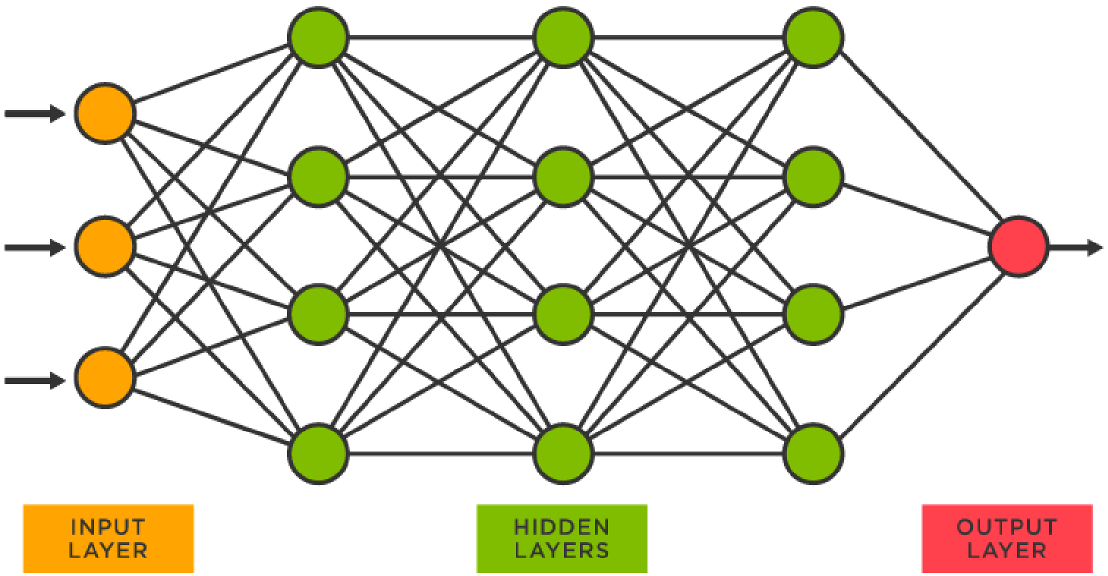
SUPPORT VECTOR MACHINES & NEURAL NETWORKS

LECTURE 8 – ARTIFICIAL NEURAL NETWORKS

A. Basic structure of neural networks
   - Neuron, activation function, perceptron, feedforward NN
B. Backpropagation and learning
   - Loss/reward function, online vs. batch learning and algorithms
C. Multi-layer neural networks and deep learning
   - Scale, feature and computation, ReLU and SGD
D. Radial basis function neural network (RBFN)
E. Convolutional neural network (CNN)

Artificial neural networks

- An artificial neural network (ANN or NN in short) is a mathematical/computational model that mimics the operations of human brains to create artificial intelligence through some learning algorithms.



< tibco.com>

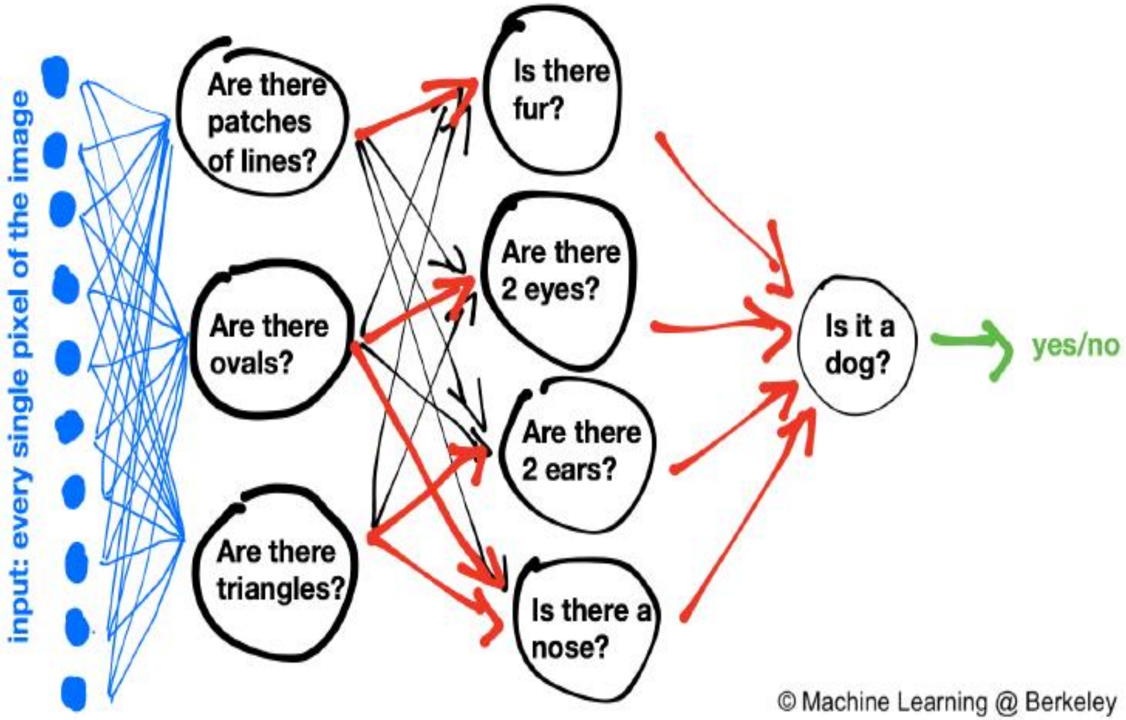
Recent advance in deep learning

- Deep learning for *computer vision*, *image procession, pattern recognition, approximate reasoning, etc.*

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcRAqwxA0i0s2cvaWXxZRPV5Y53a4vOyjtHURQ&usqp=CAU

Recent advance in deep neural network

- Identify a dog in a photo (Machine Learning Crash Course: Part 3 - ML@B Blog)



© Machine Learning @ Berkeley

- Pixels line segments distinct features judgement

Some of the key works in NN

- Alexander Bain (1873 – Mind and Body) and William James (1890 – The Principles of Psychology) uncovered preliminary theoretical bases of “thoughts and body activities are from interactions of neurons (via electric flows) in the brain”.
- Tested by C. S. Sherrington (1898) that led to the concept of habituation.
- Warren McCulloch and Walter Pitts (1943) built the first “threshold logic” computational model.
- The concept of NN (B-type unorganized machines) had first been officially raised by Alan Turing in his 1948 paper.

Some of the key works

- F. Rosenblatt (1958) created the first "perceptron"/artificial neuron. (Some called him "father of deep learning").
- Paul Werbos (1974) PhD Dissertation at Harvard pioneered the concept of "backpropagation".
- J. J. Hopfield (1982) introduced one classical type of artificial neural network called recurrent Hopfield network.
- D. E. Rumelhart and J. McClelland (1986) provided a full exposition on the use of connectionism in computers to simulate neural processes.
- G.E. Hinton, S. Osindero, and Y. Teh (2006) proposed a fast learning algorithm for deep belief nets.

Mathematical foundation

- Key function: uncover a non-explicit input-output relation.

- Universal approximation: (From Wikipedia)

Universal approximation theorem: Let $C(X, Y)$ denote the set of continuous functions from X to Y. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes σ applied to each component of x.

Then σ is not polynomial if and only if for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m), \varepsilon > 0$ there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$$

where

$$g(x) = C \cdot (\sigma \circ (A \cdot x + b))$$

Key function: uncover nonlinear input-output relationship

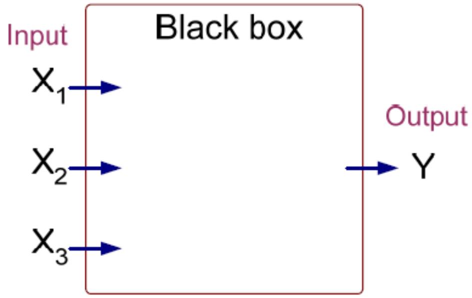
- Basic model of an artificial neural network

data

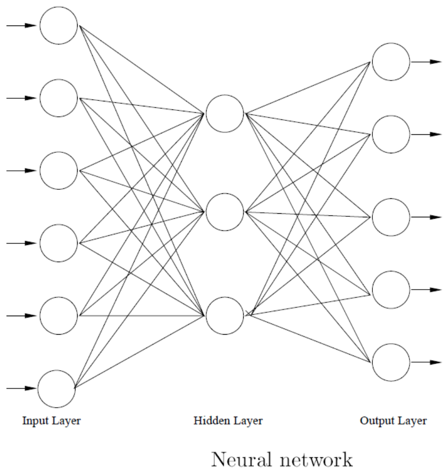
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

relation/function

$$\mathbf{y} = f(\boldsymbol{x}) = ?$$



method/algorithm



- Prediction? Classification?
- Patterns? Universal approximation?

Question

- Basic model of an artificial neural network

data

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

$R(\boldsymbol{x}, \boldsymbol{y}) = ?$ Relationship

$\mathbf{y} = f(\boldsymbol{x}) = ?$ Function

Artificial Neural Networks (ANN)

- Principle:
  - complexity can be embedded in layered simplicity (layered simplicity can generate desired complexity)

Implication:
- the intelligence (computational power) of a neural network comes from properly layered neurons

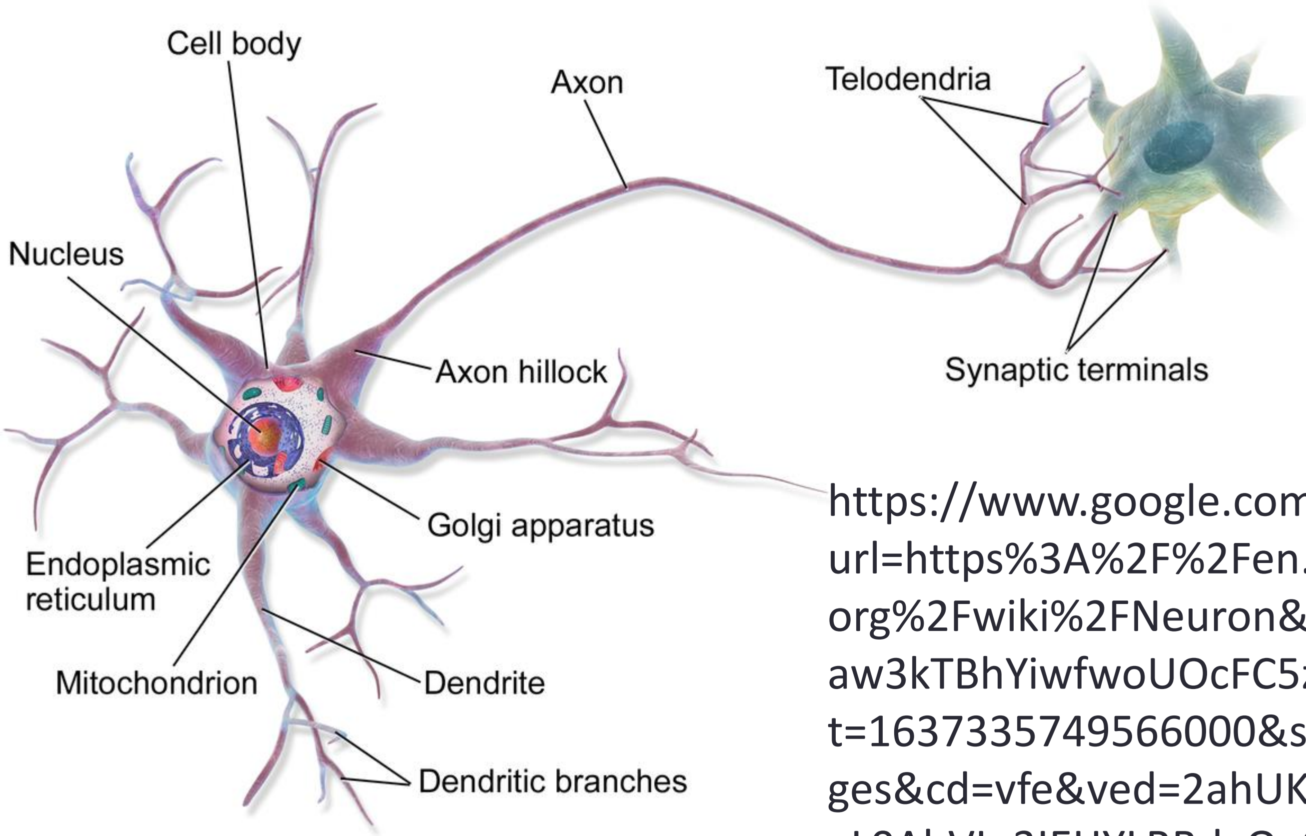
Warren McCulloch and Walter Pitts' work of 1943 ("A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/BF02478259) opened the subject by creating a computational model for neural networks.

Basic concepts of neural networks

- A brain is composed of some network of neurons.
- A typical neuron receives input – either excitation or inhibition – from many other neurons.
- When its net excitation reaches a certain level, the neuron fires.
- The firing is propagated through a branching axon to many other neurons, where it in turn acts as input to those neurons.
- A neuron always computes the same function.
- We learn because the strength of connections between neurons changes.

- Because the strength of the connections between the neurons in the network can change, the relationship of the network's output can change, the relationship of the network's output to its input can be altered by experience.

