Introduction to Neural Networks

Most slides are from Noriko Tomuro

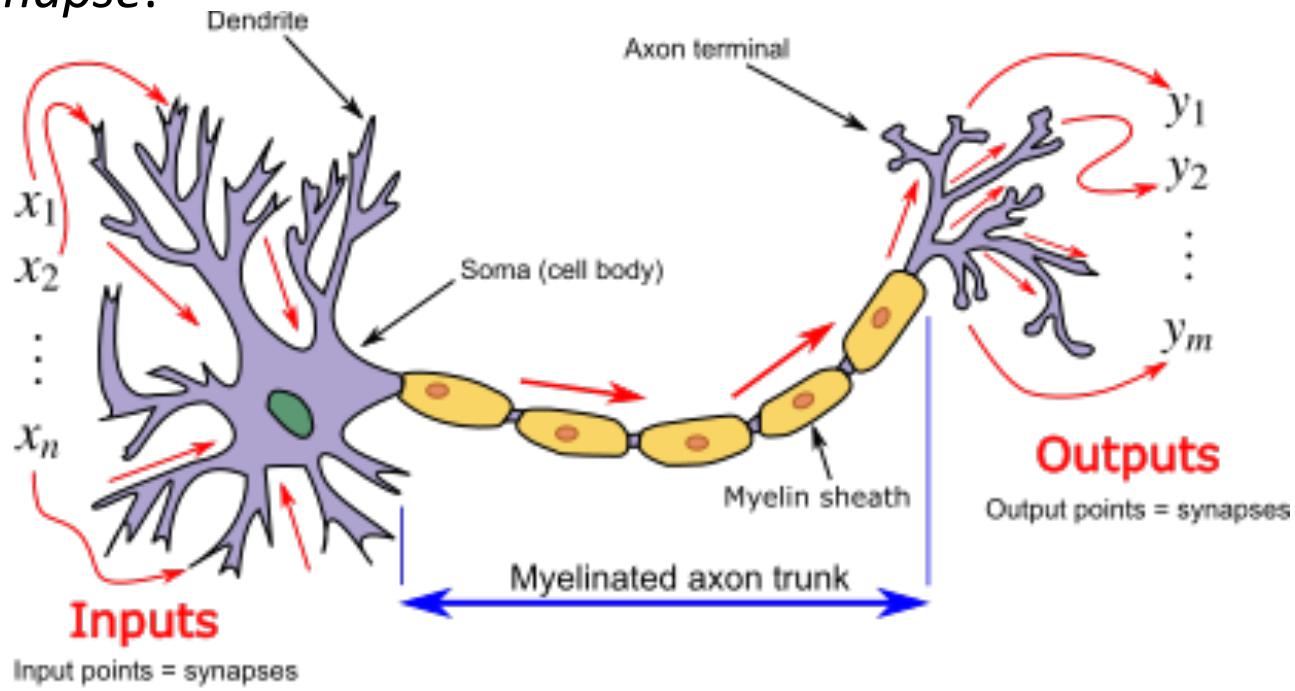
Neural Networks: Overview

- ***Origin** of Neural Networks – **AI*** [since 1940's]:

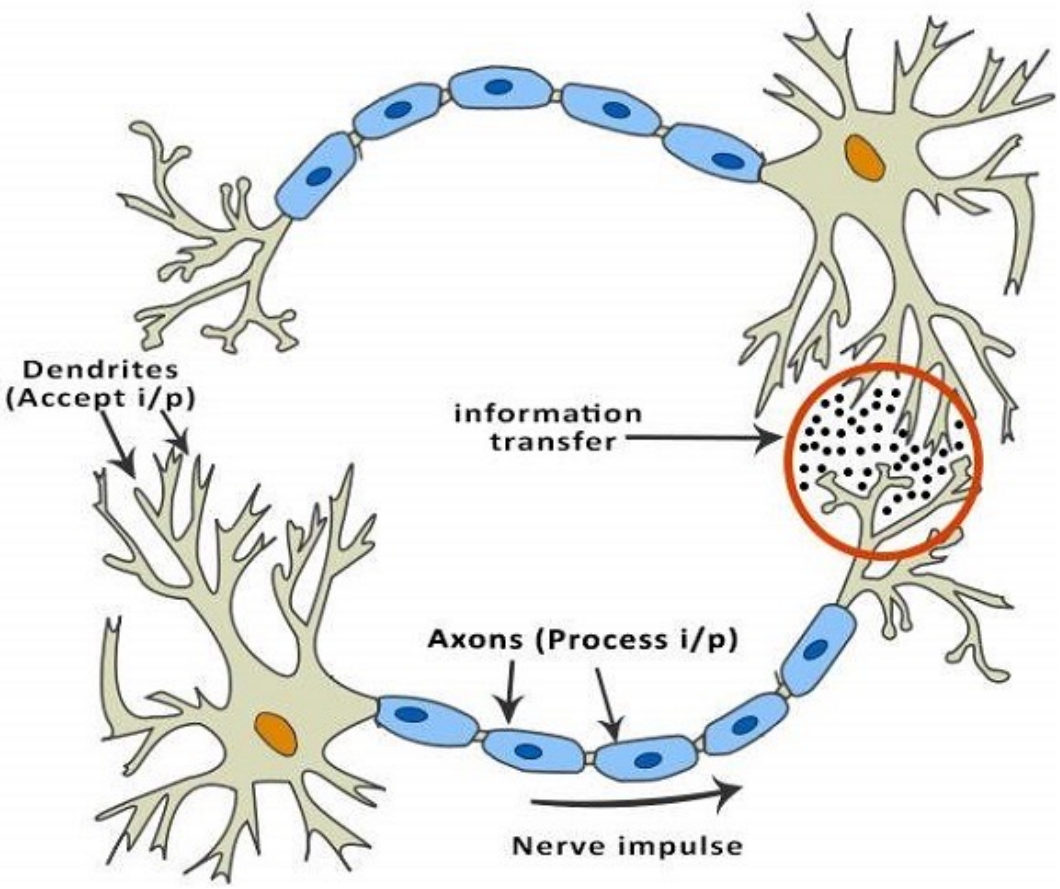
  "Artificial neural networks (ANNs) or connectionist systems are computing systems inspired by the **biological neural networks** that constitute animal **brains**." [Wikipedia]

- The goal of ANNs is to obtain highly effective machine learning algorithms (i.e., *human intelligence*) using computer models that **mimic human brains** – in the physical **architecture** (a network of densely connected nerve cells (**neurons**), thus called *'connectionist'*), as well as the **working** (processing and transmitting chemical and electrical signals).

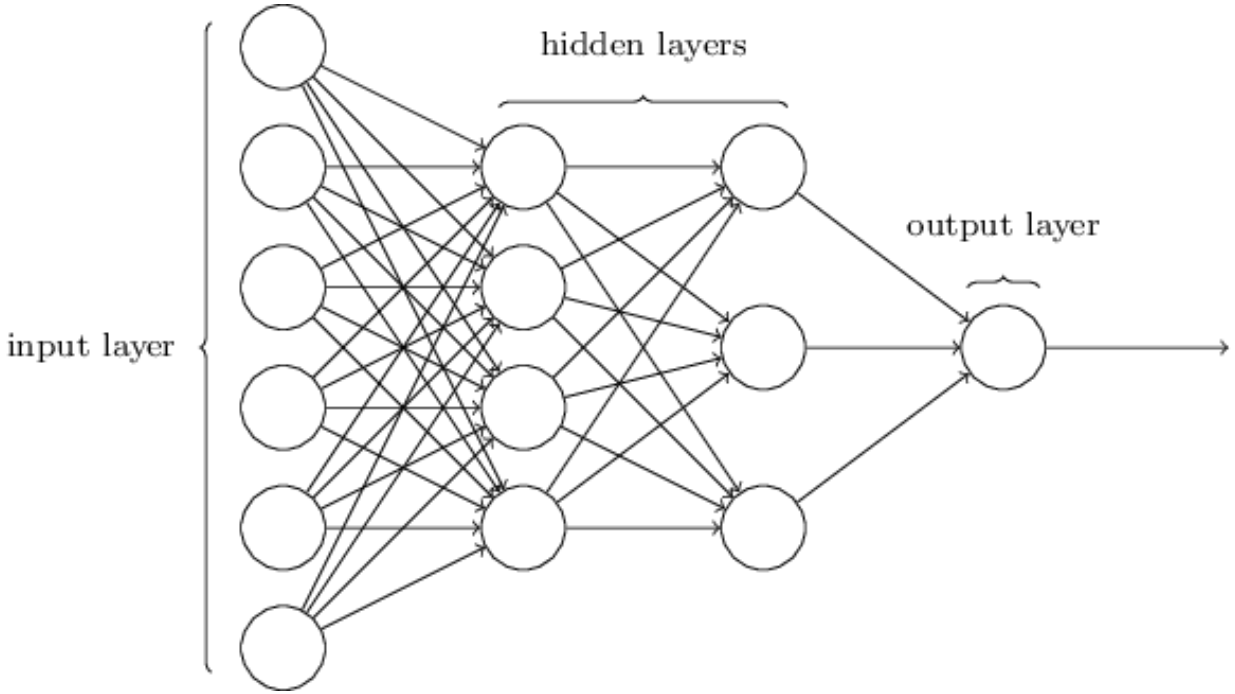
- A human brain consists of roughly 10^{11} (= **100 billion**) neurons, and each one is connected to $10^{4\text{-}5}$ (= 10,000 to 100,000) other neurons.
- A neuron receives input signals via *dendrites*. The signal raises or lowers the electrical potential inside the body of the neuron. If this membrane potential reaches some threshold, the neuron **fires**, and a pulse is sent down the axon. The axons divide (arborise) into connections to many other neurons, connecting to each of these neurons in a *synapse*.



- Neurons in a brain are densely interconnected, forming a network. Information signals (electrical pulses) are propagated through the brain though neuron synapses.

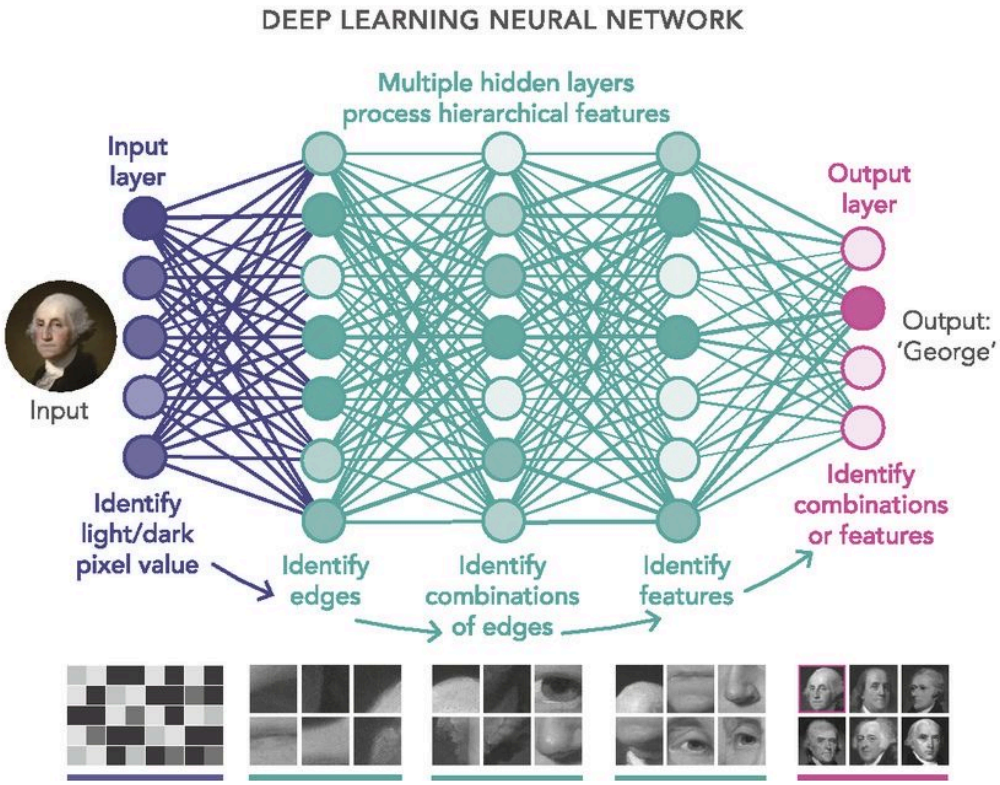


- Artificial Neural Networks (ANNs) imitate human brains using a densely connected simple processing elements/**nodes**.
- Nodes are also made into **layers** (one input, one output, and zero or more 'hidden' layers in between).

