden layers with 12 and 8 nodes respectively, and **more deep**: three hidden layers with 32, 16, 8 nodes respectively). Use the provided diabetic data sets (here) to train and test your design. Use the ReLU activation for the hidden layers and the sigmoid activation for the output neuron, loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'], epochs = 150.

100%

- **Assignment 4**: Redo assignment #3 using 80% of the data for training and 20% of the data for testing. Also, plot the training accuracy and loss curves for your designed networks

Thank You
&
Questions