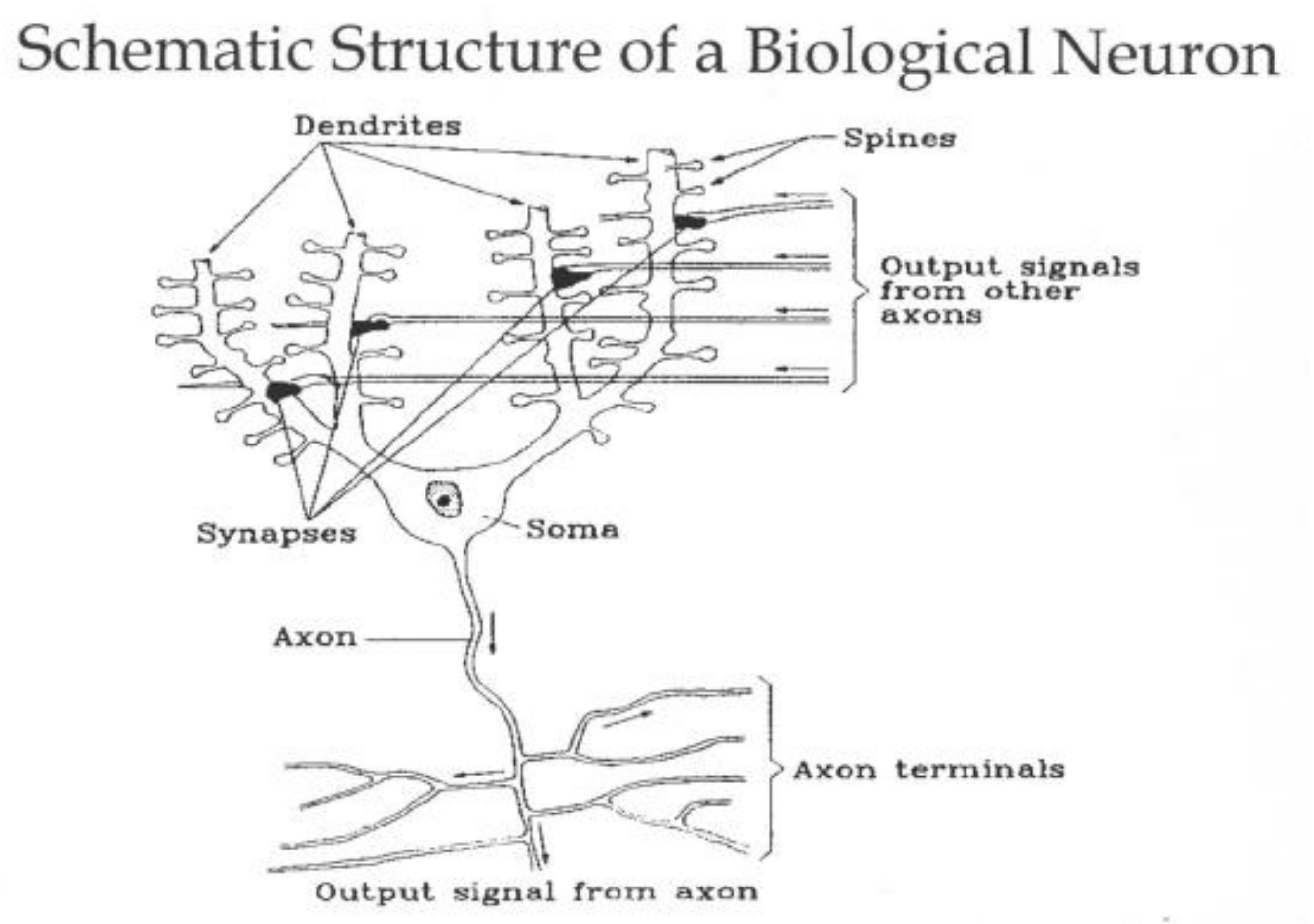
100 billions Neurons in Human Brain



https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FNeuron&psig=AOvVaw3kTBhYiwfwoUOcFC5zmV1K&ust=1637335749566000&source=images&cd=vfe&ved=2ahUKEwi5_Y6PnaL0AhVIo3IEHYLRBdgQr4kDegUIARCzAQ

Artificial neural networks

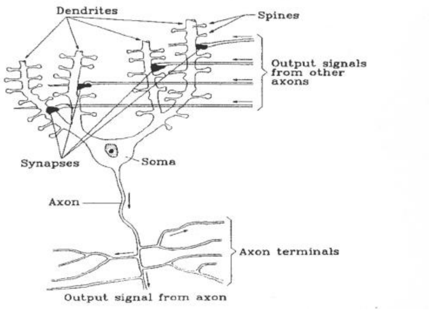
Neuron – the computational element



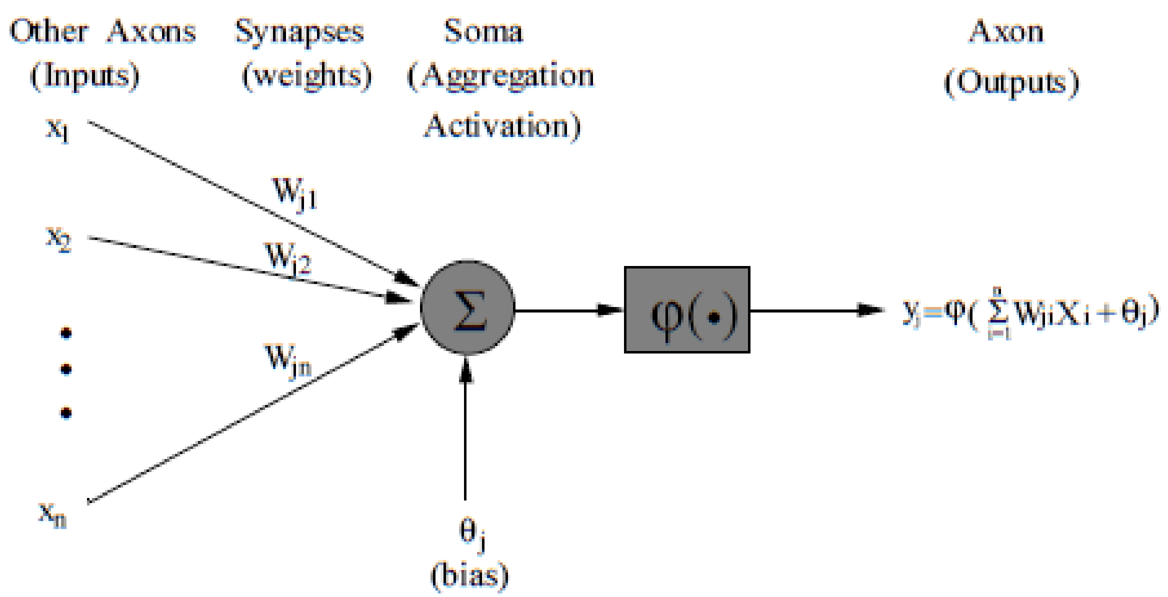
Artificial neural networks

Neuron – the computational element

Schematic Structure of a Biological Neuron



Mathematics of a Conceptual Neuron

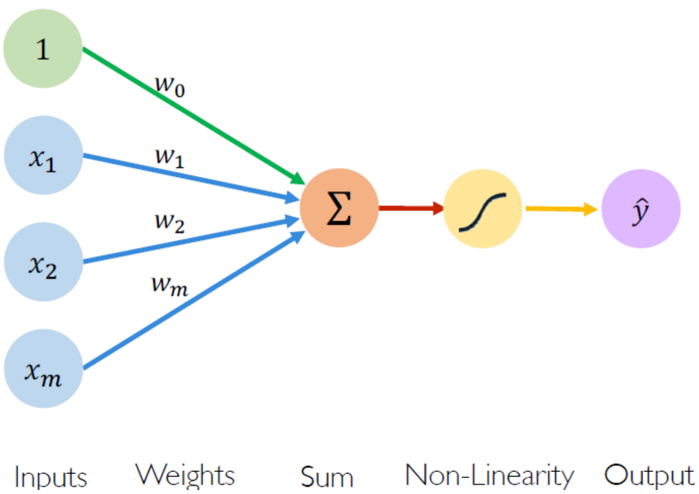


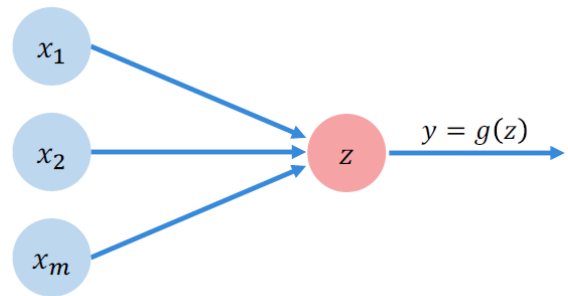
output of neuron j: $y_j = \phi(\boldsymbol{w}_j{}^T \boldsymbol{x} + b_j)$

activation function $\phi(\cdot) : \mathbb{R} \to \mathbb{R}$

Feedforward perceptron

- Simplified – one output (from MIT 6.S191 introtodeeplearning.com)





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

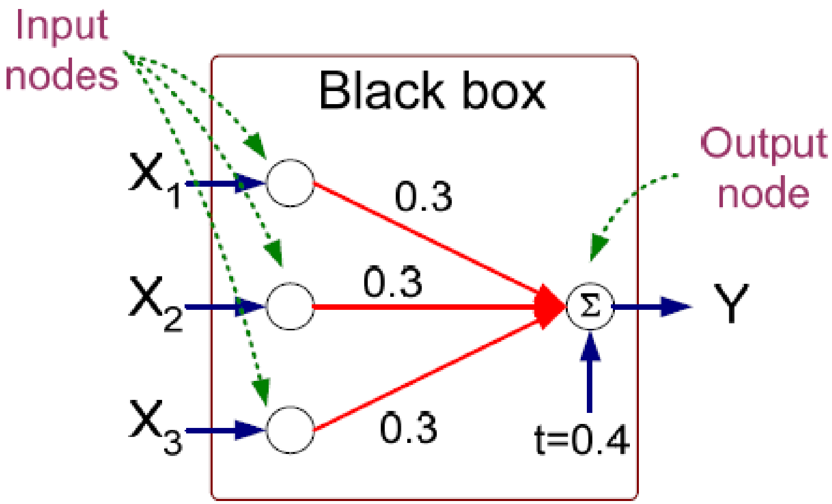
$$y = \phi(\boldsymbol{w}^T \boldsymbol{x} + b)$$

How much can a perceptron do?

- Data

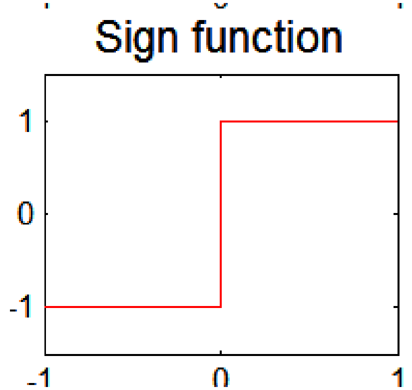
Connections and weights

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



How much can a perceptron do?

- Activation function



$$sign(x) = \begin{cases} +1, & \text{if } x > 0 \\ -1, & \text{if } x \leq 0 \end{cases}$$

Perceptron model

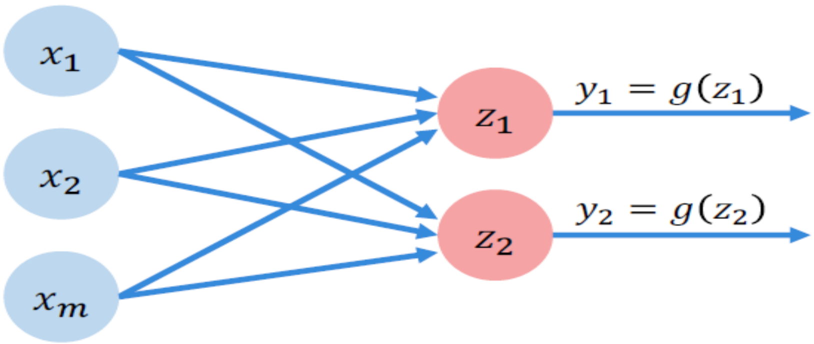
$$y = \phi(\boldsymbol{w}^T \boldsymbol{x} + b)$$
$$= sign\ (0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4)$$

(Work like SVM?)

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

Multi-output perceptron

- Simplified –multiple outputs (from MIT 6.S191introtodeeplearning.com)



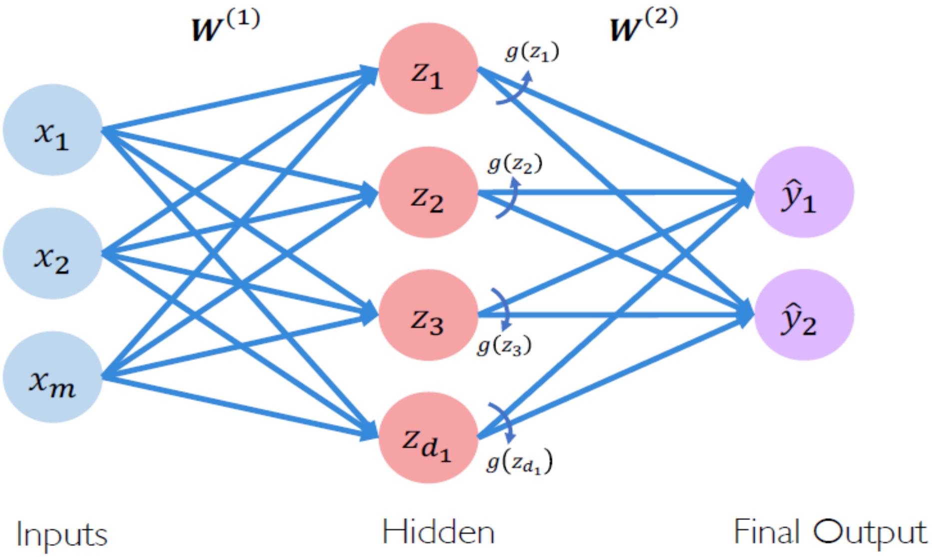
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

$$y_j = \phi(\boldsymbol{w}_j^T \, \boldsymbol{x} + b_j),\ j = 1{,}2 \text{ Type equation here.}$$

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \phi(\boldsymbol{w}_1^T \, \boldsymbol{x} + b_1) \\ \phi(\boldsymbol{w}_2^T \, \boldsymbol{x} + b_2) \end{pmatrix} \triangleq \boldsymbol{\Phi}(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$$

Single hidden layer (shallow) perceptron NN

- Simplified (from MIT 6.S191introtodeeplearning.com)



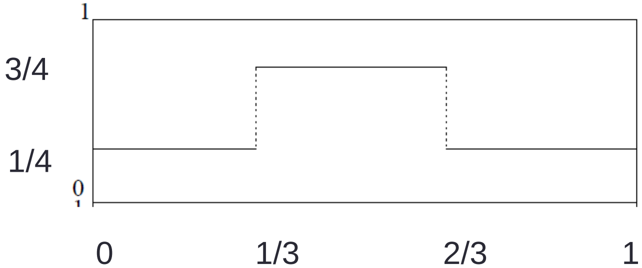
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}, \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$

$$y_j = \phi\left(\left(\boldsymbol{w}_j^{(2)} \right)^T \boldsymbol{\Phi}\left(\left(\boldsymbol{W}^{(1)} \right)^T \boldsymbol{x} + \boldsymbol{b}^{(1)} \right) + b_j^{(2)} \right), \; j = 1,2$$

$$\boldsymbol{y} = \boldsymbol{\Phi}\left(\left(\boldsymbol{W}^{(2)} \right)^T \boldsymbol{\Phi}\left(\left(\boldsymbol{W}^{(1)} \right)^T \boldsymbol{x} + \boldsymbol{b}^{(1)} \right) + \boldsymbol{b}^{(2)} \right)$$

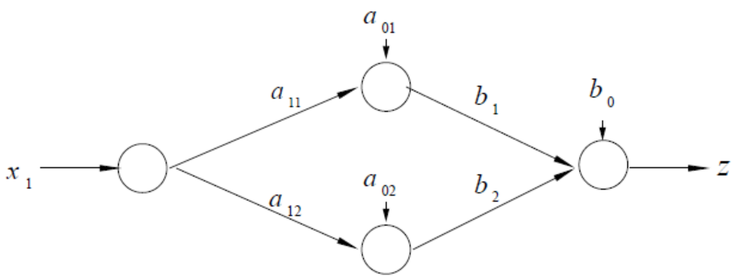
How much can a shallow network do?