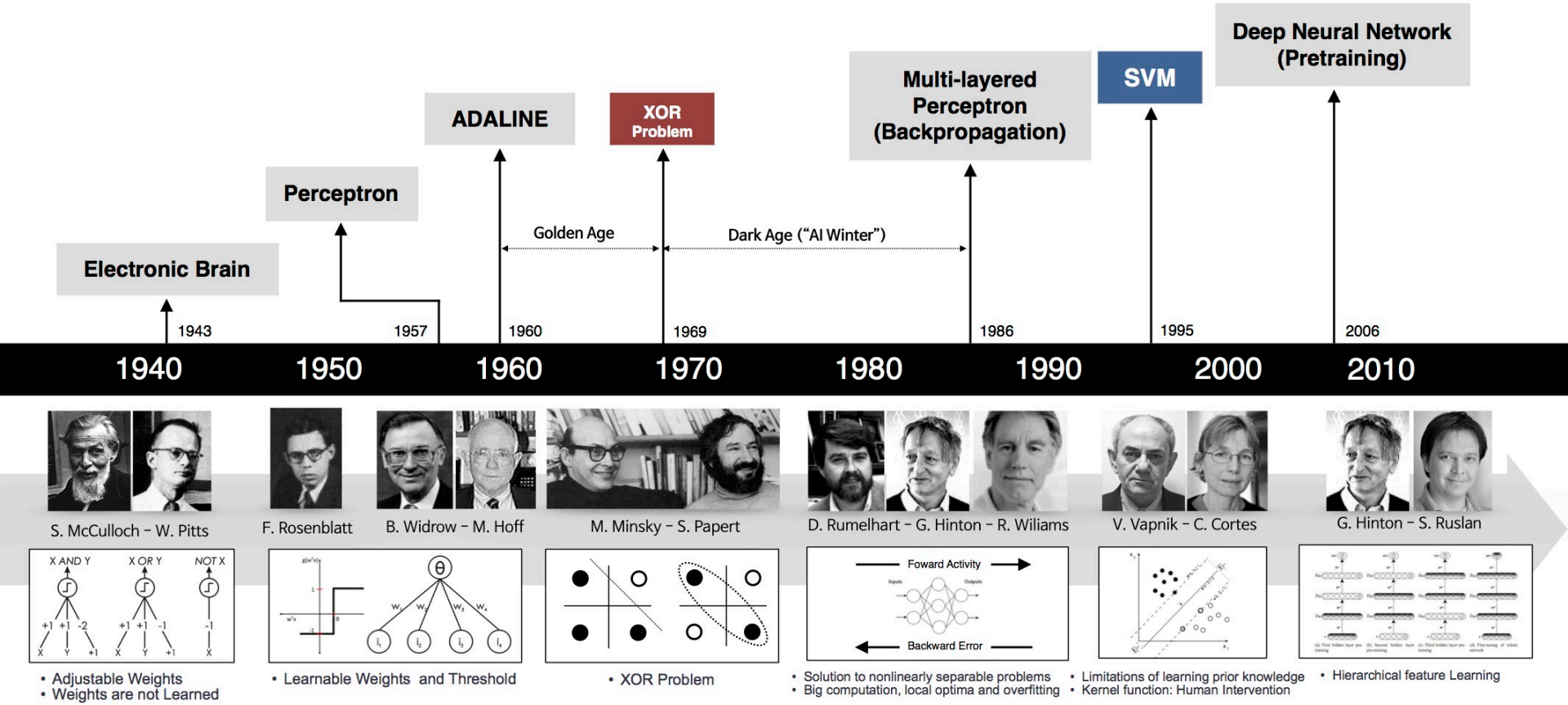


- The basic scheme is that signals presented to the input layer are **propagated forward**, through hidden layers, to the output layer.

- ***Modern*** *Neural Networks –* ***Deep Learning****.*
  "until 2006 we didn't know how to train neural networks to surpass more traditional approaches. What changed in 2006 was the discovery of techniques for learning in so-called **deep neural networks**. A deep neural network (DNN) is an artificial neural network (ANN) with ***many layers*** between the input and output layers."



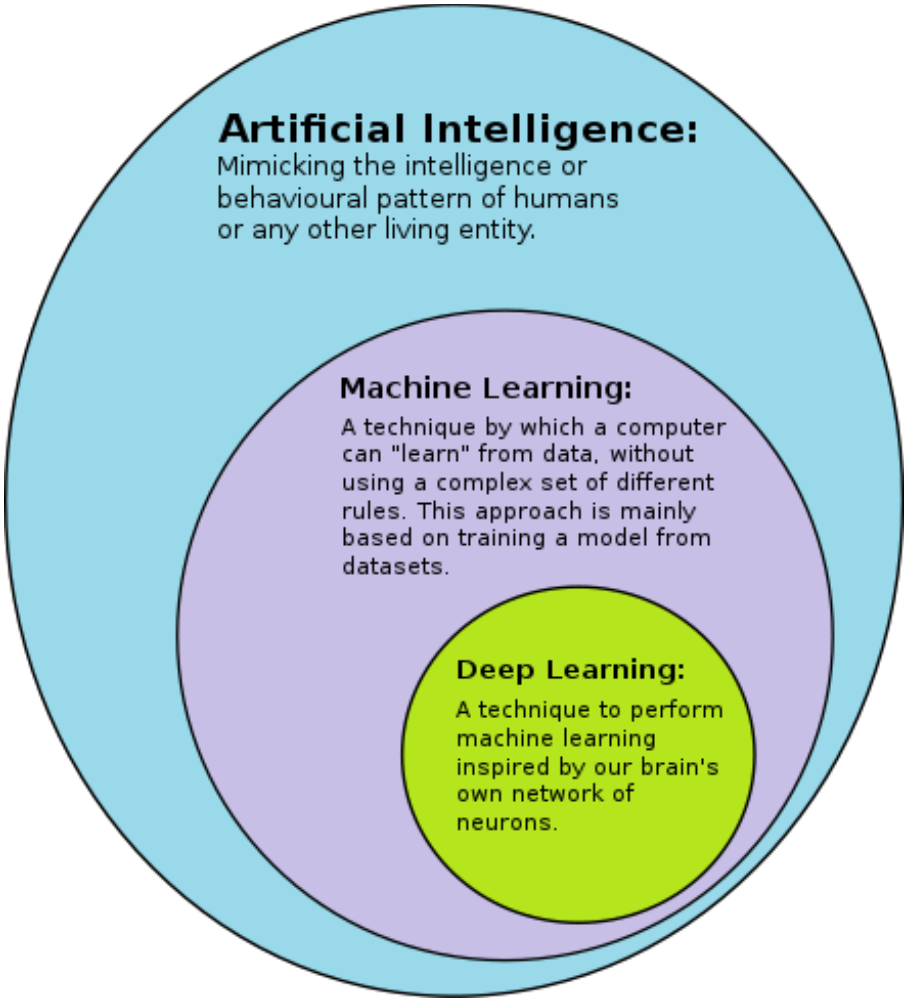
ANN History Chart



Also a good reference on the history of Neural Networks:
"A brief history of Neural Nets and Deep Learning" by A. Kurenkov

Deep Learning in AI

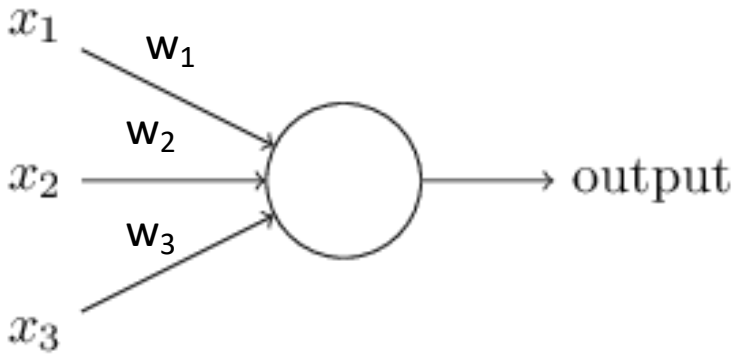
- **Deep Learning** is part of a broader family of machine learning (ML) ***methods*** based on artificial neural networks.
- Note: *Data Science* uses ML and deep learning techniques to analyze data (i.e., for *analytics*). Data Science also encompasses non-AI tasks such as data gathering, data cleaning and data manipulation.



Perceptron

- **Perceptron** is the basic computing element (processing unit) used in ANNs – it simulates a human neuron.
- A perceptron receives input signals (one or more) and outputs a **binary** value: **1** if the sum of the signals was above a certain **threshold** or **0** otherwise -- indicating the state of the neuron: '**on**' (when 'a neuron was fired') or '**off**'.
- Input signals are also **weighted** when they are aggregated.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$



- And the network of interconnected perceptrons shown in the previous slide simulates a human brain.

[Supplement] Perceptron and Decision Boundary

- Since a perceptron outputs binary values (1/0), it could be used to as a **binary classifier**.
- IMPORTANT: A perceptron defines a **decision boundary** of the binary classifier which is **linear**.

Output **1** if

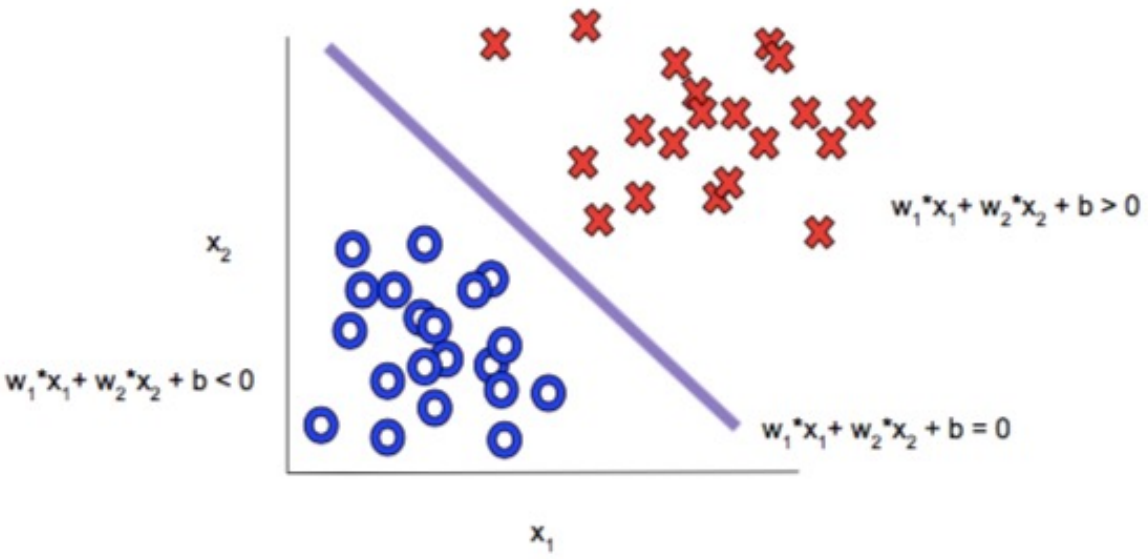
$$\sum_{i=1}^{2} w_i \cdot x_i > b$$

equals to:

$$\sum_{i=1}^{2} w_i \cdot x_i - b > 0$$

which is

$$\boldsymbol{w_1 \cdot x_1 + w_2 \cdot x_2 - b > 0} \text{ or}$$
$$\boldsymbol{w_1 \cdot x_1 + w_2 \cdot x_2 + b' > 0}$$



Two 'AI Winters' (1)

Development of ANN technologies has met two significant challenges in the history of AI.