- Data



square wave

- Connections & weights



Square Wave Network $b_0 = -3.350$

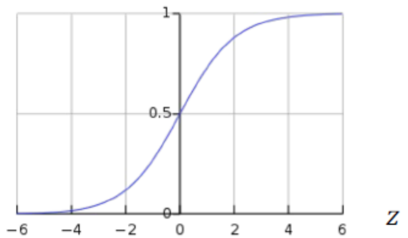
j	a_{0j}	a_{1j}	$\hat{x}_1$	b_j
1	-30	100	0.3	2.225
2	70	-100	0.7	2.225

How much can a shallow network do?

Activation function

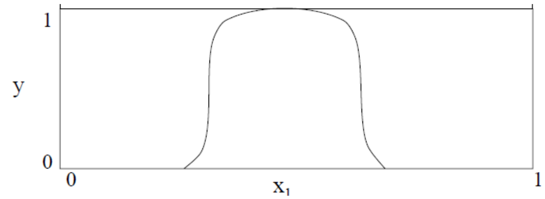
Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

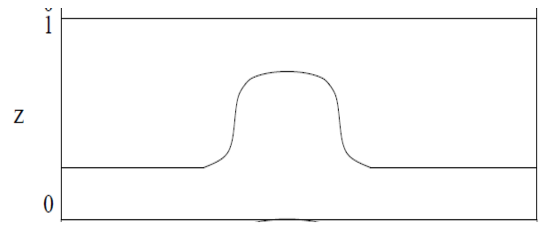


sigmoid function

shallow network model

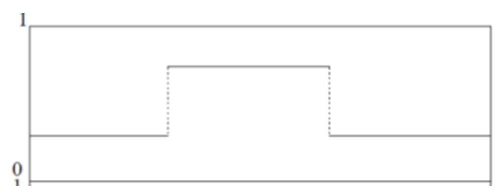


hidden nodes



output node

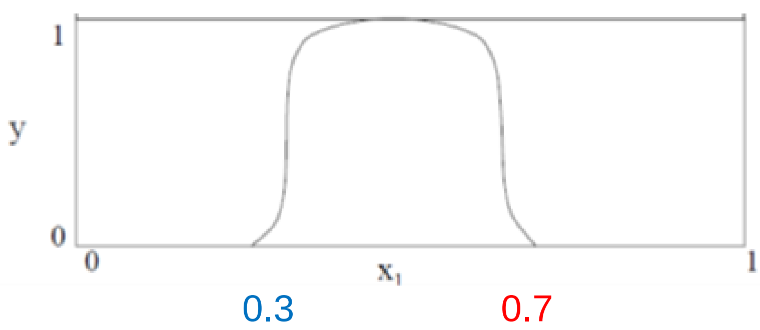
(work like an approximator?)

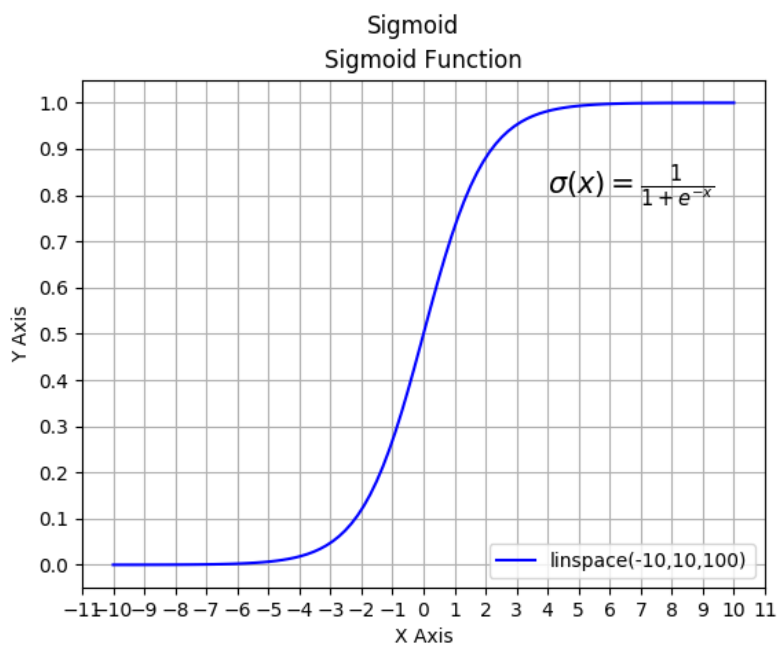


squared wave

Exercise

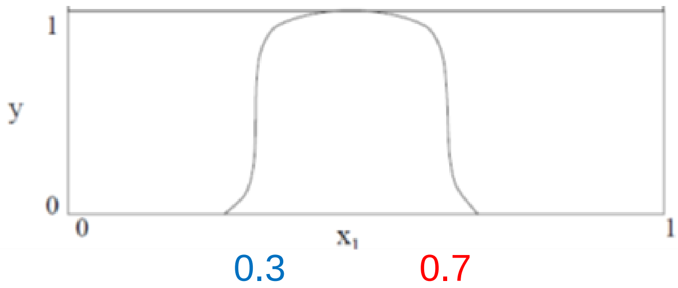
- Sigmoid function value table
- $y_1 = sig\ (100x - 30)$
- $y_2 = sig(-100x + 70)$
- $z = sig\ (2.225y_1 + 2.225y_2 - 3.350)$

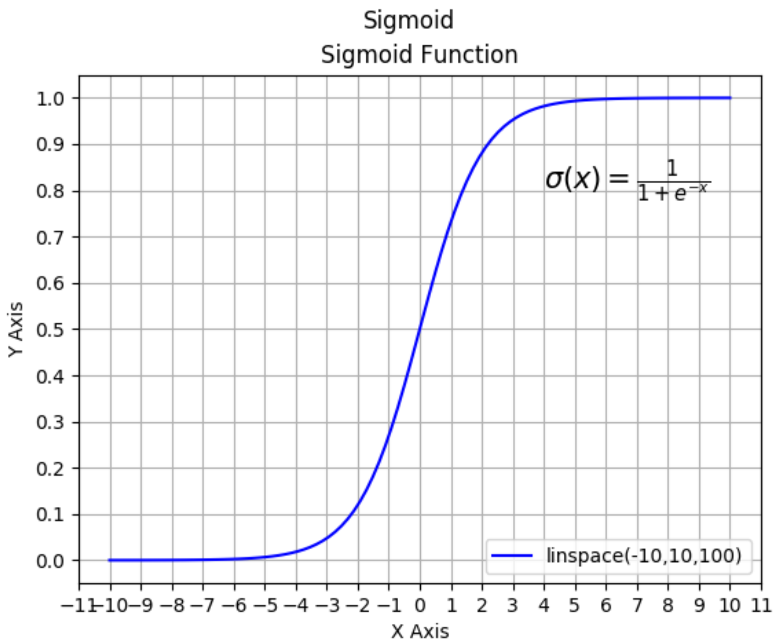


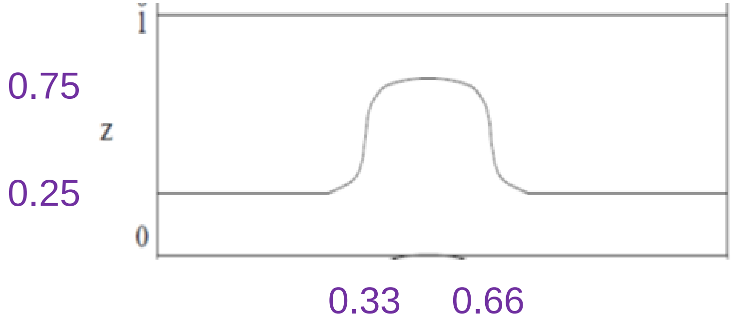


Exercise

- Sigmoid function value table
- $y_1 = sig\ (100x - 30)$
- $y_2 = sig(-100x + 70)$
- $z = sig\ (2.225y_1 + 2.225y_2 - 3.350)$

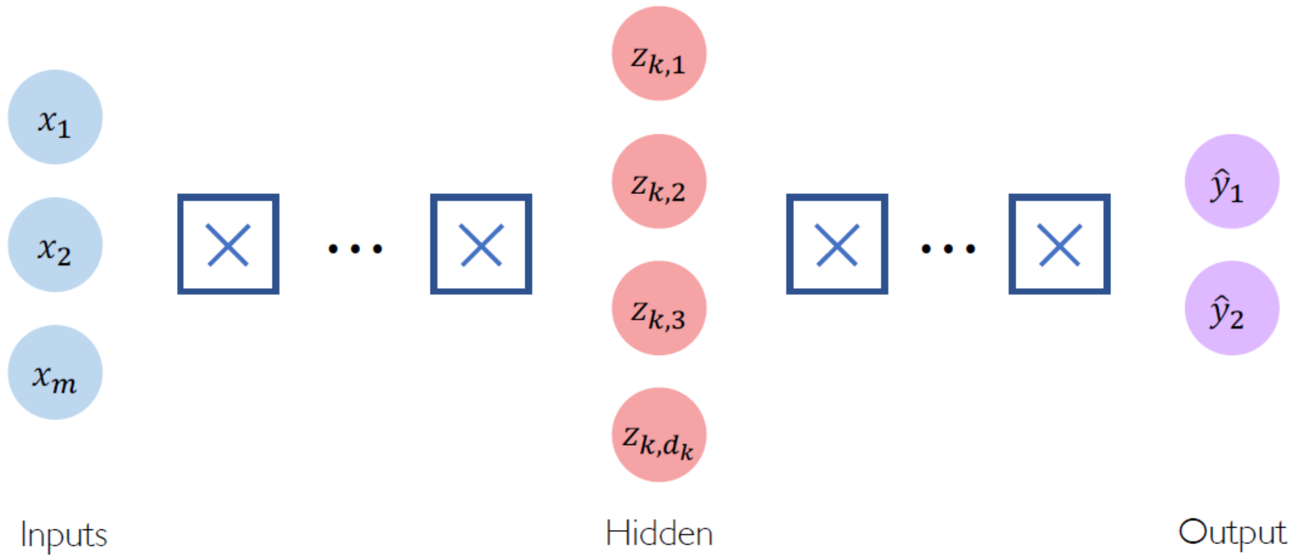






Multi-layer (deep) perceptron NN

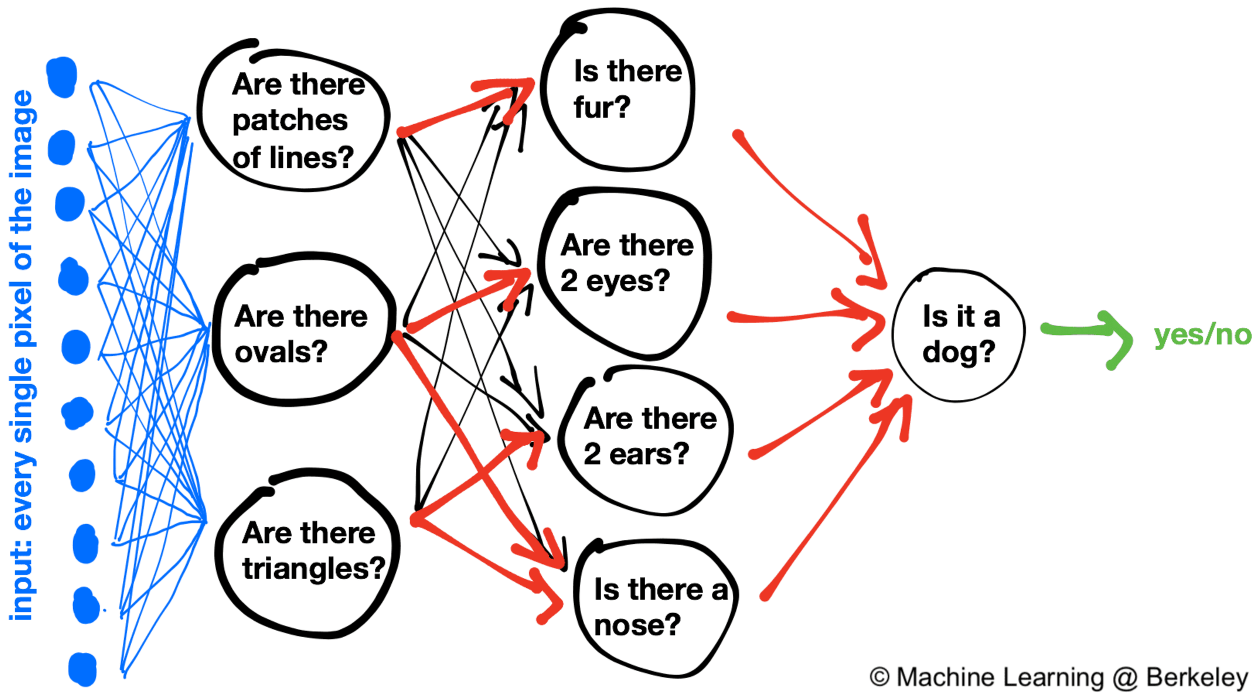
- Simplified (from MIT 6.S191introtodeeplearning.com)

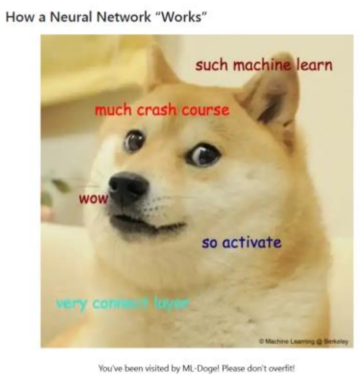


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

How much can a deep network do?