(1) The XOR problem

- Since inception (in the 1940's), perceptrons were enthusiastically received in the AI community and created much **hype** as the revolutionary technique that would allow us to build an ultimate AI machine: the 'Thinking Machine'.
- However, Minsky and Papert published a book in 1969 (titled 'Perceptrons') where they argued that a perceptron with **one single layer** can only model ***linearly separable*** functions – and it cannot even model a simple function such as **XOR** (exclusive-or, which is **not** linearly separable).
- Because of this publication by the influential AI researchers, research in ANNs **died down** and eventually lead to a so-called **'AI Winter'** (in the 1970's; a period in the AI history characterized by disillusionment and funding freeze).

[Supplement] Logical Operators and Truth Tables

- Basic connectives:
  1. "*p* **and** *q*" is the conjunction, noted "$p \wedge q$".
     e.g. "The earth is flat and March has 31 days."
  2. "*p* **or** *q*" is the disjunction, noted "$p \vee q$".
     e.g. "The earth is flat or March has 31 days."

     NOTE: The meaning of or here is **inclusive**, that is, if one is true, the truth of the other can be either true or false (i.e., not necessarily false). For example, "I will buy a car, or I will take a vacation."
  3. "**not** *p*" is the negation, noted " $\bar{p}$ " (or "$\neg p$" or "$\sim p$").
     e.g. "The earth is not flat." or "It is not the case where the earth is flat."

- **Exclusive-Or** ($\oplus$)
  - The regular Or ($\vee$) is inclusive – it is true if either literal is true, or BOTH literals are true.
  - Another, more strict Or, is Exclusive-Or, denoted $\oplus$. It is true strictly when EITHER literal is true, not both.

Truth Tables and Logic Operators as Binary Functions

AND

$p \wedge q$

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

OR

$p \vee q$

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

XOR

$p \oplus q$

p	q	$p \oplus q$
T	T	**F**
T	F	T
F	T	T
F	F	F

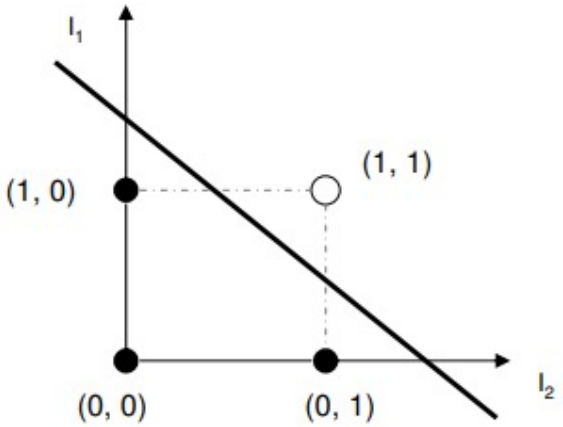
NOT

$\neg p$

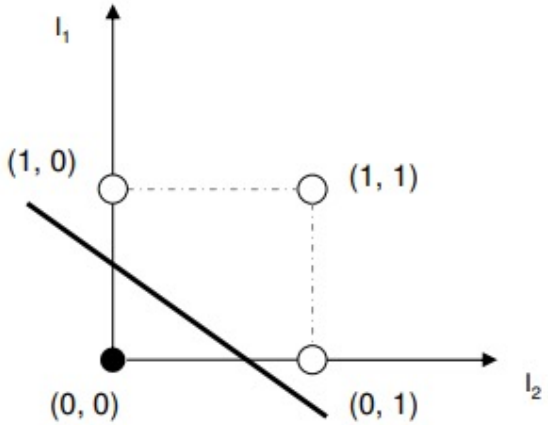
p	$\neg p$
T	F
F	T

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$	$\neg p$
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

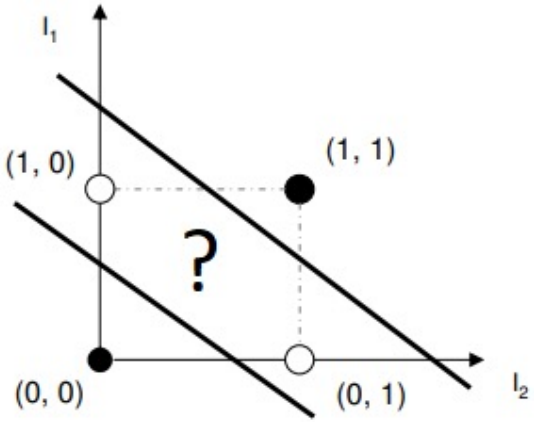
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

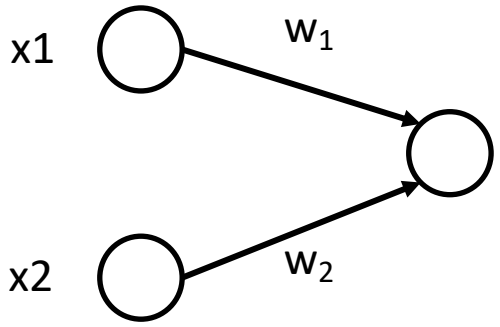


XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



Exercise: Design a perceptron that computes AND by giving weighs to w_1 and w_2. Use threshold == 1.5.

x1	*x2*	*x1 AND x2*
1	1	1
1	0	0
0	1	0
0	0	0

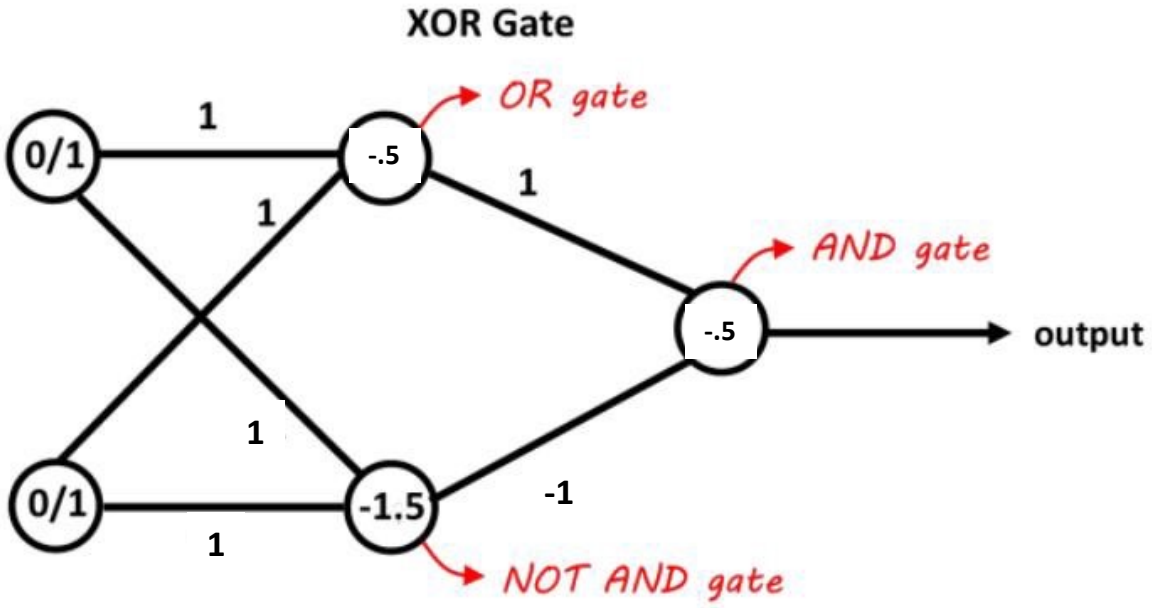


$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

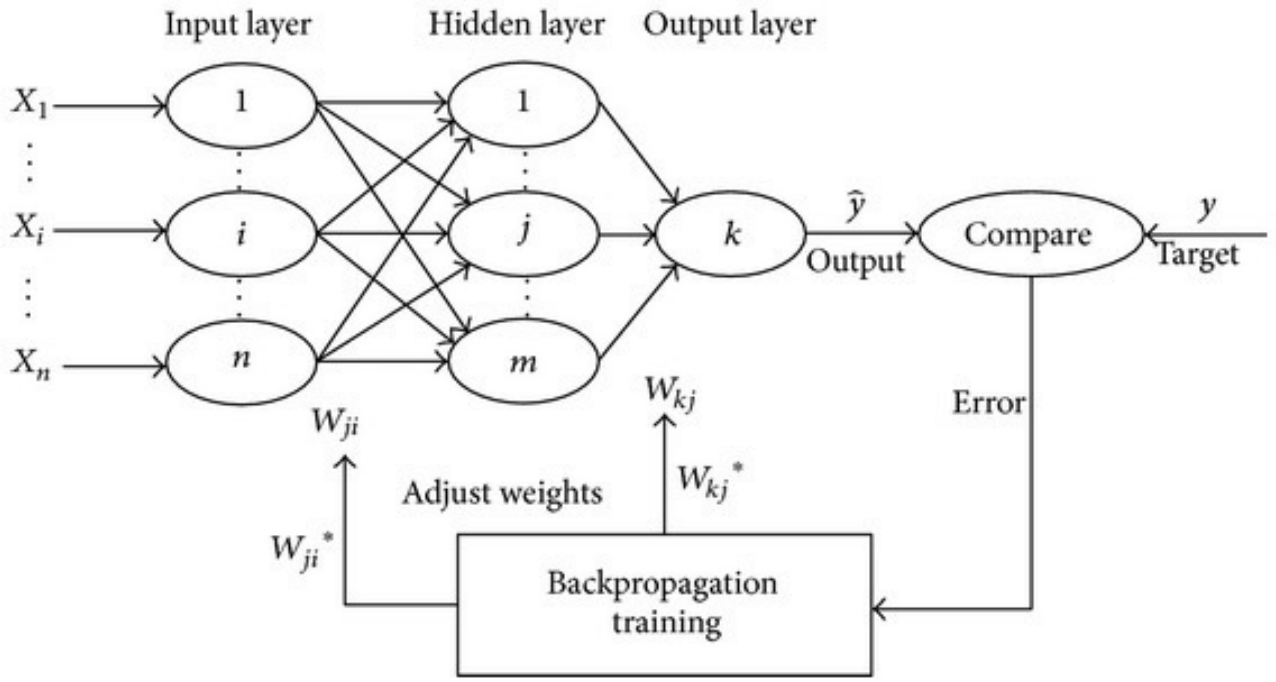
NOTE: XOR can be modeled by **multi-layer perceptrons (MLPs)**. But the real problem was that the **simple perceptron *learning* algorithm** proposed back then did not work for multi-layered perceptrons.

Exercise: Verify the MLP below computes XOR. Note the values inside the circles indicate thresholds.

x1	*x2*	*x1* ***XOR*** *x2*
1	1	0
1	0	1
0	1	1
0	0	0



- [resolution] A decade later, in 1986 the new NN learning algorithm **'Backpropagation'** was invented by Rumelhart, Hinton and Williams. This algorithm uses an optimization technique (gradient descent) to back-propagate the classification **error** from the output layer all the way back to the input layer through (any number of) hidden layers in the network – **Multi-layer** Backpropagation.