- Identify a dog in a photo (Machine Learning Crash Course: Part 3 - ML@B Blog)

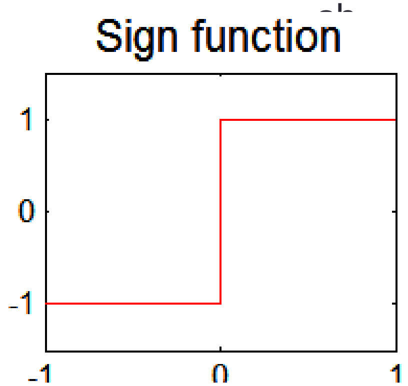




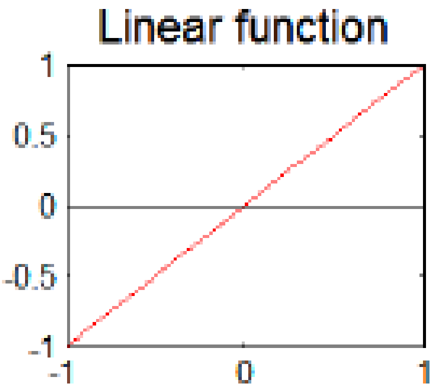
- Pixels line segments distinct features judgement
- convolution layer regular layer

Activation functions

- Objective: to fire a neuron



vs.



bio-neuron

possible-neuron?

- Issues:

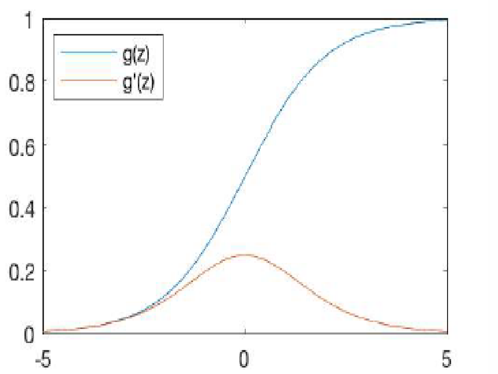
  - sharp vs. dull

  - first order information (gradient information)

Activation functions

- Commonly used activation functions

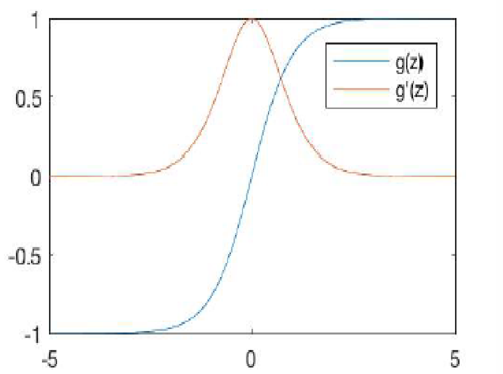
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

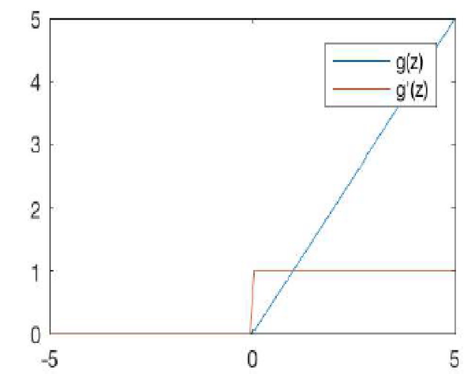
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

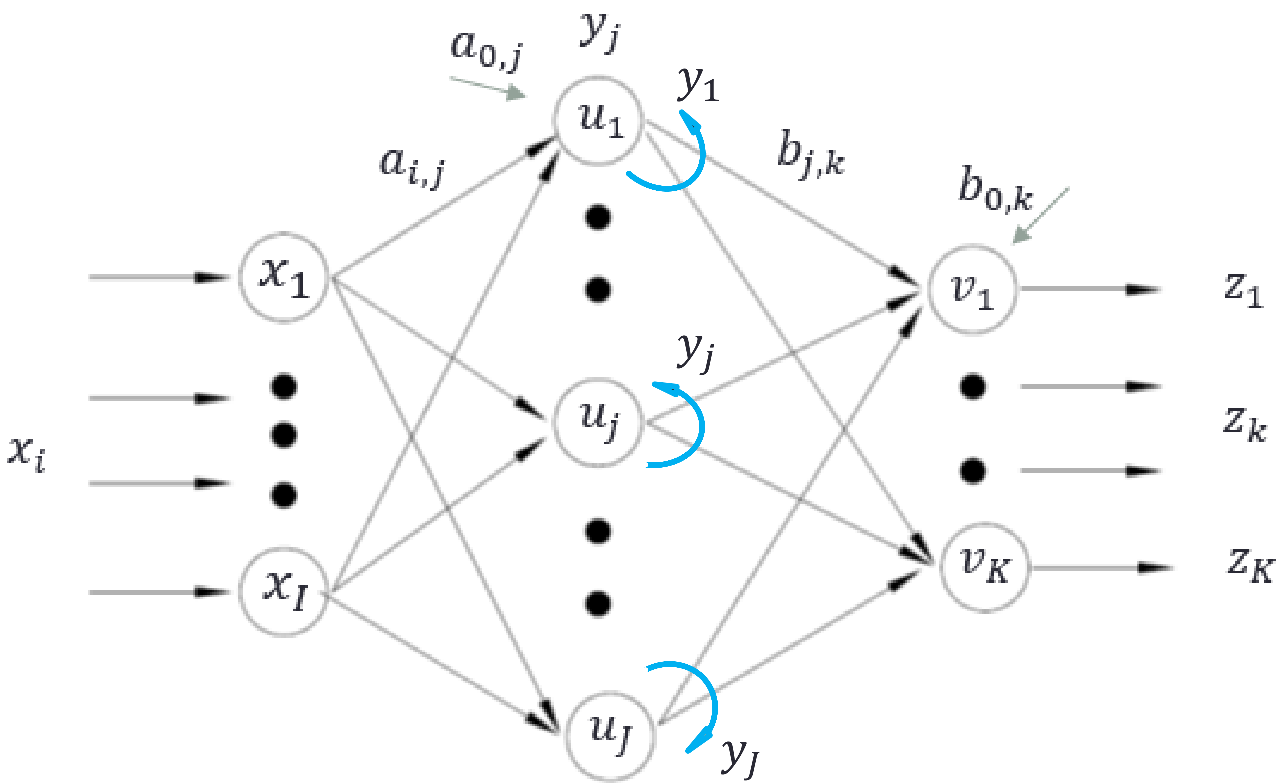
- Pros and cons?

Fundamentals of multi-layer perceptron NN

- Feedforward with backpropagation
  - for each neuron/node, activation function is fixed, connection weights may change (learning)
  - input information feeds forward for computing outputs (in testing and in use)
  - error/loss information propagates backward for adjusting connection weights (in training)
- References:
  - David Rumelhart, Geoffrey Hinton, Ronald Williams(1986)
  - David Parker (1982,1985) / Yann Le Cun (1986)
  - First Discovery of back propagation goes to Paul Werbos (1974 Harvard PhD thesis "Beyond Regression")

Feed forward computations

- Example: 3-layer input-hidden-output shallow network



$$u_j = a_{0j} + \sum_{i=1}^{I} a_{ij} x_i, \quad v_k = b_{0k} + \sum_{j=1}^{J} b_{jk} y_j,$$

$$y_j = g(u_j), \; j = 1, \ldots, J, \quad z_k = g(v_k), \; k = 1, \ldots, K$$

Feed forward computations

- Example: 3-layer input-hidden-output shallow network

$$u_j = a_{0j} + \sum_{i=1}^{I} a_{ij} x_i, \quad v_k = b_{0k} + \sum_{j=1}^{J} b_{jk} y_j,$$

$$y_j = g(u_j), \; j = 1, \ldots, J, \quad z_k = g(v_k), \; k = 1, \ldots, K$$

$$\boldsymbol{z} = \boldsymbol{\Phi}(B^T \boldsymbol{\Phi}(A^T \boldsymbol{x} + \boldsymbol{a}_0) + \boldsymbol{b}_0)$$

ReLU leads to a piecewise linear approximator

- Hanin, Boris; Sellke, Mark (March 2019). "Approximating Continuous Functions by ReLU Nets of Minimal Width". *Mathematics*. MDPI. **7** (10): 992. arXiv:1710.11278. doi:10.3390/math7100992.

$$\boldsymbol{z} = \boldsymbol{\Phi}(B^T \boldsymbol{\Phi}(A^T \boldsymbol{x} + \boldsymbol{a}_0) + \boldsymbol{b}_0)$$

$\phi(v) = ReLU(v) = \max\{0, v\}$ is a piecewise linear function

$\Rightarrow$ $\boldsymbol{z}$ is piecewise linear in $\boldsymbol{x}$

$\Rightarrow$ NN using ReLU activation provides a piecewise linear approximation of the underlying input-output relation.

Good for large scale operations of deep networks!

Backpropagation learning

- Example: 3-layer input-hidden-output shallow network

  Mean squared error (*2-norm*) model

  - Objective: to find the weights/coefficients $\{a_{ij}, b_{jk}\}$ that provides the best fit between the neural network output ($\boldsymbol{z}$) and the target function value ($\boldsymbol{t}$).

Backpropagation learning

- Example: 3-layer input-hidden-output shallow network

- Model: Minimizing the mean squared error

$$E = \frac{\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(z_{kn} - t_{kn})^2}{NK}$$

N: number of examples in the data set

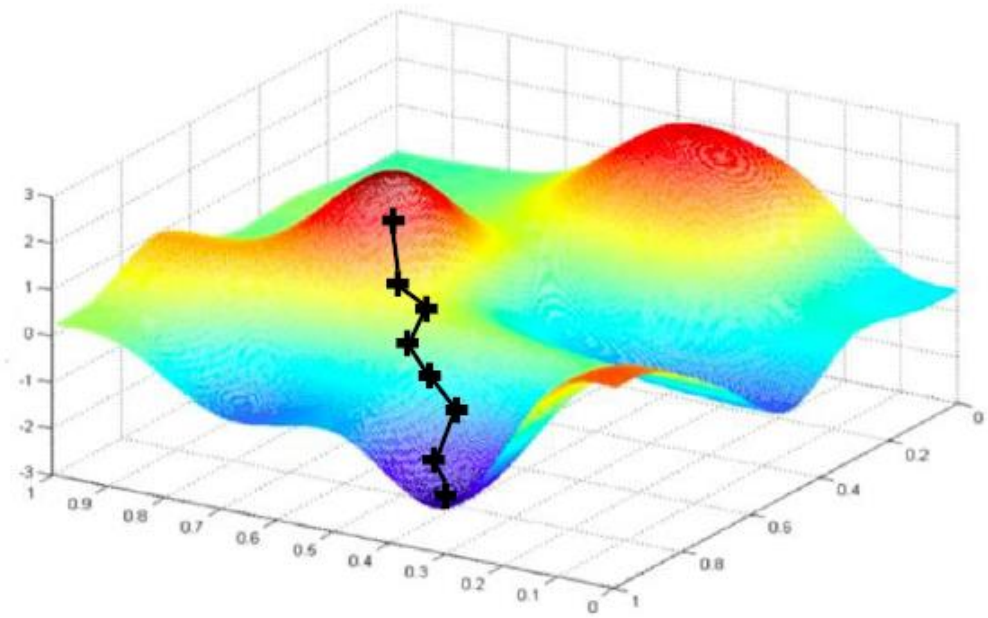
K: number of outputs of the network

t_{kn}: kth target output for the nth example

z_{kn}: kth output for the nth example

Delta learning rule – gradient decent method

- Objective: $\min E(a_{ij}, b_{jk})$ - quite complex
- Principle: adjust current weights along the negative gradient direction of the error/loss function with a proper step-length to reduce the error step by step.



Gradient decent direction in approximation

- Taylor expansion theorem

  - $f \in C^1$

  $$f(x^2) = f(x^1) + \nabla f(\bar{x})(x^2 - x^1)$$

  - $f \in C^2$

  $$f(x^2) = f(x^1) + \nabla f(x^1)(x^2 - x^1)$$

  $$+\frac{1}{2}(x^2 - x^1)^T F(\bar{x})(x^2 - x^1)$$

Approximation

When $x \approx x^1$

$$f(x) \approx f(x^1) + \sum_{k=1}^{m-1} \frac{1}{k!} d^k f(x^1; x - x^1)$$

Take $m = 2$

$$f(x) \approx f(x^1) + \nabla f(x^1)(x - x^1)$$

Assume $\nabla f(x^1) \neq 0$.

- Take $x - x^1 = \nabla f(x^1)$, i.e., moving from x^1 in the gradient direction at x^1

$$f(x) \approx f(x^1) + \|\nabla f(x^1)\|^2 > f(x^1)$$

Approximation

- For $x - x^1 = -[\nabla f(x^1)]$, i.e., moving from x^1 in the negative gradient direction

$$f(x) \approx f(x^1) - \|\nabla f(x^1)\|^2 < f(x^1)$$

- For any $d \triangleq x - x^1$

$$\nabla f(x^1)(x - x^1) = \|d\| \underbrace{\|\nabla f(x^1)\| \cos\theta}_{\text{projection of } \nabla f(x^1) \text{ onto } d}$$

Gradient decent method

- Facts: For a differentiable function $f(\boldsymbol{x})$: $\mathbb{R}^n \rightarrow \mathbb{R}$

1. Moving along the gradient direction $\nabla f(\boldsymbol{x})$ will increase the objective value.
2. Moving along the negative gradient direction$-\nabla f(\boldsymbol{x})$ will decrease the objective value.
3. The gradient direction $\nabla f(\boldsymbol{x})$ is the steepest ascent direction for moving.
4. The negative gradient direction $-\nabla f(\boldsymbol{x})$ is the steepest decent direction for moving.
5. Gradient decent method

   $$\boldsymbol{x}_{new} = \boldsymbol{x}_{current} - \theta \nabla f(\boldsymbol{x}_{current})$$

   with a step-length $\theta > 0$.