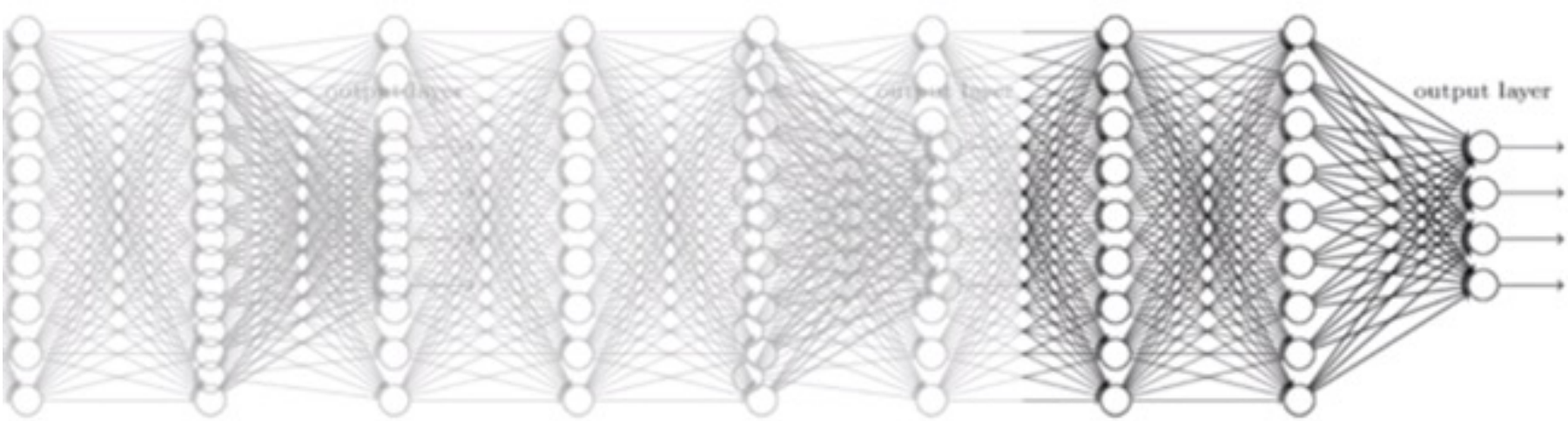


Two 'AI Winters' (2)

(2) Backpropagation

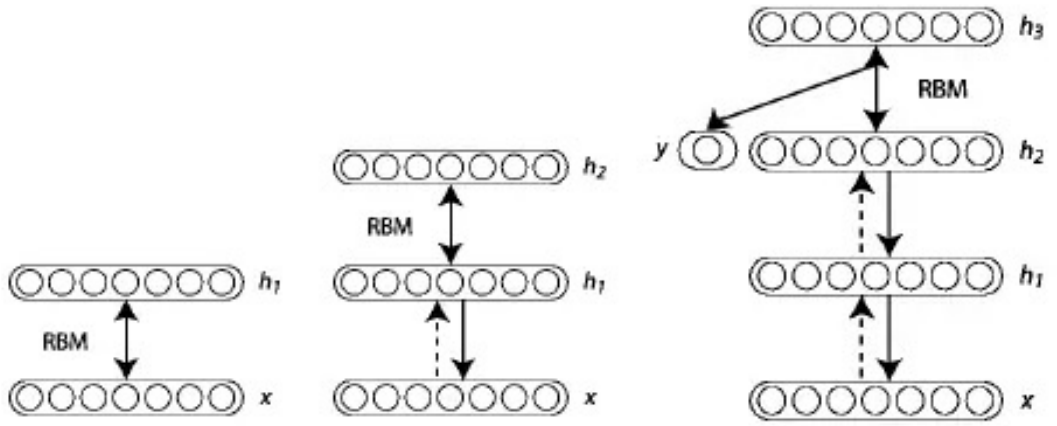
- After Backpropagation was (re-)invented, ANNs regained popularity in the AI community and the technology flourished. It was also proven mathematically that "multi-layer feedforward networks are **universal approximators**" -- multiple layers allow neural nets to model ***any*** function.
- However, already by the late 1980's, it was known that deep neural nets trained with Backpropagation just **did not work very well**. In addition to accuracy, the algorithm was also **slow** to converge. In the meantime, other traditional, non-NN machine learning algorithms (e.g. SVM, Random Forests) showed comparable or even better performance.

- The reason for mediocre performance was that, with many layers in the network, the gradient-based propagation of error ends up with either huge or tiny amount and the resulting neural net just does not work very well - the '**vanishing** or **exploding gradient problem**'.
- So, around the mid 90s, a ***new AI Winter*** for neural nets began - the community once again lost faith in ANNs.



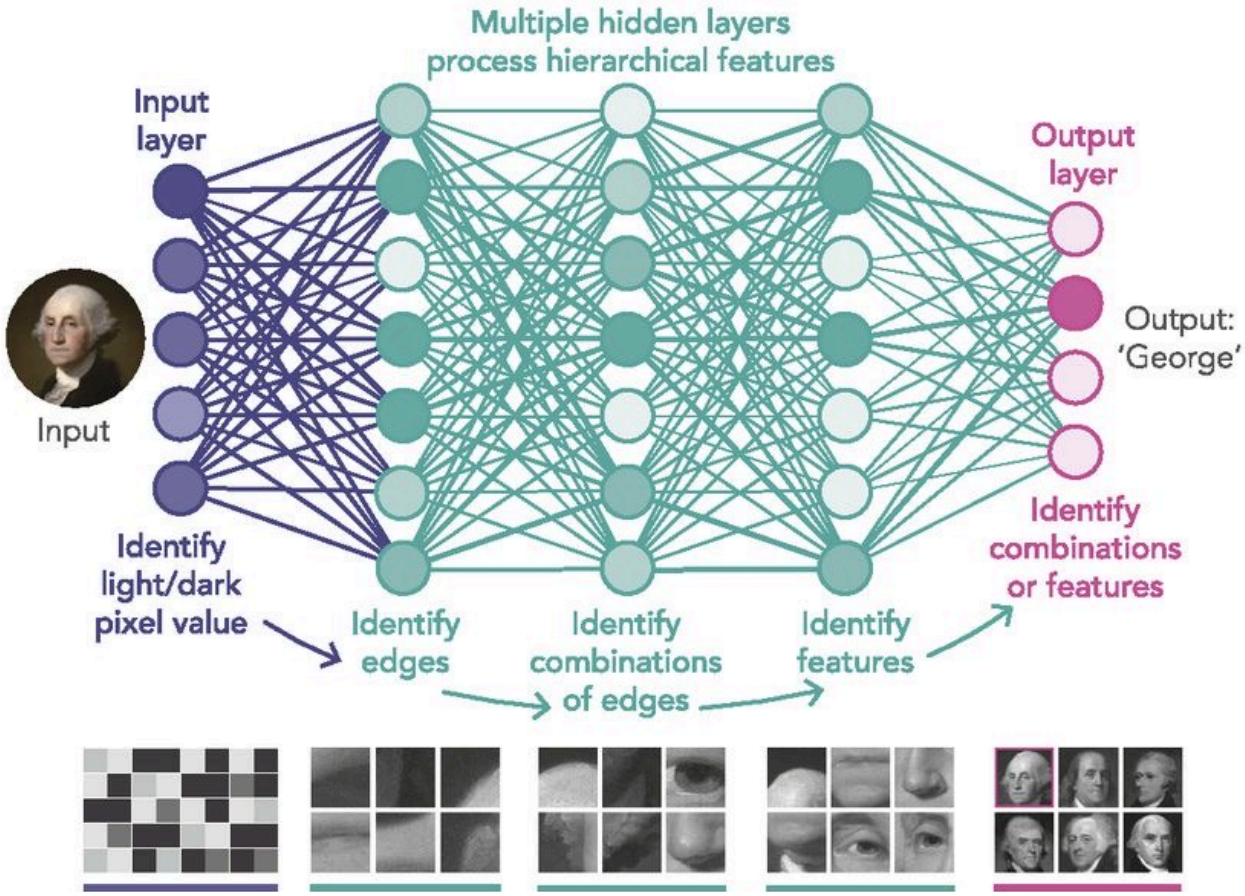
https://www.skynettoday.com/overviews/neural-net-history

- [resolutions] In **2006**, a Science paper published by **Hinton**, et al., "*A fast learning algorithm for deep belief nets*", made a breakthrough and the interest in NNs started to surged again. He also rebranded NN as "**deep learning**".
- The main premise of the paper is that NNs with many layers really could be trained well, if the weights are initialized in a clever way rather than randomly. The basic scheme is to train each layer one by one with unsupervised training (a **Restricted Boltzmann Machine** (RBM)), and then finishing with a round of supervised learning just as is normal for NNs.



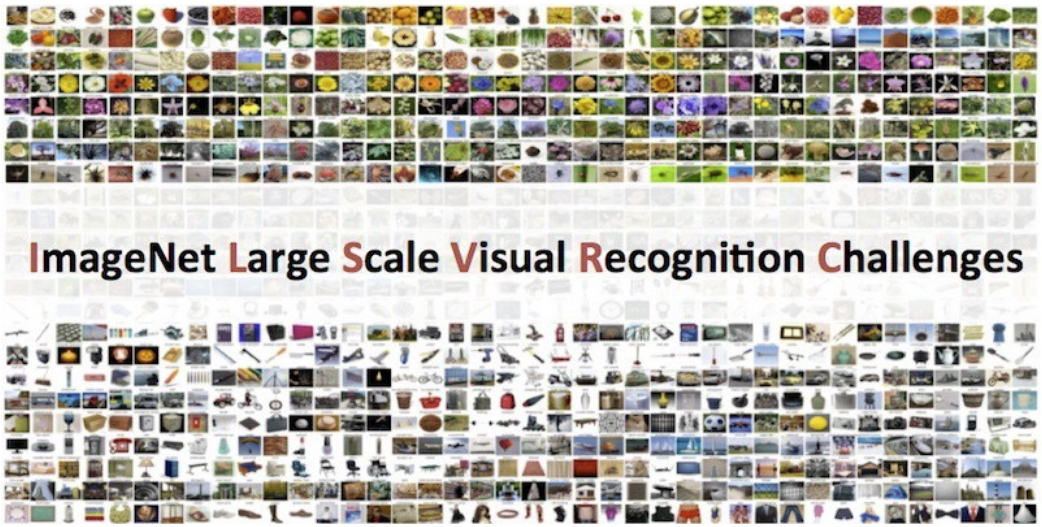
(a) Train RBM for $\mathbf{x}$ (b) Train RBM for $\mathbf{h}^1$ (c) Train RBM for $\mathbf{h}^2$ and y

- After several modifications and improvements, deep learning started to take off. One of the key discoveries was that, having many layers of computing units, good **high-level representation of data could be learned**.



Deep Learning Revolution

- In September 2012, a convolutional neural network (CNN) called AlexNet achieved a top-5 error of 15.3% in the ImageNet 2012 Challenge, **more than 10.8 percentage** points lower than that of the runner up. This was made feasible due to the use of graphics processing units (GPUs) during training.
- Some researchers state the competition victory established the start of a ***deep learning revolution*** that has transformed the AI industry.



https://en.wikipedia.org/wiki/ImageNet#ImageNet_Challenge

Top Applications of Deep Learning

Many lists exist, but here is one example:

1. Self Driving Cars
2. News Aggregation and Fraud News Detection
3. Natural Language Processing
4. Virtual Assistants
5. Entertainment
6. **Visual Recognition**
7. Fraud Detection
8. Healthcare
9. Personalisations
10. **Various Forecasting**
11. Colourisation of Black and White images
12. Adding sounds to silent movies
13. Automatic Machine Translation
14. Automatic Handwriting Generation
15. Automatic Game Playing
16. Language Translations
17. Pixel Restoration
18. Photo and video Descriptions

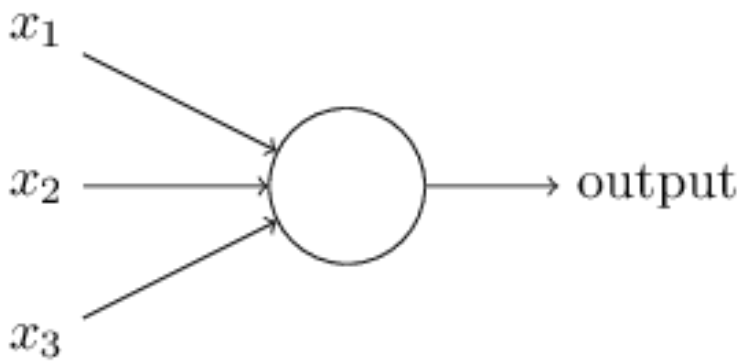
https://www.mygreatlearning.com/blog/deep-learning-applications/

Basic Concepts

(Some figures adapted from NNDL book Ch.1)

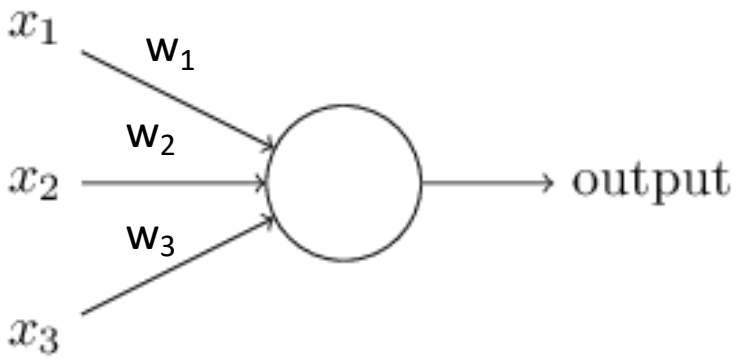
1. Perceptrons

- To mimic biological neural networks on computers, each neuron can be viewed as a separate processor, performing a very **simple** computation: deciding whether or not to fire given the input.
- In 1943, McCulloch and Pitts published the first concept of a simplified artificial brain cell, called '**perceptron**'. A perceptron is essentially a simple logic gate with several inputs and a **binary output (1/0)**; if the accumulated input signal exceeds a certain **threshold**, a positive output signal is generated.