Calculate gradient direction using chain rule

- Chain rule for the composition of two differentiable functions f and g:

$$h(x) = f\big(g(x)\big) \Rightarrow h'(x) = f'\big(g(x)\big)g'(x)$$

- Expressed in Leibniz's notation

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

- General form

$$\frac{df_1}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3} \cdots \frac{df_n}{dx}$$

NN learning mechanisms

- Example: 3-layer input-hidden-output shallow network

- Online (example by example) learning

$$N = 1$$

$$\bar{E} \triangleq KE = \frac{1}{2}\sum_{k=1}^{K}(z_k - t_k)^2$$

- (Whole) batch learning

$$E' \triangleq NKE = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(z_{kn} - t_{kn})^2$$

- Stochastic (batch) learning
  - randomly choose a small batch of examples

Online learning

- Example: 3-layer input-hidden-output shallow network

$$u_j = a_{0j} + \sum_{i=1}^{I} a_{ij} x_i, \quad v_k = b_{0k} + \sum_{j=1}^{J} b_{jk} y_j,$$

$$y_j = g(u_j), \ j = 1, \ldots, J, \quad z_k = g(v_k), \ k = 1, \ldots, K$$

- Online (example by example) learning

$$N = 1$$

$$\bar{E} \triangleq KE = \frac{1}{2} \sum_{k=1}^{K} (z_k - t_k)^2$$

- Gradient information (chain rule)

$$\frac{\partial \bar{E}}{\partial b_{jk}} = \frac{\partial \bar{E}}{\partial z_k} \frac{\partial z_k}{\partial v_k} \frac{\partial v_k}{\partial b_{jk}} = \begin{cases} p_k, \ j = 0 \\ p_k y_j, \ j = 1, \ldots, J \end{cases}$$

where

$$p_k = (z_k - t_k) z_k (1 - z_k)$$

$$y_j = g(a_{0j} + \sum_{i=1}^{I} a_{ij} x_i)$$

$$\frac{\partial \bar{E}}{\partial a_{ij}} = \left(\sum_{k=1}^{K} \frac{\partial \bar{E}}{\partial z_k} \frac{\partial z_k}{\partial v_k} \frac{\partial v_k}{\partial y_j} \right) \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial a_{ij}} = \begin{cases} q_j, \ i = 0 \\ q_j x_i, \ i = 1, \ldots, I \end{cases}$$

where

$$q_j = \left[\sum_{k=1}^{K} p_k b_{jk} \right] y_j (1 - y_j)$$

Still remember the chain rule?

- Hint:

$$\bar{E} = \frac{1}{2}\sum_{k=1}^{K}(z_k - t_k)^2 \qquad \frac{\partial \bar{E}}{\partial z_k} = (z_k - t_k)$$

$$z_k = g(v_k) \triangleq \frac{1}{1+e^{-v_k}} \qquad \frac{\partial z_k}{\partial v_k} = z_k(1-z_k)$$

$$v_k = b_{0k} + \sum_{j=1}^{J} b_{jk} y_j \qquad \frac{\partial v_k}{\partial b_{jk}} = \begin{cases} 1, & j = 0 \\ y_j, & j = 1, \ldots, J \end{cases}$$

$$\frac{\partial v_k}{\partial y_j} = b_{jk}$$

$$y_j = g(u_j) \triangleq \frac{1}{1+e^{-u_j}} \qquad \frac{\partial y_j}{\partial u_j} = y_j(1-y_j)$$

$$u_j = a_{0j} + \sum_{i=1}^{I} a_{ij} x_i \qquad \frac{\partial u_j}{\partial a_{ij}} = \begin{cases} 1, & i = 0 \\ x_i, & i = 1, \ldots, I \end{cases}$$

$$p_k \triangleq \frac{\partial \bar{E}}{\partial z_k}\frac{\partial z_k}{\partial v_k} = (z_k - t_k) z_k (1 - z_k)$$

$$q_j = \left[\sum_{k=1}^{K} \frac{\partial \bar{E}}{\partial z_k}\frac{\partial z_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}\right]\frac{\partial y_j}{\partial u_j} = \left[\sum_{k=1}^{K} p_k b_{jk}\right] y_j(1-y_j)$$

Delta learning rule – gradient decent method

- Delta rule: – Iteratively updating the weights (a_{ij}, b_{jk})

$$w^{m+1} = w^m - \lambda d^m$$

where

$$d^m = \sum_{n=1}^{N} \left(\left. \frac{\partial E}{\partial w} \right|_m \right)_n$$

$$\begin{cases} \text{Online,} & N = 1 \\ \text{Batch,} & N \\ \text{Stochastic,} & < N. \end{cases}$$

$$\lambda = \text{step} \quad \text{length}$$

- Related Questions:

1. Will it converge to a local minimum?
2. How efficient?
3. How to choose the step-length?

Delta learning rule

Enhancement with memory:

- Momentum

$$w^{m+1} = w^m - \lambda[\mu d^m + (1-\mu)d^{m-1}]$$

- Adaptive learning rate / Second order information learning.

Complexity of training

- Potential problems:

This is an excerpt from *Post Capture Pocket Guide*.

Sensor Resolution (megapixels)	Typical Image Resolution (pixels)	Maximum Print Size	Print Resolution	Maximum Output Size
2.16	1800 x 1200	6 x 4 inch	300 dpi	Snapshot prints
3.9	2272 x 1704	7.6 x 5.7 inch	300 dpi	'Jumbo' snapshot prints
5.0	2592 x 1944	8.6 x 6.5 inch	300 dpi	8 x 6 inch enlargements
7.1	3072 x 2304	10.2 x 7.7 inch	300 dpi	A4 sized prints
8.0	3264 x 2448	13.6 x 10.2 inch	240 dpi	A4 sized prints
10.0	3648 x 2736	18.2 x 13.7 inch	200 dpi	A3 sized prints
12.1	4000 x 3000	20 x 15 inch	200 dpi	A3+ sized prints
14.7	4416 x 3312	22.1 x 16.6 inch	200 dpi	A2 sized prints
21.0	5616 x 3744	31.2 x 20.8 inch	180 dpi	A1 sized prints

Stochastic gradient decent (SGD)

- Basic idea:
  - Loss is the sum of N differentiable functions.

$$Loss(\boldsymbol{x}) = \sum_{j=1}^{N} f_j(\boldsymbol{x})$$

  - Intend to minimize the loss

$$\min \sum_{j=1}^{N} f_j(\boldsymbol{x})$$

  - Gradient direction of $Loss(\boldsymbol{x})$ at a point $\boldsymbol{x}^i$ is

$$\nabla\, Loss(\boldsymbol{x}) = \sum_{j=1}^{N} \nabla f_j(\, \boldsymbol{x}^i)$$

  - The new iterate is

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i - \theta_i \sum_{j=1}^{N} \nabla f_j(\, \boldsymbol{x}^i)$$

where $\theta_i > 0$ is a step-length at i^{th} iteration.

Stochastic gradient decent (SGD)

- Basic idea:

  - Instead of calculating N gradients, randomly pick some $\hat{\imath} \in \{1,2,\ldots,N\}$ and use $\nabla f_{\hat{\imath}}(\boldsymbol{x}^i)$ for $\sum_{j=1}^{N} \nabla f_j(\boldsymbol{x}^i)$ such that

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i - \theta_i \nabla f_{\hat{\imath}}(\boldsymbol{x}^i)$$

where $\theta_i > 0$ is a step-length at i^{th} iteration.

SGD vs. GD

- Basic idea: (image from Analytics Vidhya)

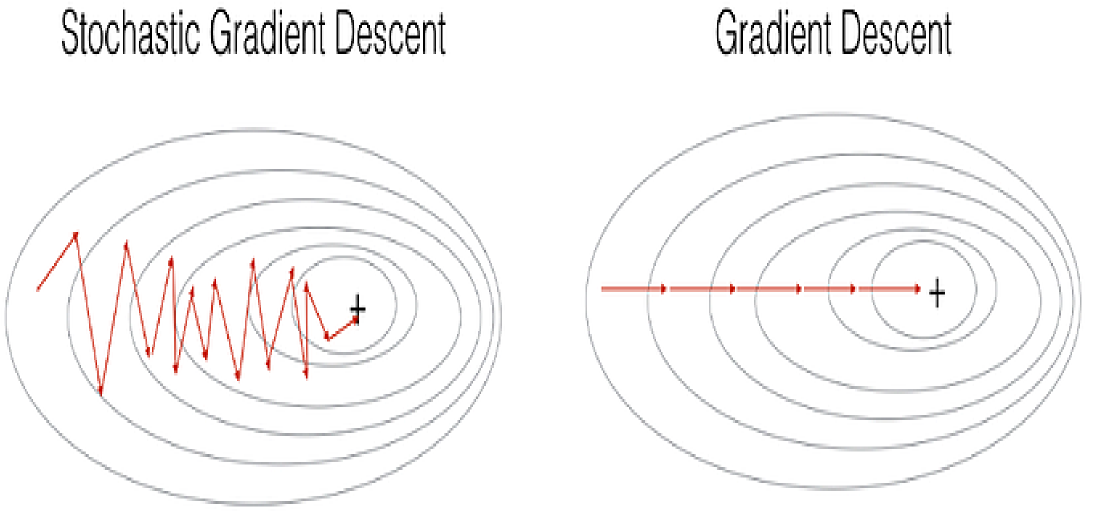
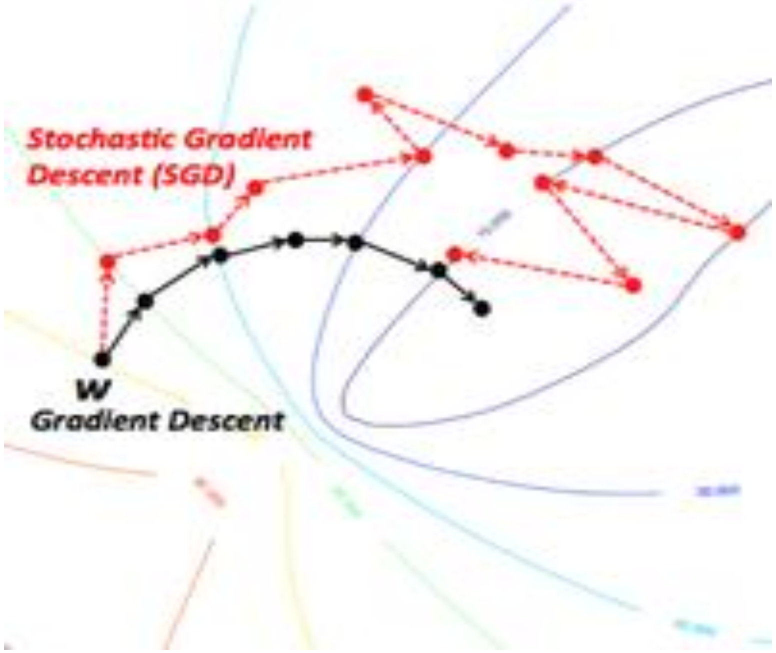


Figure 1 : SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

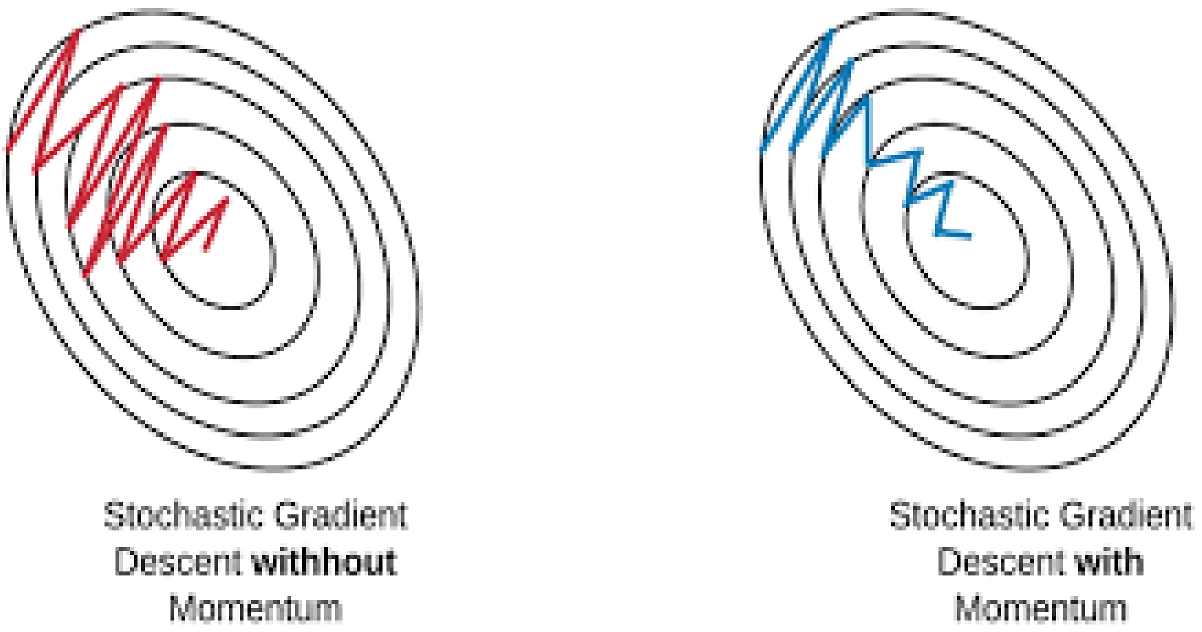
SGD vs. GD

- Basic idea: (image from golden.com)
  - SGD could be nasty



Stochastic gradient direction - SGD

- Reduce variations: (image from wikidocs.net)



Stochastic gradient direction - SGD

- Issues:

1. Will SGD converge to a local minimum?
   - SGD may serve as an unbiased estimator such that

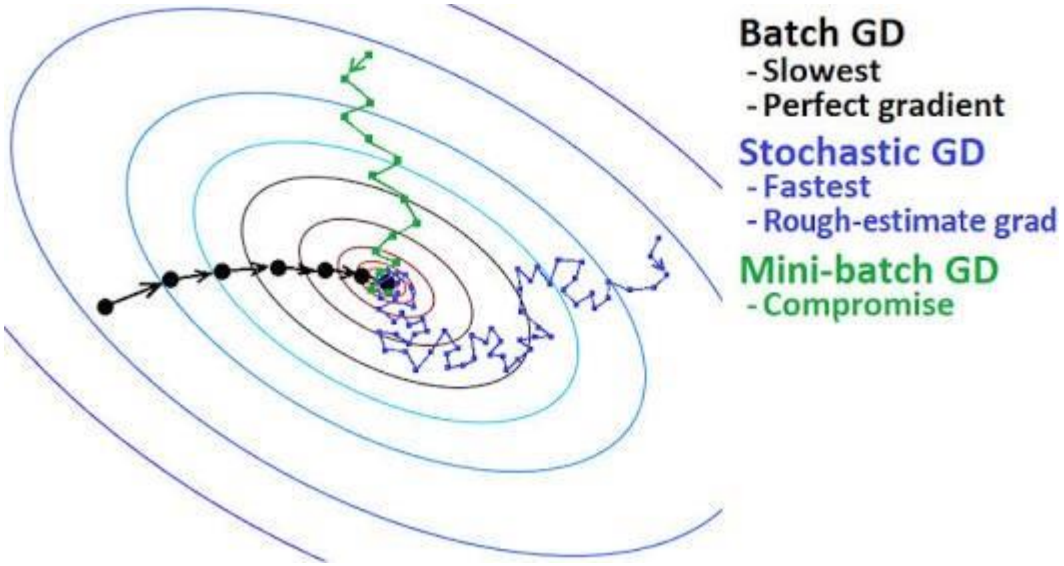
$$\mathbb{E}\big(sgd(x)\big) = \nabla Loss(\boldsymbol{x})$$

2. How to decide step-length at each iteration?
   - large at beginning, small at the end ?
   - overfitting
3. Randomly select one each time or stay on the same?
   - does it really matter?
4. Will it be better to select more than one each time?

Good for large scale operations of deep networks!

Batch gradient decent

- (image from https://sweta-nit.medium.com/)



Initialization and stopping of training

- Initial weights

  - Set the hidden node weights to small random numbers distributed evenly around 0.

  - Initialize half of each output node's weights with values of 1 and the other half with -1; if there is an odd number of nodes, initialize bias weights at 0.

- Stopping rule

  - Stop learning after a finite number of iterations (epochs) or E becomes small enough, or not much more improvement can be made.

Implementation examples

- Gradient decent (MIT 6.S191)

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

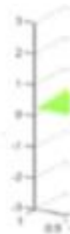
Can be very computationally expensive

Implementation examples

- Stochastic gradient descent (MIT 6.S191)

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



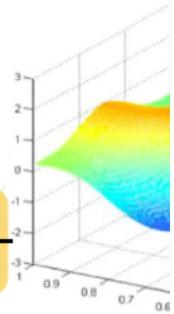
Easy to compute but:
very noisy
(stochastic)!

Implementation examples

- Stochastic gradient descent (MIT 6.S191)

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

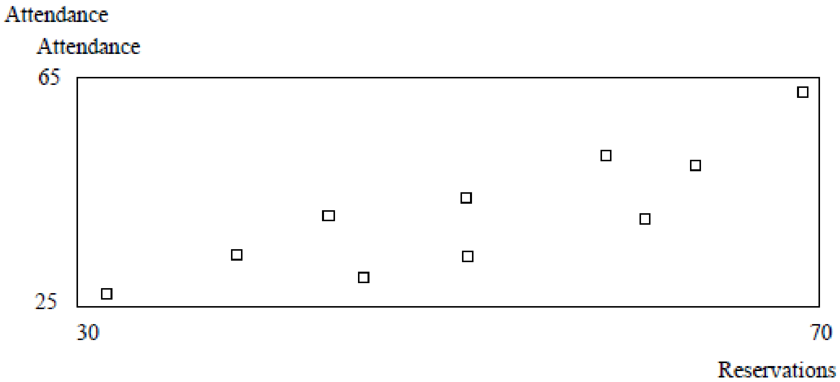
Learning for generalization

- Questions:
  1. Learning provides the best fit for the training examples through optimization. But, will the good/expected performance be generalized (or, holds valid) for new examples in use?

  2. Noise in the training data may cause the overfitting problem that prevents generalization. How to avoid overfitting?

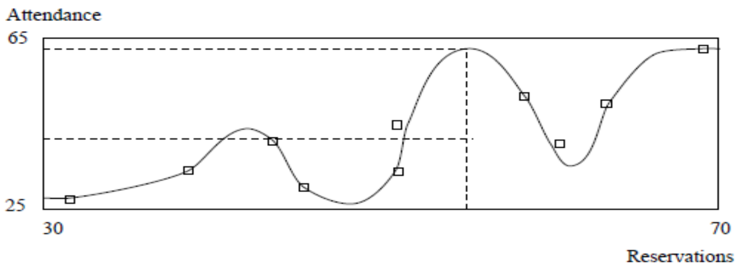
Generalization

- Example: restaurant's historical data for new year eve dinner

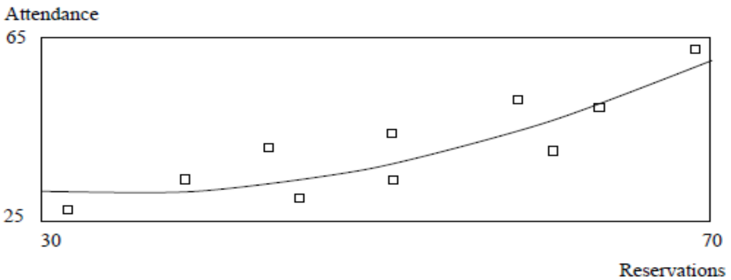


10 Years Data

- NN Performance Better generalization ?



Network Outputs



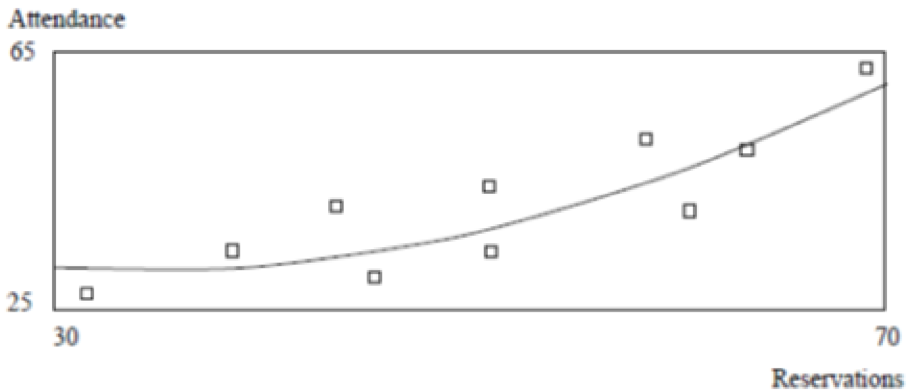
More Real Model

Overfitting prevention

- Commonly adopted rules:
    1. reduce noise in the data
    2. increase the sample size
    3. do not over-train the network
    4. limit the number of hidden nodes
    5. conduct cross validation

Noise and sample size

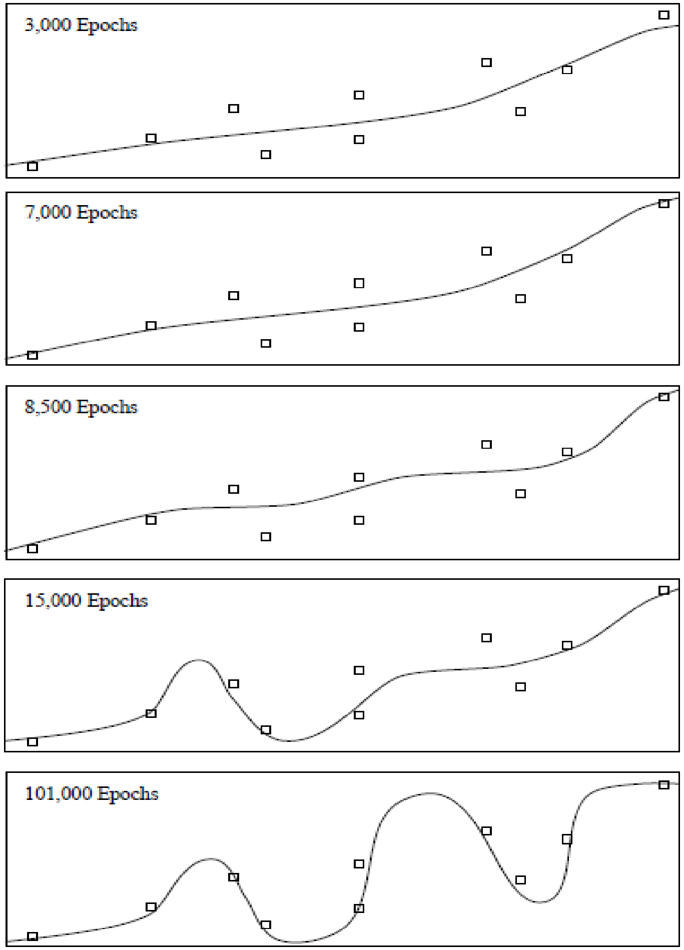
- Statistical pre-treatment



More Real Model

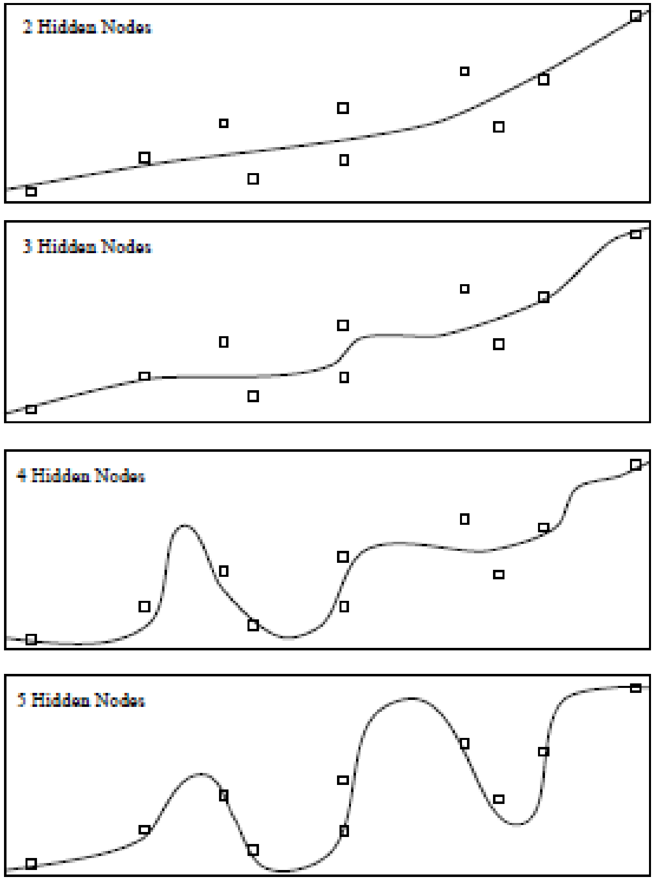
Over training

- The course of training for an NN with 5 hidden nodes

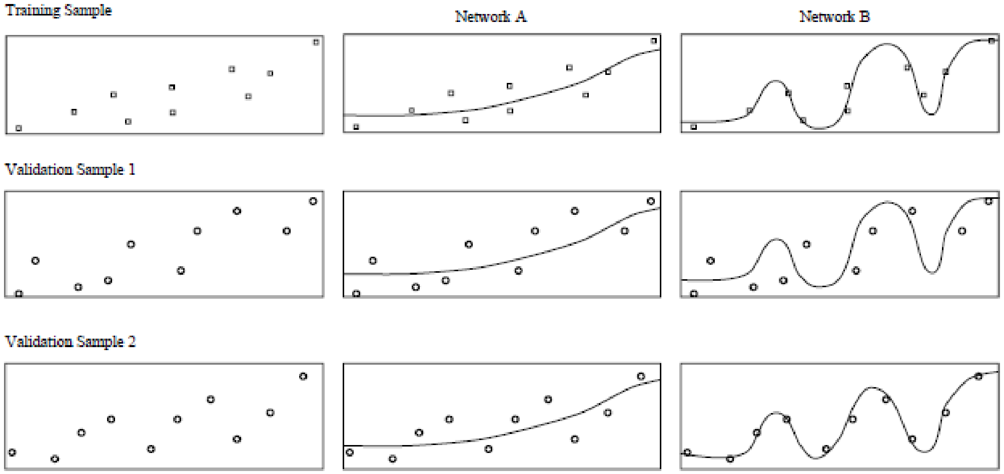


Nodes in the hidden layer

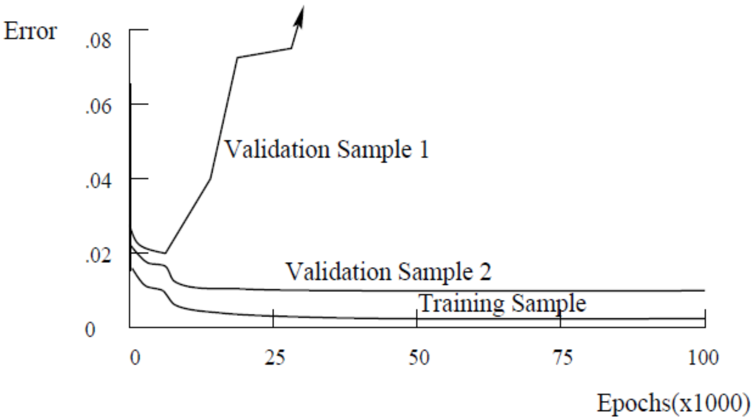
- Limit the number of hidden nodes
  - – reduce the unnecessary complexity



Cross validation for the right network



Output of two networks compared to training and validation samples



Error on Two Validation Samples

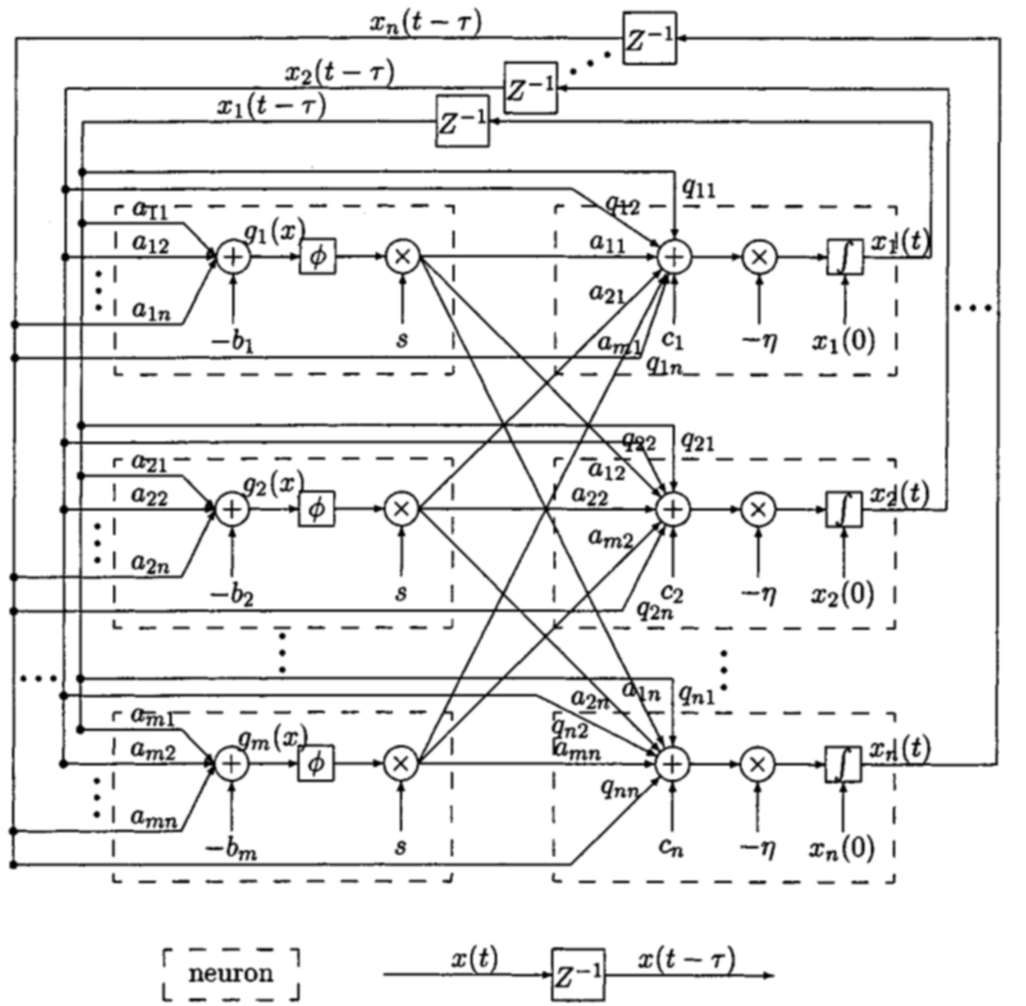
*validation uses the weights obtained by training.