- Then how does learning occur in the brain? The principal concept is *plasticity*: modifying the strength of **synaptic connections between neurons** and creating new connections.
- In 1957, Rosenblatt introduced **weights** on connections, which are real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is thereby determined by whether the weighted sum $\sum_j w_j \cdot x_j$ is less than or greater than some threshold value.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

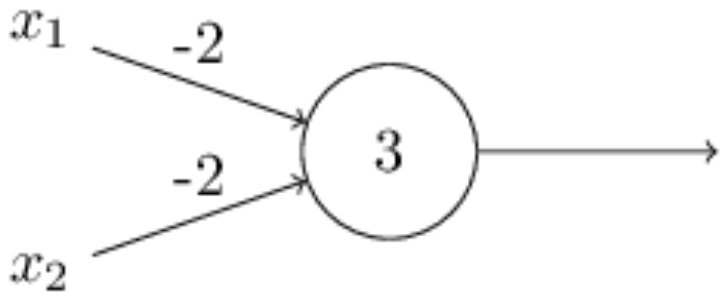


NOTE: Two notational changes

- Instead of a weighted sum $\sum_j w_j \cdot x_j$, we will use a **dot product** between w and x as two vectors, $\boldsymbol{w} \cdot \boldsymbol{x}$ (or later $w^T x$ when both are multi-dimensional tensors).
- Change *threshold* to a **bias** (b) of the perceptron.

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Note that, with this change, we have **threshold == -b**.
Then we indicate the bias inside the circle, e.g.



NOTE: Step function

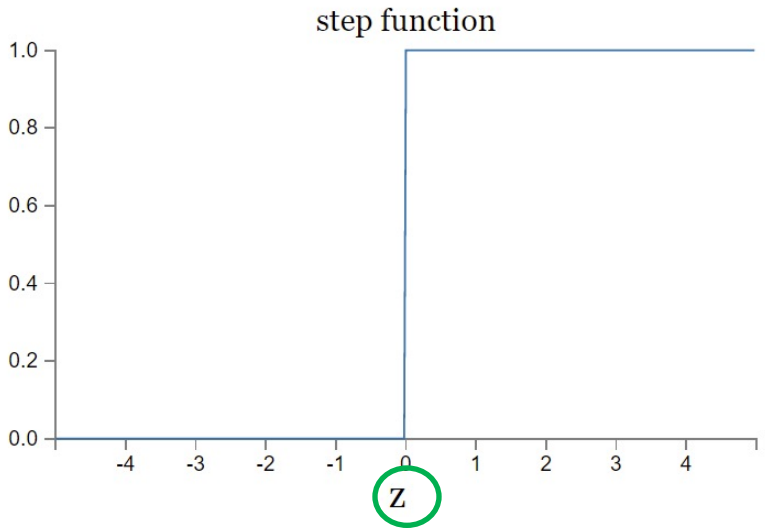
- Perceptron's final output, either 0 or 1, is considered the result/output of applying a step function to the dot product

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

is equivalent to a function *f* such that

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

where $z = w \cdot x + b$

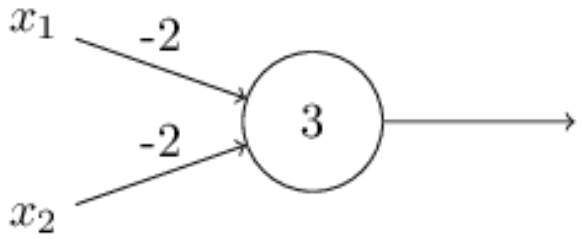


This kind of function is called a **step function**.

1.1 Expressiveness of Perceptron

- Simple perceptrons are still useful (e.g. for logic gates). The one below computes/models the logical function NAND (= not(AND)).

$$p\ NAND\ q \equiv \overline{p\ AND\ q}$$
$$\equiv not(p\ AND\ q)$$

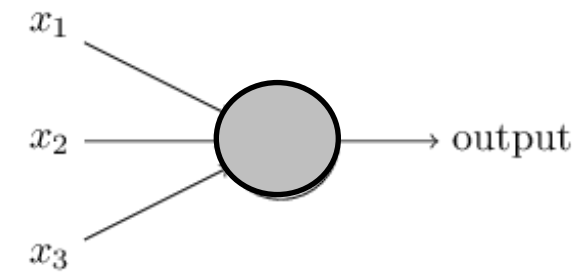


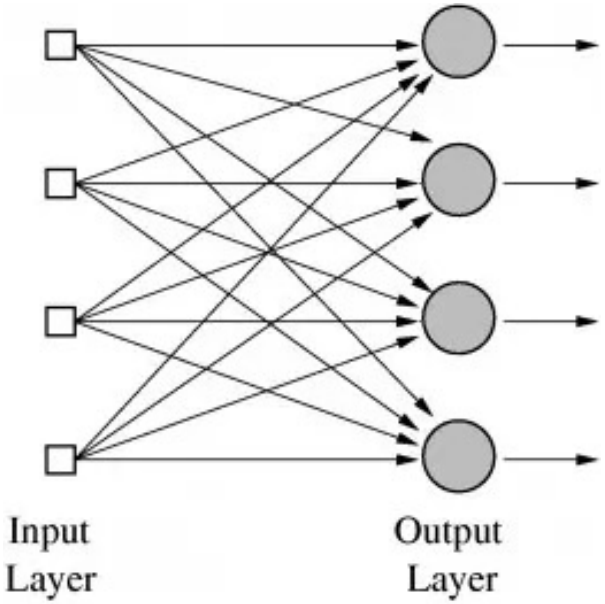
p	*q*	*p NAND q*
1	1	0
1	0	1
0	1	1
0	0	1

x1	*x2*	*perceptron*
1	1	(-2)*(1)+(-2)*(1)+3=-1 --> **0**
1	0	(-2)*(1)+(-2)*(0)+3=1 --> **1**
0	1	(-2)*(0)+(-2)*(1)+3=1 --> **1**
0	0	(-2)*(0)+(-2)*(0)+3=3 --> **1**

https://en.wikipedia.org/wiki/Artificial_neural_network

- However, perceptrons with a **single layer** (i.e., no hidden layer) can only model ***linearly separable*** functions (even with multiple output nodes).

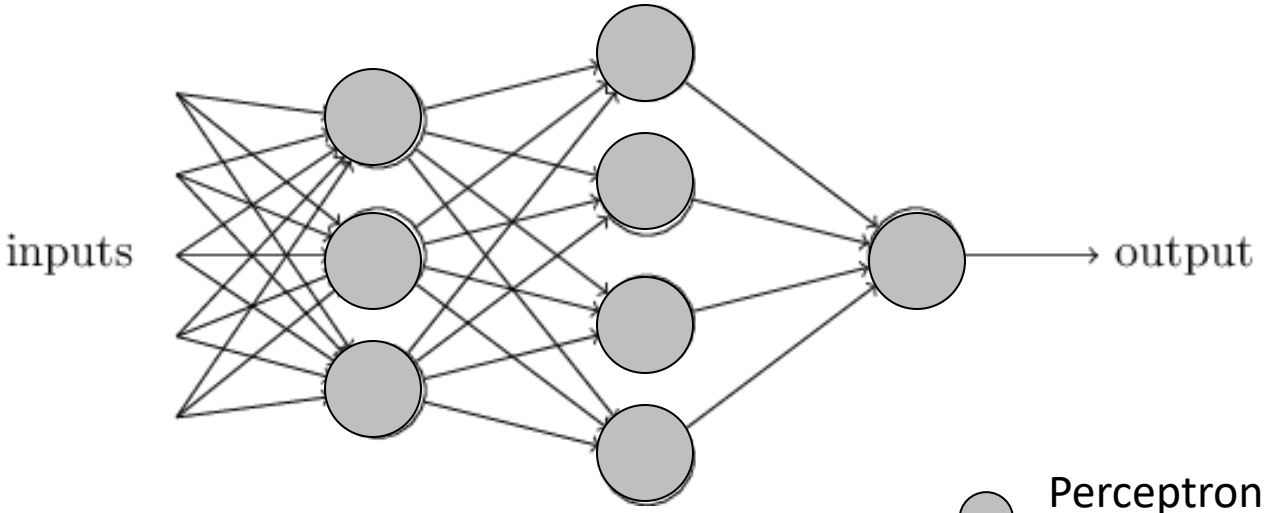




https://en.wikipedia.org/wiki/Artificial_neural_network

Multi-Layer Perceptron (MLP)

- A single perceptron is too simple and cannot model complex functions. But if we connect perceptrons into a **network**, we can model a wide range of functions. Perceptron networks that are formed in layers are called a **Multi-Layer Perceptron** (MLP).



https://en.wikipedia.org/wiki/Artificial_neural_network

- MLPs can compute some ***non-linear functions such as XOR***. But there are limitations on the classes of functions MLPs can compute – MLPs cannot compute all/any functions.
- In addition, perceptrons have a few other limitations:
  - Perceptron's output is not flexible because it is either 1 or 0 (and nothing between).
  - Also the output is not smooth: only a small change in the weighted sum near the threshold changes the output drastically (to/from 1/0). This characteristics makes it difficult to develop a learning algorithm.

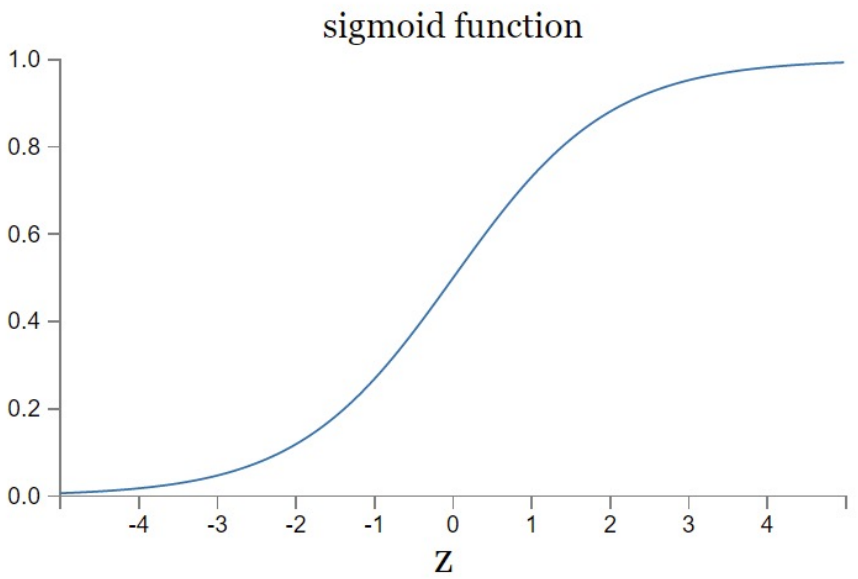
https://en.wikipedia.org/wiki/Artificial_neural_network

2. Sigmoid Neurons

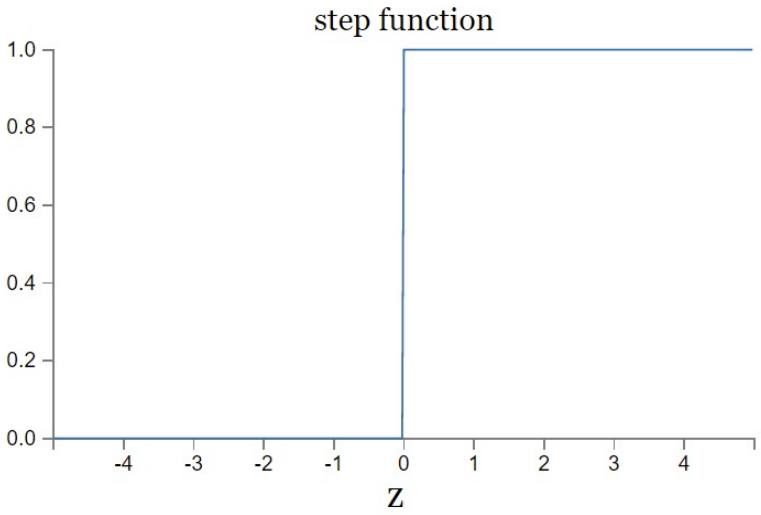
- To solve the limitations of perceptrons, one idea to overcome those is to apply the **sigmoid/logistic** function after the dot product between weights and inputs (then plus bias).
- The sigmoid function is continuous and **differentiable**. It is also a **non-linear** function.