More about ANN

- Multi-layer perceptron (MLP) network is most popular in use.
- MLP can be shown mathematically as a universal approximator under some assumptions.
- MLP networks are not the only feedforward neural networks.
- Recurrent networks and radial basis function (RBF) networks are also feedforward neural networks.
- Feedbackward neural networks exist for non-supervised learning and mathematical optimizer with hardware implementation of analogue circuits.

Example - Feedback neural net solver for QP problems

- IEEE TNN, Vol 11, No. 1, 2000, 230-240 (Y-H Chen & S-C Fang) Neurocomputing with Time Delay Analysis for Solving Convex Quadratic Programming Problems



Learning with sequential data

- Examples:

  - Auto texting

    "Hei Google What ti….

    What tim….

    What time …."

  - Music nodes

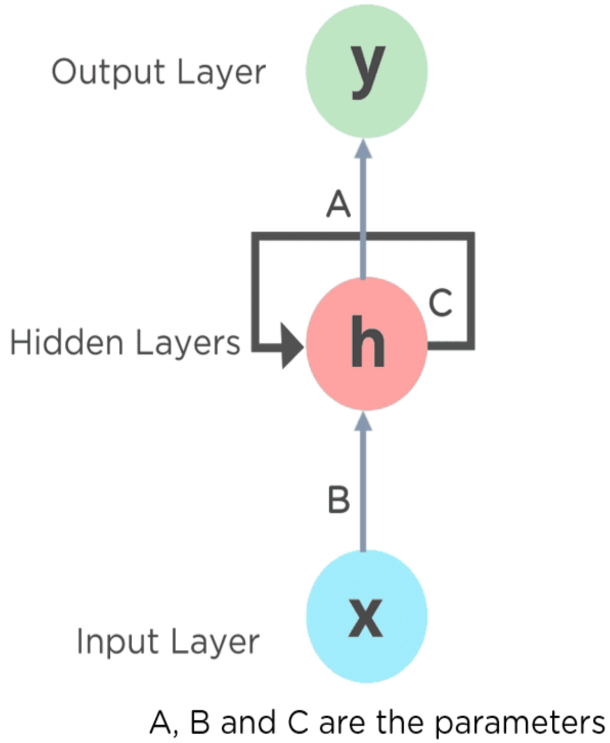
    "Doe Ray Me Far …

    Doe Ray Me Far Sew …

    Doe Ray Me Far Sew La …"

Recurrent neural network (RNN)

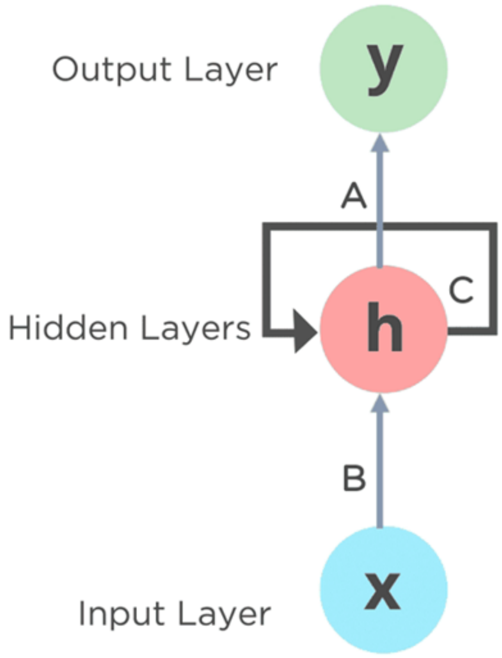
- RNN is a feedforward neural network that stores short memory of previous results and brings to current status to process sequential data.



<image from simplilearn.com>

Example

- 高考复习进度
- 顺序 1：每周七天，导师出席与否，依序每日复习一科目
  中文》数学》英文》物理》化学》生物》时事分析》中文
- 顺序 2: 导师请假日，自行复习昨日复习科目



A, B and C are the parameters

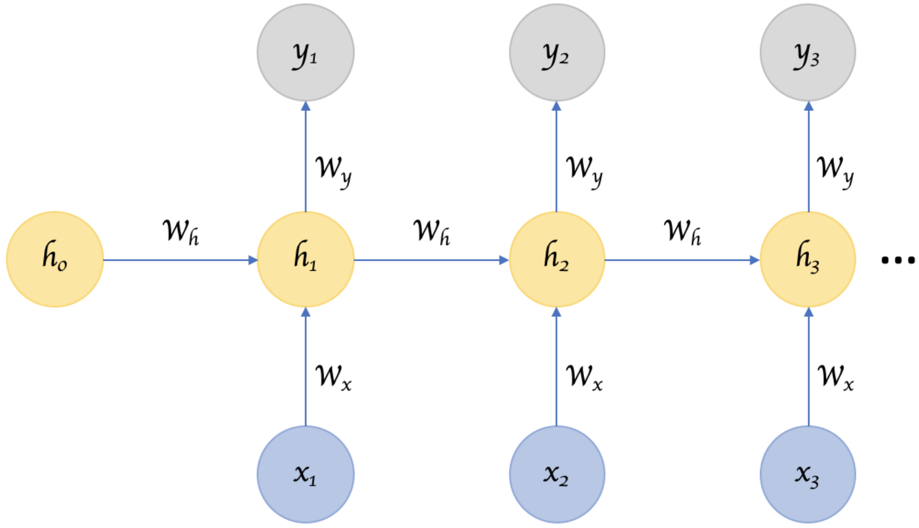
y =复习科目

C =昨日科目 (recurrent)

x = （星期，导师）

Recurrent neural network (RNN)

- MLP networks are commonly used for classifiers and regression.
- Recurrent networks are particularly good for temporal forecasting.



Example: Elman network

$$h_t = \sigma_h\left(W_x x_t + W_h h_{\{t-1\}} + b_h\right); \quad y_t = \sigma_y(W_y h_t + b_y)$$

Radial basis function networks

- RBF network is a feedforward neural network using a radial basis functions as its activation function.
- Radial basis function has the form of

$$g(x, \theta, b) = \phi(\frac{x-\theta}{b})$$

where $\phi : R^n \to R$, $x \in R^n$, $\theta \in R^n$ in a "center vector", and $b \in R$ is a "spread parameter".

- Gaussian function is a typical example:

$$g(x) = \exp(-\frac{||x-\theta||}{b})$$

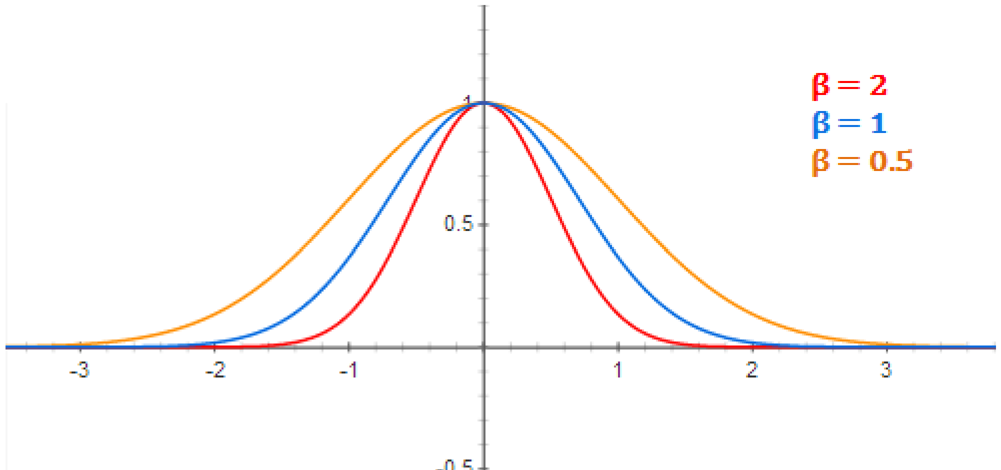
Gaussian-type function

- Gaussian function is a typical example:

$$g(x) = \exp(-\frac{\|x-\theta\|}{b}) \quad \text{or} \quad g(x) = \exp(-\beta\|x-\theta\|^2)$$

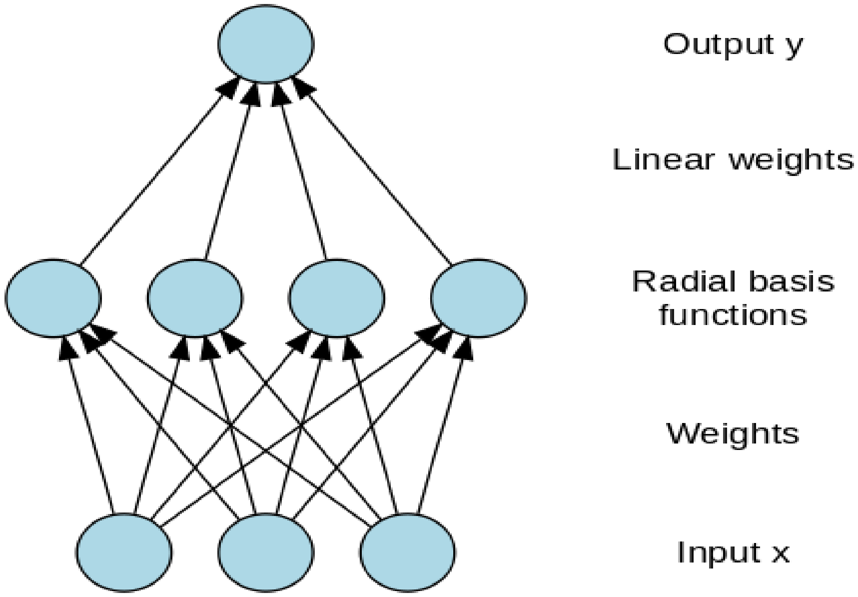
- Observations:

1. For any b (or β) > 0, $0 < g(\boldsymbol{x}) \leq 1$.
2. For the same b(or β) > 0, $g(\mathrm{x})$ is closer to 1 as x is closer to $\boldsymbol{\theta}$.
3. For the same $\boldsymbol{\theta}$, $g(\boldsymbol{x})$ is closer to 1 as b goes larger $(or\ \beta\ goes\ smaller)$.



Radial basis function network

- Simple architecture RBFN (from Wikipedia)



$y \in \mathbb{R}$

$c_i \in \mathbb{R},\ i = 1, \ldots, N$

N RBF neurons

1

$\boldsymbol{x} \in \mathbb{R}^n$

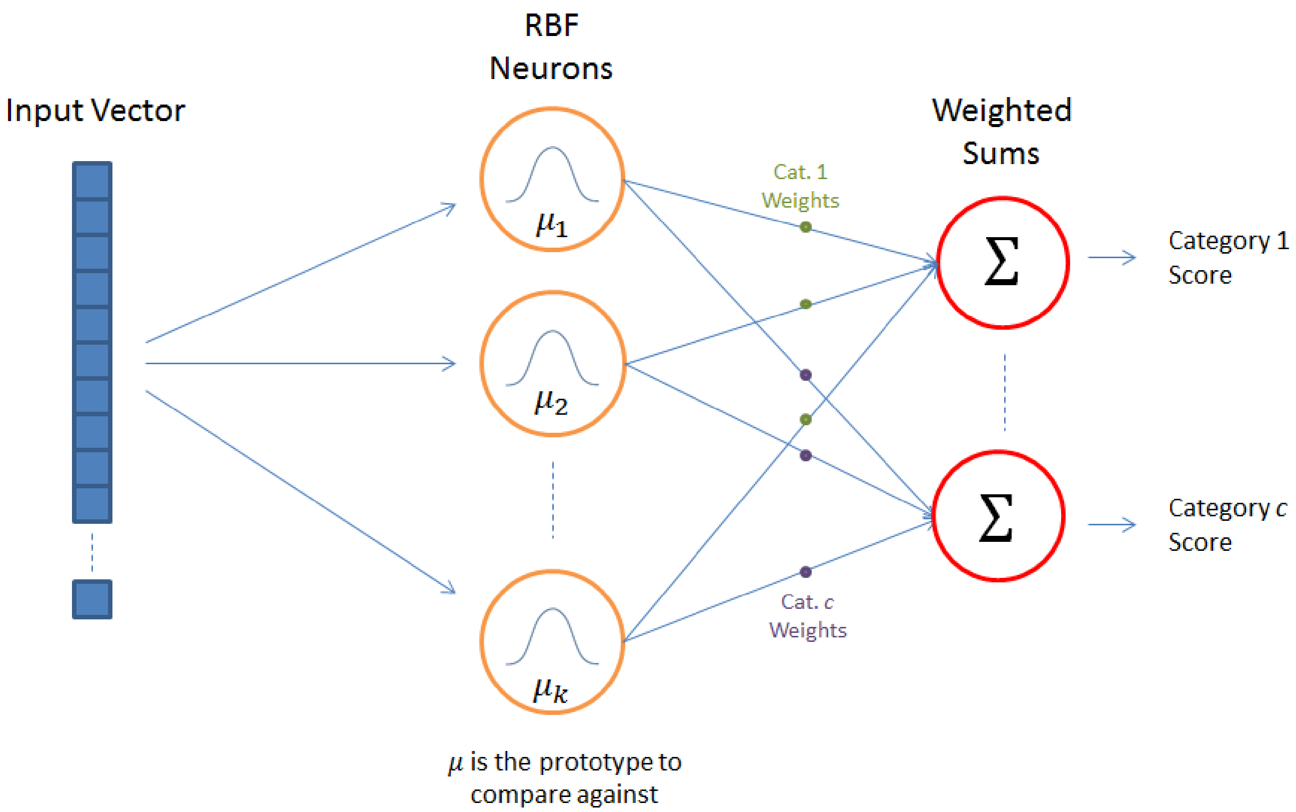
Output

$$y = f(\boldsymbol{x}) = \sum_{i=1}^{N} c_i g_i(\boldsymbol{x}) = \sum_{i=1}^{N} \mathrm{c_i} \exp(-\beta \|\boldsymbol{x} - \boldsymbol{\theta}_i\|^2)$$

where c_i and $\boldsymbol{\theta}_i$ may be separately learned by optimizing the fit.