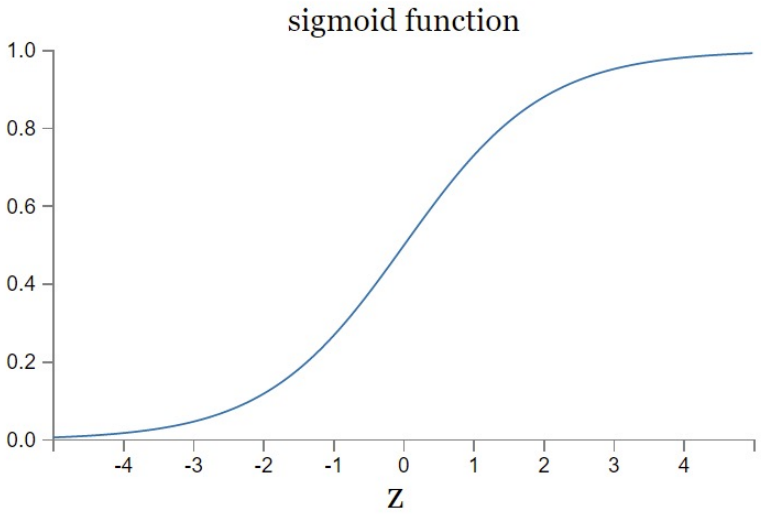
$$\text{output} = \boldsymbol{\sigma}(\boldsymbol{z}) = \frac{1}{1 + e^{-\boldsymbol{z}}}$$

$$\text{where } \boldsymbol{z} = \sum_i w_i \cdot x_i + b$$

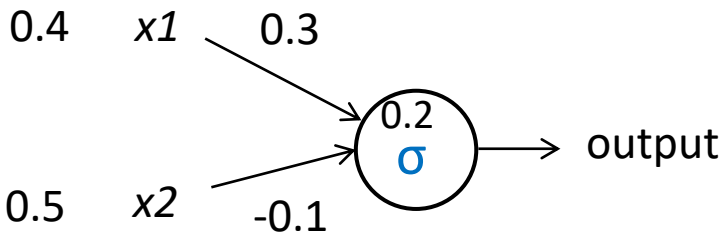


- Now with the introduction of Sigmoid function, we have developed the notion of **activation function** – an additional function applied to the weighted sum (**z**), before producing the output.
  - The activation function of perceptron is a '**step function**' mapping z (-∞ to ∞) to exactly either 1 or 0.
  - The **sigmoid** function *squashes* z (-∞ to ∞) in the range of [0, 1] as a real value.



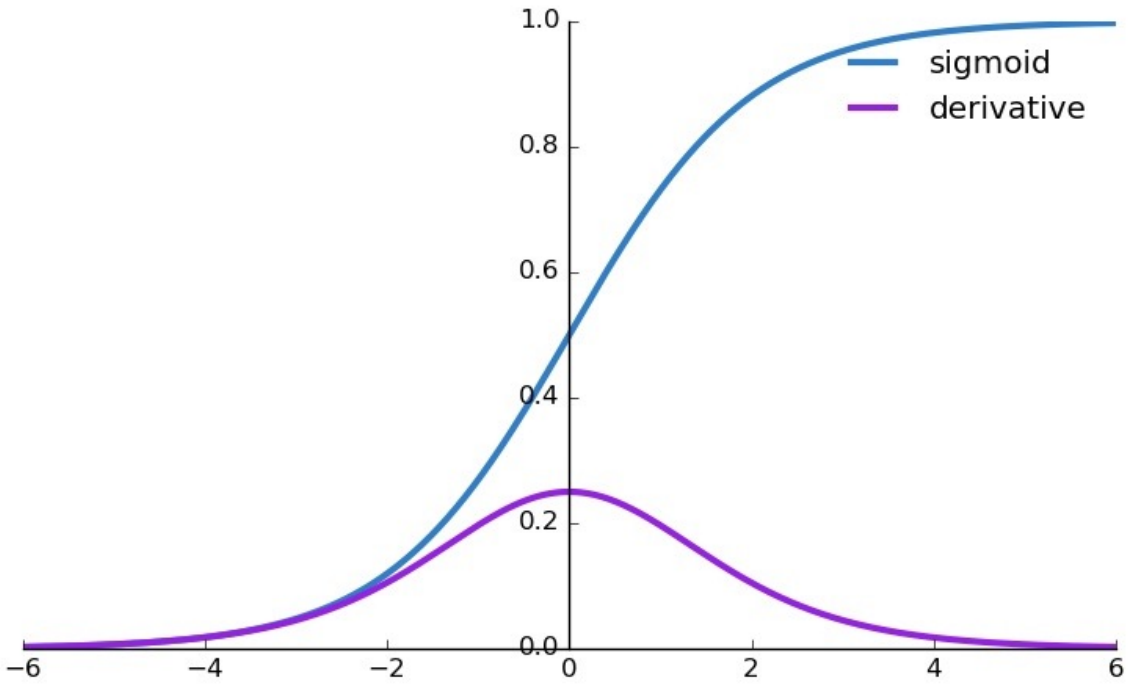


Exercise: Consider the following sigmoid neuron. Note the number inside indicates a bias. When the input of <x1, x2>=<0.4, 0.5> is presented, what would be the output?



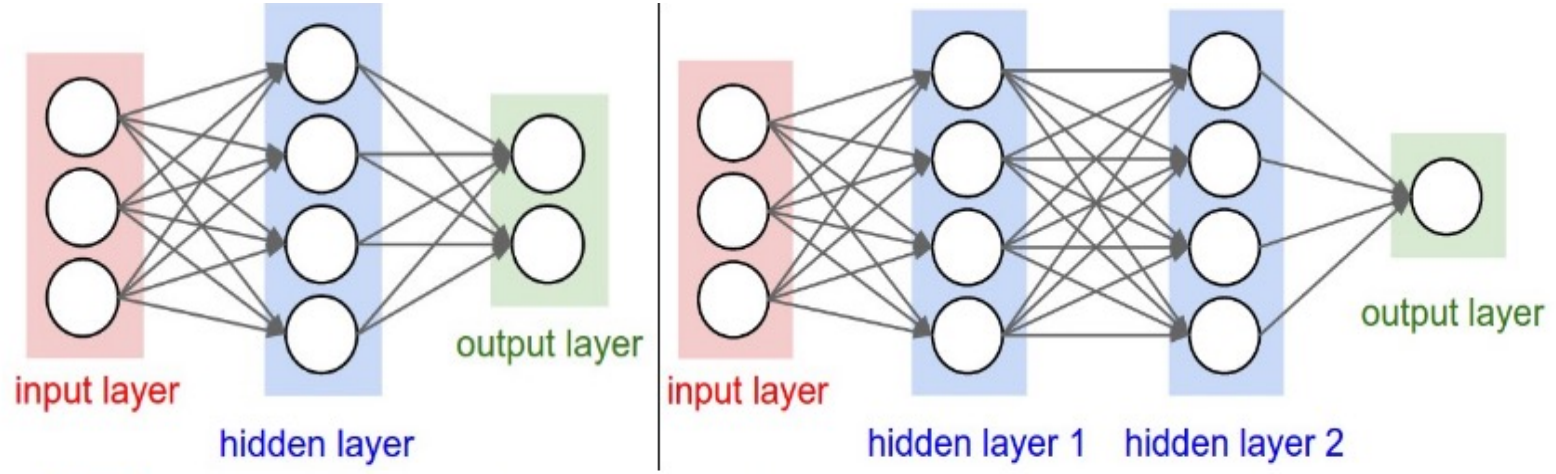
- **(*)** Note: the derivative of the sigmoid function $\sigma'(z)$ is:

$$\sigma'(z) = \sigma(z) \cdot \big(1 - \sigma(z)\big)$$



3. Neural Network Architecture

- "***N-layer*** neural **network**" – By naming convention, we do NOT include the input layer because it doesn't have

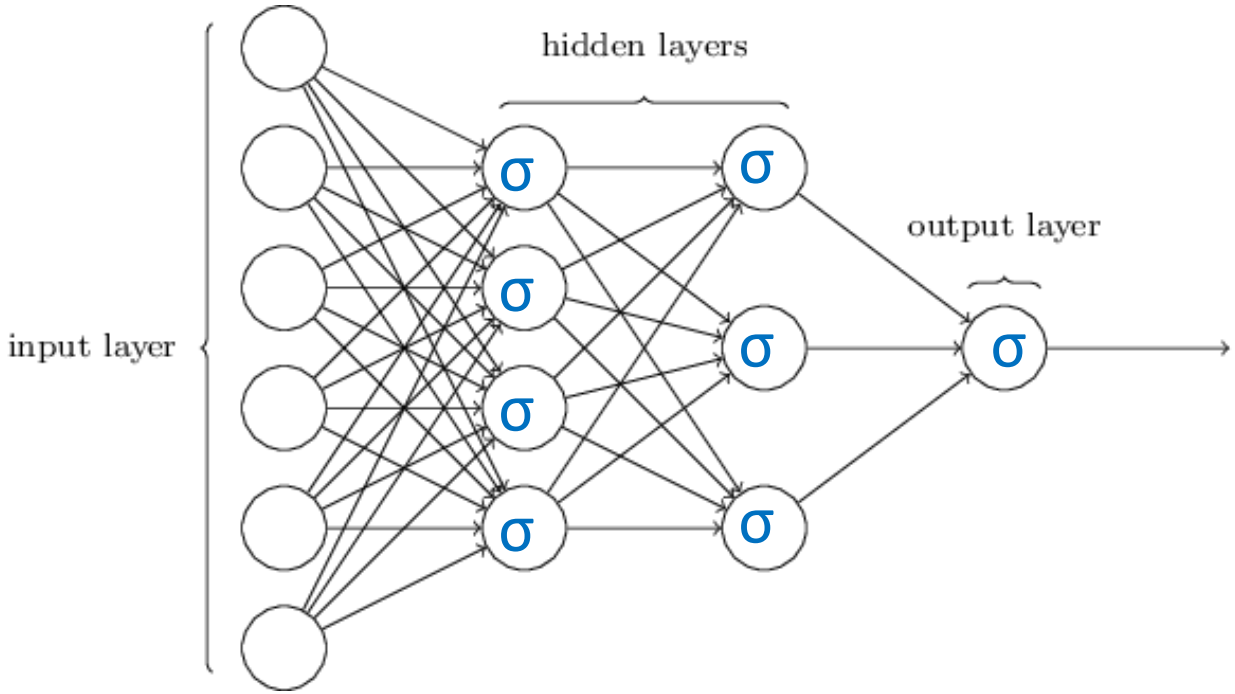


Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

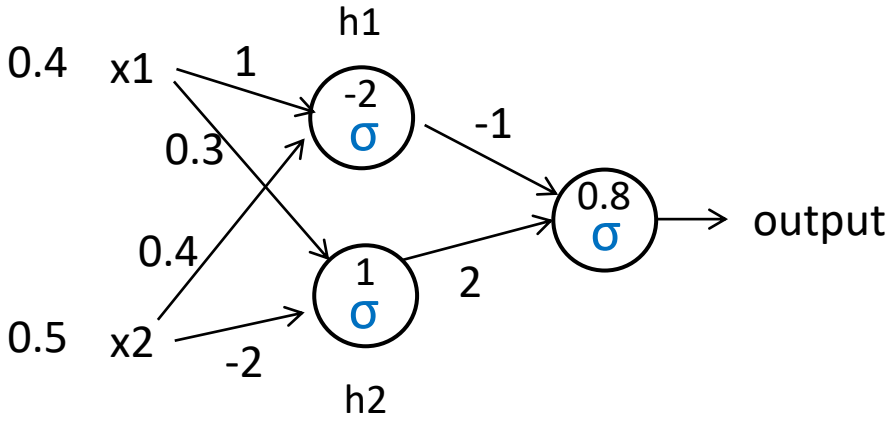
This type of network is called "**Feedforward**".

https://en.wikipedia.org/wiki/Artificial_neural_network

- Note: Every node (except for ones in the input layer) produces an output based on the activation function. For instance, if the activation function is sigmoid (**σ**),



Exercise: Consider the following two-layer neural network with sigmoid activation function. Note that each circle node applies the sigmoid function, and the number inside a circle indicates a bias for the node. If the input <0.4, 0.5> presented, what would be the output?



Now, **how to decide on the network architecture** – number of nodes, hidden layers, etc.?

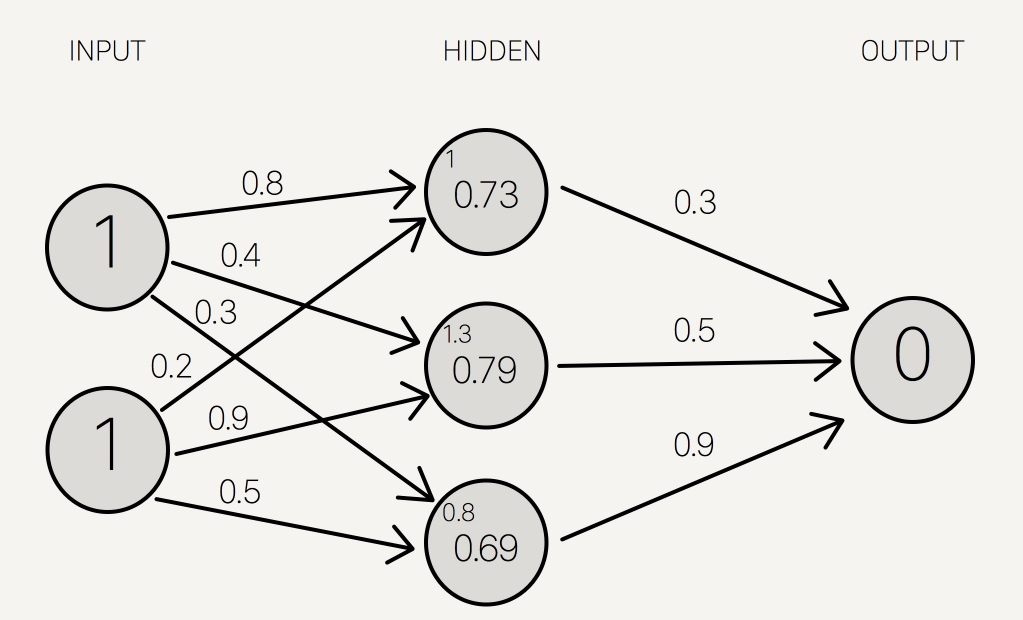
- Number of nodes on the input and output layers are dependent on the dataset – X and Y.

X				Y
Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	Iris setosa
4.9	3.0	1.4	0.2	Iris setosa
7.0	3.2	4.7	1.4	Iris versicolor
6.4	3.2	4.5	1.5	Iris versicolor
6.3	3.3	6.0	2.5	Iris virginica
5.8	3.3	6.0	2.5	Iris virginica

- Number of **hidden layers and nodes** are up for experimentation – trial and error.
- But note the ***trade-off***: Complex models/networks (with many hidden layers and/or nodes) can solve complex problems, but the computation slows down (because of the large number of parameters/weights to learn). Also complex models are prone to ***overfitting***.

4. Neural Network Learning

- Learning in NNs is to **learn the weights and biases** in the network.



- NN learning algorithms *start with small random weights/biases* and **iteratively** adjust them – as the network performance gradually increases.

https://patents.stackexchange.com/questions/18965/do-the-weights-of-a-neural-network-get-any-kind-of-ip-protection

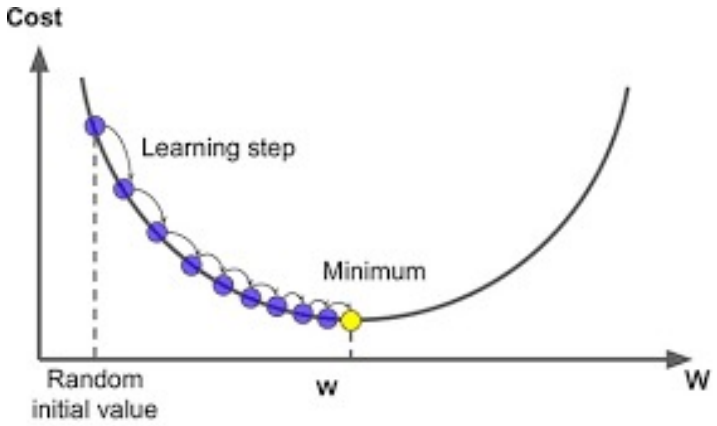
4.1 Gradient Descent

- To learn weights and biases, we compare the **current network output** against the **desired output** indicated in the dataset and try to **minimize the error**/difference.
- Functions that quantify the error are called a **Cost function** (or an error/loss/objective function).
- There are many cost functions used in NN. The most common one is the quadratic, ***mean squared error*** (MSE), as in regression:

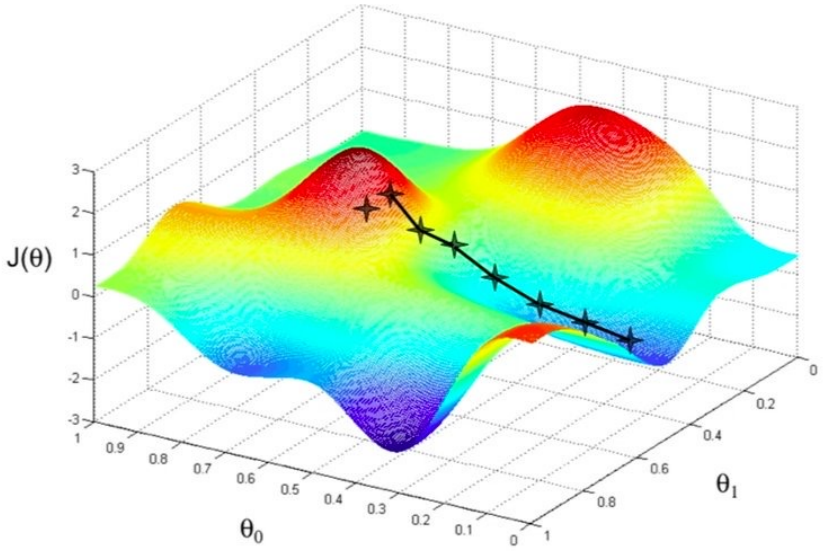
$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

where w is the weights, b is the biases, n is the number of instances in the dataset, y(x) is the desired output for the instance x and a is the current network output for x.

- To minimize the mean squared error, we use a technique called ***gradient descent***.
- The formula for the cost function forms a surface. Since we want to minimize the error, we want to go *down* the surface – to wit, we use **gradient** (the first derivative of the cost function) and try to descend by the gradient to the **global minimum**.



2D view of an example error surface



3D view of an example error surface

- The gradient is the multivariate generalization of **derivative**.
- The (first) **derivative of a function** y = f(x) of a variable x is a measure of the rate at which the value of y changes with respect to the change of the variable x. It is called *"the derivative of f with respect to x"*.
- If the function is plotted on a graph, the derivative of the function is the **slope**.
- For a given function f, the **derivative of f** (notated as **f'** (*f-prime*)) **is also a function**. This function gives the slope for each given point of x.

e.g. For $f(x) = x^2$, $f'(x) = 2x$

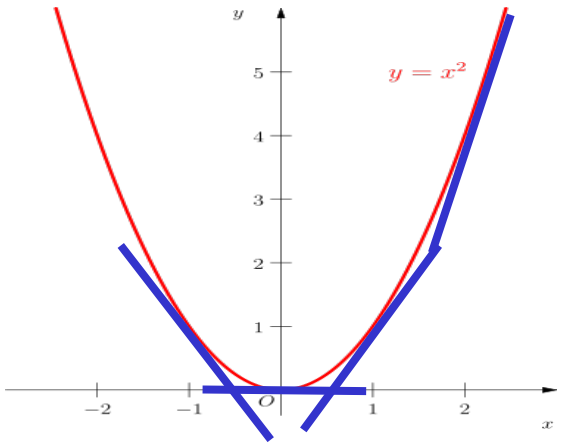
$f'(-1) = -2$

$f'(0) = 0$

$f'(1) = 2$

$f'(2) = 4$

$f'(3) = 6$



4.2 Gradient Descent Learning

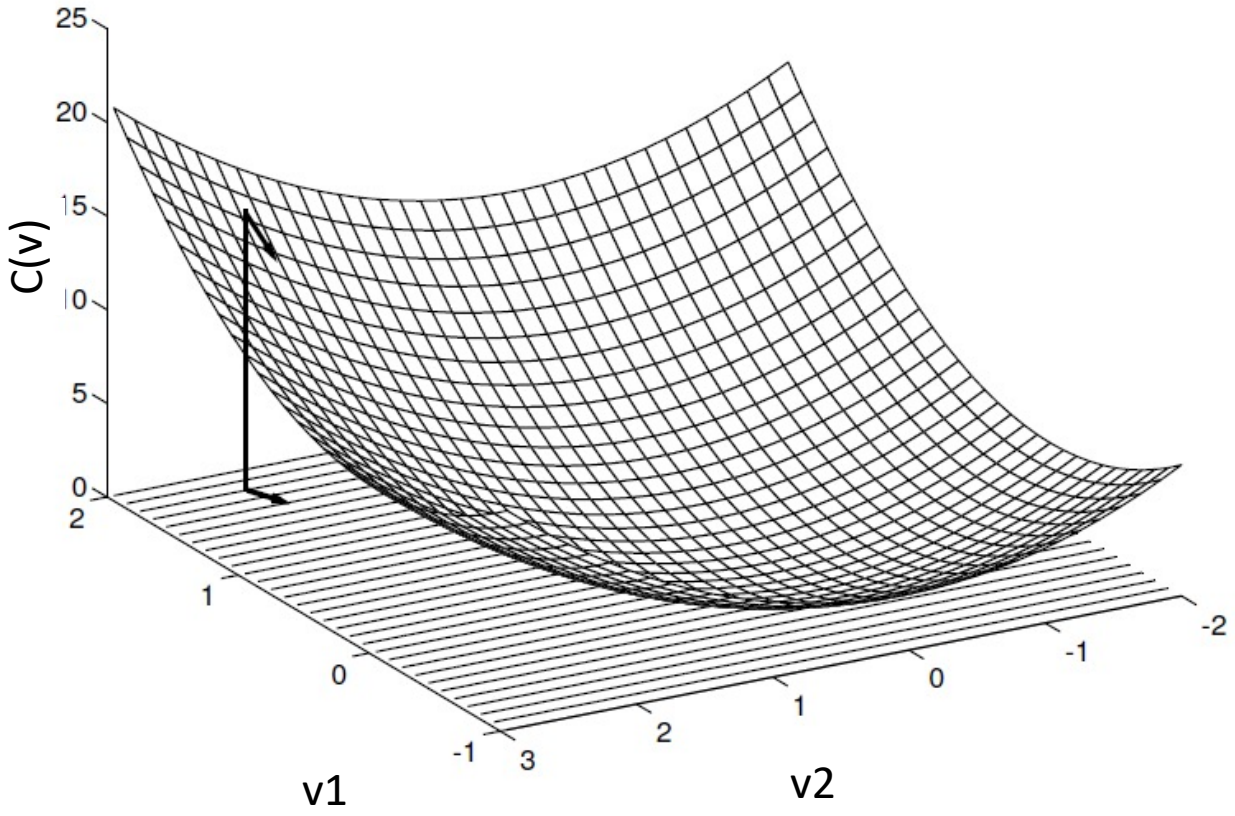
We can define the gradient of the cost function as follows:

- Let *C(v)* is the cost function (MSE) where *v* is a vector of variables (weights and biases, $\vec{v} = [v_1, \cdots, v_k]$). The gradient of *C*, notated ∇C, is a vector:

$$\nabla C(\vec{v}) = \left[\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \cdots, \frac{\partial C}{\partial v_k}\right]$$

where ∂ denotes a partial derivative, and $\frac{\partial C}{\partial v_i}$ is the partial derivative of C with respect to the variable v_i.