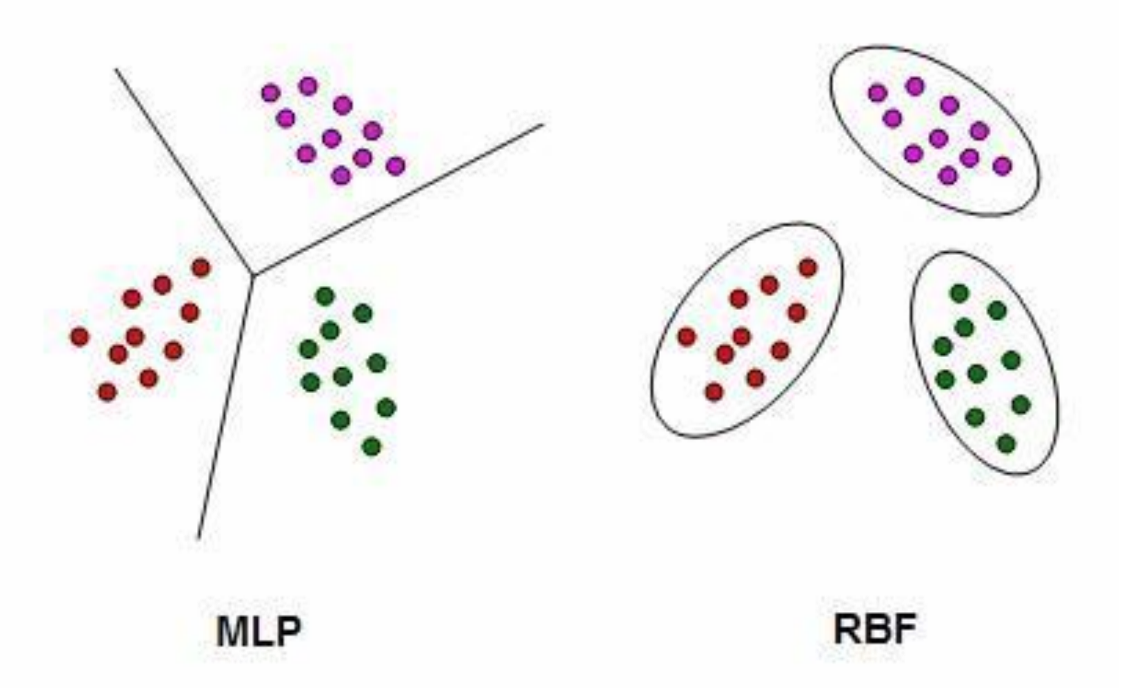
RBFN multi-outputs

- Simple RBF network architecture
  <https://mccormickml.com/2013/08/15/radial-basis-function-network-rbfn-tutorial/>



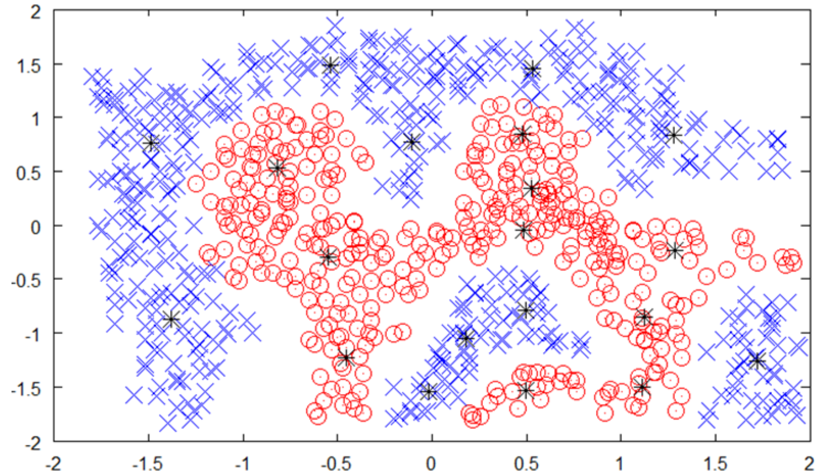
Different philosophy

- Radial basis function networks vs. MLP
  - Image from https://towardsdatascience.com/radial-basis-functions-neural-networks-all-we-need-to-know-9a88cc053448

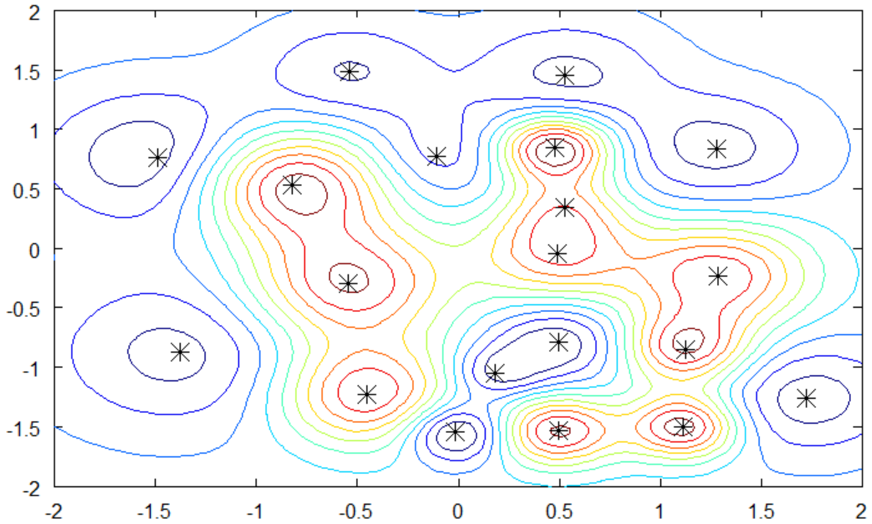


Intuition behind RBFN

- Clustering in categories

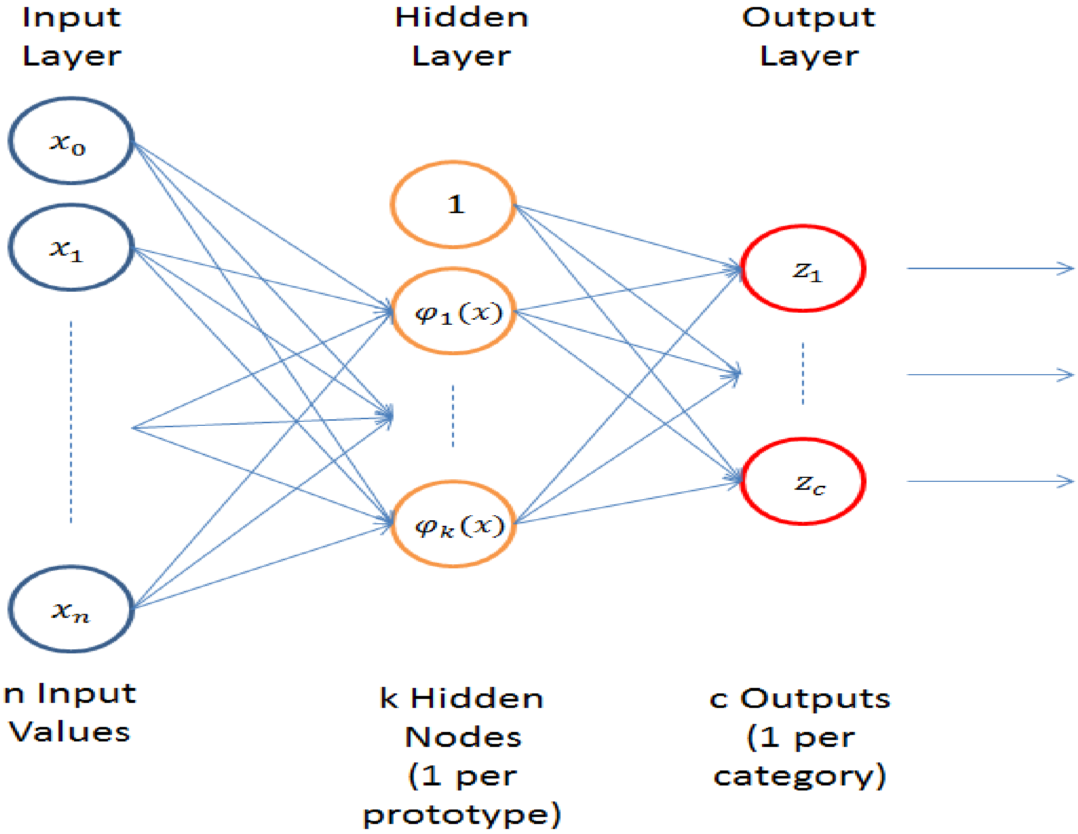


- Membership/possibility



RBFN as a neural network

- Image from <https://mccormickml.com/2013/08/15/radial-basis-function-network-rbfn-tutorial/>



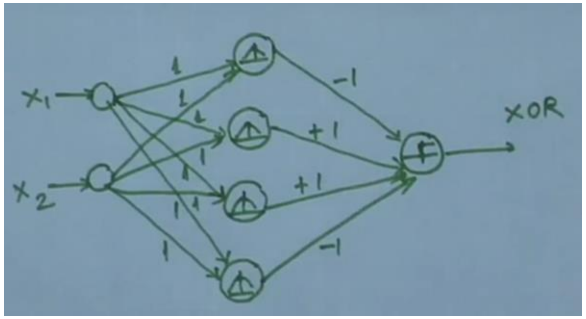
Example – XOR operation

- Truth table
-

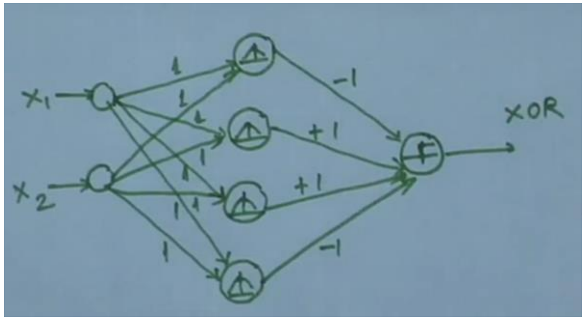
$x\ (XOR)\ y$	$y = 0$	$y = 1$
$x = 0$	0	1
$x = 1$	1	0

Architecture of XOR RBFN:

- 2 input nodes
- 4 RBF neurons
- 1 output for XOR
- sign function for output



Example – XOR operation



- Architecture of XOR RBFN:
  - 2 input nodes: (x_1, x_2) for (x, y)
  - all weights equal to 1
  - 4 RBF neurons:

$$\boldsymbol{\theta}_1 = (0,0)^T, \boldsymbol{\theta}_2 = (0,1)^T, \boldsymbol{\theta}_3 = (1,0)^T, \boldsymbol{\theta}_4 = (1,1)^T$$

$$\beta = \frac{1}{2}, \ g_i(\boldsymbol{x}) = \exp(-\frac{1}{2}\|\boldsymbol{x} - \boldsymbol{\theta}_i\|^2)$$

  - 1 output for XOR
  - connection weights $c_1 = -1, c_2 = 1, c_3 = 1, c_4 = -1$
  - sign function for output

Example – XOR operation

- RBFN output

$Input$	g_1	g_2	g_3	g_4	$\sum c_i g_i$	$output$
$(0,0)$	1.0	0.6	0.6	0.4	-0.2	0
$(0,1)$	0.6	1.0	0.4	0.6	0.2	1
$(1,0)$	0.6	0.4	1.0	0.6	0.2	1
$(1,1)$	0.4	0.6	0.6	1.0	-0.2	0

MLP vs. RBFN

- Comments from <researchgate.net>

Feature of network architecture	Neural network type	
	MLP	**RBF**
Signal transmission	Feed-forward	Feed-forward
Process of building the model	One stage	Two different independent stages: • First stage: the probability distribution is established by means of radial basis functions • Second stage: the network learns the relations between input x and output y Note: The lag is only visible in RBF in the output layer
Threshold	Yes	No
Type of parameters	Weights and thresholds	• Location and width of basis function • Weights binding basis functions with output
Functioning time	Faster	Slower (bigger memory and size required)
Learning time	Slower	Faster

Source: own, on the basis of Bishop (1995); Haykin (2011); Migdał Najman and Najman (2013); Skubalska-Rafajłowicz (2011); West (2000).

Approximation power of neural networks

- MLP networks
  - Cybenko showed that a backpropagation MLP, with one hidden layer and any fixed continuous sigmoidal nonlinear function, can approximate any continuous function arbitrarily well on a compact set.

    *G. Cybenko. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems, 2:303-314, 1989.

  - When used as a binary-valued neural network with the hard-limiter (step) activation function, a backpropagation MLP with two hidden layers can form arbitrary complex decision regions to separate different classes.

    *R. P. Lippmann. An introduction to computing with neural networks. IEEE Acoustics, Speech, and Signal Processing Magazine, 4(2):4-22, 1987.

Approximation power of neural networks

- MLP networks (universal approximation)
  - Leshno et al. showed that “a standard multilayer feedforward network with a locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy if and only if the network's activation function is not a polynomial.”

*Leshno, M., Lin, V., Pinkus, A., Shochen, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks, 6, 861-867.

Approximation power of neural networks

- RBF networks
  - The most well-known result is due to Park and Sandberg, who showed that if the RBF function used in the hidden layer is continuous almost everywhere, bounded and integrable on $\mathbb{R}^n$, and the integration is not zero, then a three-layered neural network can approximate any function in $L^p(\mathbb{R}^n)$ with respect to the L^p norm with $1 \leq p < +\infty$.

*Park, J., Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. Neural Computation, 3(2), 246-257.

*Park, J., Sandberg, I. W. (1993). Approximation and radial-basis-function networks. Neural Computation, 5, 305-316.

Approximation power of neural networks

- RBF networks (universal approximation)
  - One of the most general results is due to Liao, Fang and Nuttle, who showed that, if the radial-basis activation function used in the hidden layer is continuous almost everywhere, locally essentially bounded, and not a polynomial, then the three-layered radial-basis function network can approximate any continuous function with respect to the uniform norm. Moreover, Radial Basis Function Networks (RBFN) can approximate any function in $L^p(\mu)$, where $1 \leq p < +\infty$ and μ is any finite measure, if the radial-basis activation function used in the hidden layer is essentially bounded and not a polynomial.

*Liao, Y., Fang, S. C., Nuttle, H. L. W. (2003). Relaxed conditions for radial-basis function networks to be universal approximators. Neural Networks, 16, 1019-1028.