[Example] Below is the surface of the cost function (with two variables v1 and v2). We want to *descend* this surface to the global minimum – so we add the **negative** **of the gradient** to **each variable** when we update weights/biases during training.



- We also **update weights/biases by small amount** at a time/iteration during training. The fraction is specified by the hyperparameter **η** (eta) -- the **learning rate**.

$$\Delta\vec{v} = -\boldsymbol{\eta} \cdot \nabla C(\vec{v})$$

- Now we have the '**variable update rule**' in the learning algorithm:

$$\vec{v} = \vec{v} + \Delta\vec{v}$$

which equates to

$$\vec{v} = \vec{v} + -\boldsymbol{\eta} \cdot \nabla C(\vec{v}) = \vec{v} - \boldsymbol{\eta} \cdot \nabla C(\vec{v})$$

Back to the gradient for each variable v_i. It can be derived by calculus.

$$\frac{\partial C}{\partial v_i} = \frac{\partial}{\partial v_i}\left[\frac{1}{2n}\cdot\sum_d \| y_d - a_d \|^2\right]$$

$$= \frac{\partial}{\partial v_i}\left[\frac{1}{2n}\cdot\sum_d (y_d - a_d)^2\right]$$

$$= \frac{1}{2n}\cdot\sum_d \left[\frac{\partial}{\partial v_i}(y_d - a_d)^2\right]$$

$$= \frac{1}{2n}\cdot\sum_d \left[(2)\cdot(y_d - a_d)\cdot\frac{\partial}{\partial v_i}(y_d - a_d)\right]$$

$$= \frac{1}{n}\cdot\sum_d \left[(y_d - a_d)\cdot\frac{\partial}{\partial v_i}(y_d - a_d)\right]$$

$$= \frac{1}{n}\cdot\sum_d \left[(y_d - a_d)\cdot\frac{\partial}{\partial v_i}(-a_d)\right]$$ (cont.)

where $d \in D$ (the dataset) and $|D| = n$.
y_d denotes the desired output of the instance *d*, and a_d denotes the network output for *d*.

C is the cost function, and v_i is a variable of interest.

Since the activation function is Sigmoid (**σ**), the network output $a_d = \sigma(z)$. Continuing from the previous slide:

$$\frac{\partial C}{\partial v_i} = \frac{1}{n} \cdot \sum_d \left[(y_d - \sigma(z)) \cdot \frac{\partial}{\partial v_i} \left(-\sigma(z) \right) \right]$$

$$= \frac{1}{n} \cdot \sum_d \left[-(y_d - \sigma(z)) \cdot \{\boldsymbol{\sigma'(z)}\} \cdot \frac{\boldsymbol{\partial}}{\boldsymbol{\partial v_i}} (\boldsymbol{z}) \right] \quad \ldots by\ the\ \textbf{\textit{chain rule}}$$

$$= \frac{1}{n} \cdot \sum_d \left[-\left(y_d - \sigma(z)\right) \cdot \left\{ \sigma(z) \cdot \left(1 - \sigma(z)\right) \right\} \cdot \frac{\partial}{\partial v_i} (z) \right] \ldots by\ (*)$$

$$= \frac{1}{n} \cdot \sum_d \left[(\sigma(z) - y_d) \cdot \left\{ \sigma(z) \cdot \left(1 - \sigma(z)\right) \right\} \cdot \frac{\partial}{\partial v_i} \left(\sum_i v_i \cdot x_i + b \right) \right]$$

where

$$\frac{\partial}{\partial v_i} \left(\sum_i v_i \cdot x_i + b \right) = \begin{cases} \boldsymbol{x_i} & \text{if } v_i \text{ is a weight} \\ 1 & \text{if } v_i \text{ is a bias} \end{cases}$$

4.3 Batch vs. Online vs. Stochastic

- Notice the gradient derived in the last slide is the average of gradients from the instances in the training set $\left(\nabla C = \frac{1}{n}\sum_d [..]\right)$. In this scheme, called the ***batch*** mode, weights and biases are updated only after **all** instances (n) are processed.
- Although it is the true form of gradient descent learning, when the training set is very large, the batch mode can take a long time, and learning thus occurs slowly.
- One alternative is to speed up learning is to update weights and variables after **each/one** instance. This scheme is called the ***online*** mode ($\nabla C_x = [..]$).
- However, frequent updates cost high computation time.

- A good alternative is **mini-batch**. It is a hybrid of batch and online modes. The idea is to estimate the gradient ∇C by using the average of the gradients of a **small sample** of ***m*** (<= *n*) randomly chosen instances. This mode is called ***stochastic*** **gradient descent.**
- We assume the average gradient of the mini-batch would approximate the gradient of the whole dataset:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

- Then during training, we **update the weights and biases** after every mini-batch:

$$\vec{v} = \vec{v} - \eta \cdot \nabla C(\vec{v}) = \vec{v} - \left[\frac{\eta}{m} \cdot \sum_{j=1}^{m} \nabla C_{X_j}(\vec{v})\right]$$

Where $\nabla C_{X_j}(\vec{v})$ is a gradient vector for a **single instance X_j** in a mini-batch, and

$$\nabla C_{X_j}(\vec{v}) = \left[\frac{\partial C_{X_j}}{\partial v_1}, \frac{\partial C_{X_j}}{\partial v_2}, \ldots, \frac{\partial C_{X_j}}{\partial v_k}\right]$$

Then for each variable v_i (and the activation is sigmoid σ),

$$\frac{\partial C_{X_j}}{\partial v_i} = \left(\sigma(z) - y_{X_j}\right) \cdot \left\{\sigma(z) \cdot \left(1 - \sigma(z)\right)\right\} \cdot \frac{\partial}{\partial v_i}(z)$$

where

$$z = \sum_i v_i \cdot x_i + b \quad \text{and} \quad \frac{\partial}{\partial v_i}(z) = \begin{cases} x_i & \text{if } v_i \text{ is a weight} \\ 1 & \text{if } v_i \text{ is a bias} \end{cases}$$

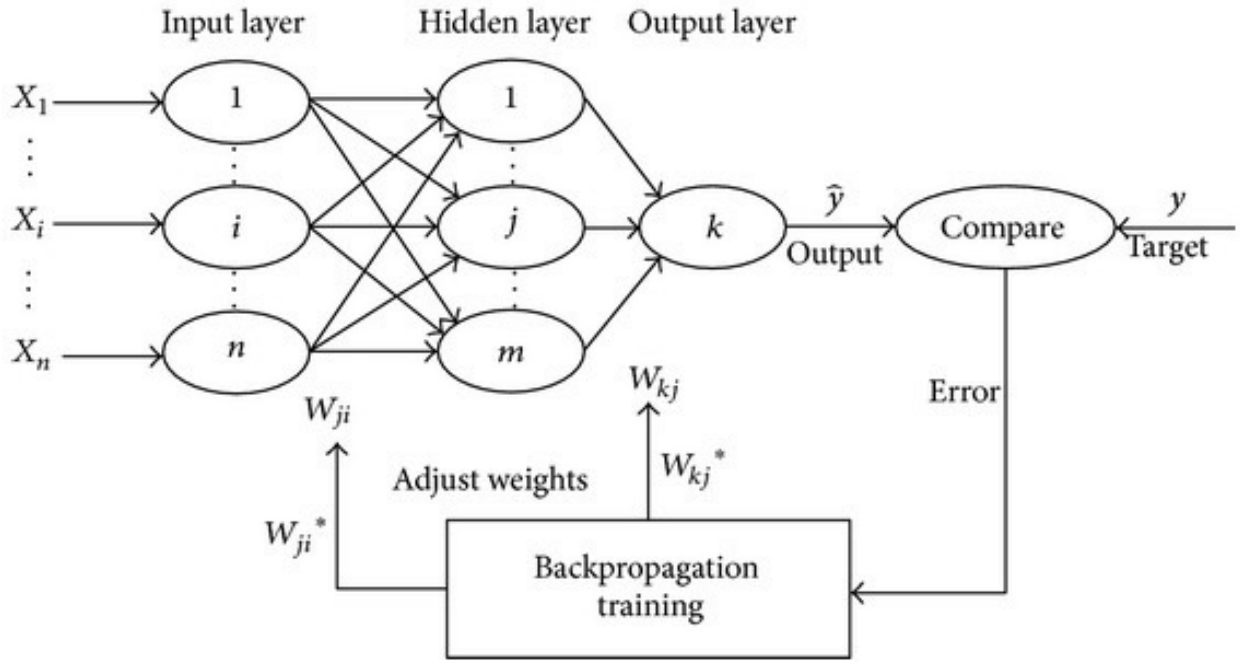
Epochs

- Mini-batches are essentially randomly partitioned subsets of the training data.
- Training of the whole training data one time through is called an ***epoch***. Given *n* and *m*, there will be $\left\lceil \frac{n}{m} \right\rceil$ number of mini-batches, thus that many times of variable updates in an epoch.

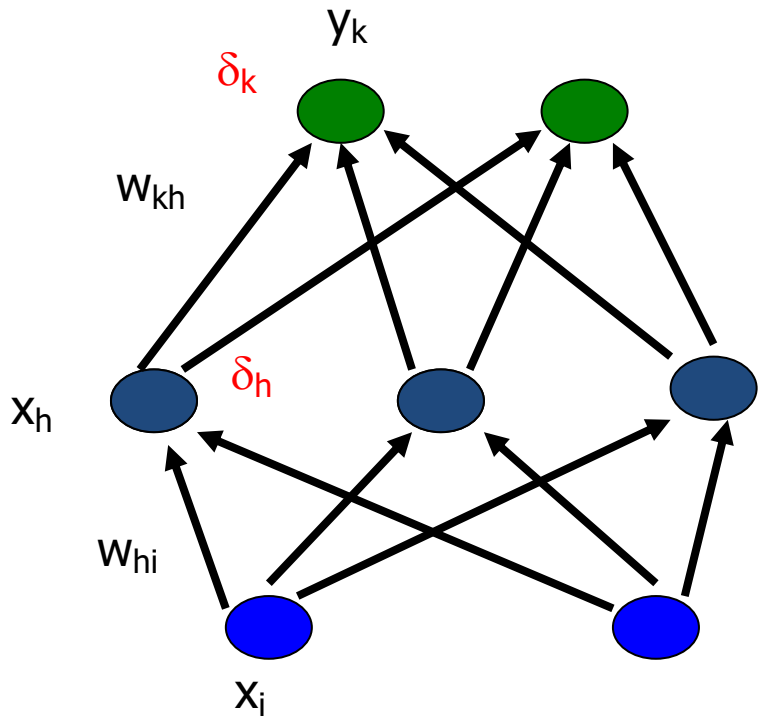
- **Mini-batch sizes** are often chosen as a power of 2, i.e., 16, 32, 64, 128, 256 etc.
- A good default for batch size might be 32.
- It is also advised to make sure that the minibatch fits in the CPU/GPU.

4.4 Backpropagation Algorithm

- We derived a variable update rule from the cost function – which presumably applies to variables connected to the output layer. But what about the **weights and variables on the hidden layers**???
- The Backpropagation Algorithm addresses that:



- The aim of the Backpropagation algorithm is to propagate the error/cost from the output layer through the hidden layer(s).



(2) Backward step:
Compute the error and propagate it from output to hidden layer

(1) Forward step:
Propagate activation from input to output layer

- We will study Backprop in depth in the next lecture.