23.10.2024

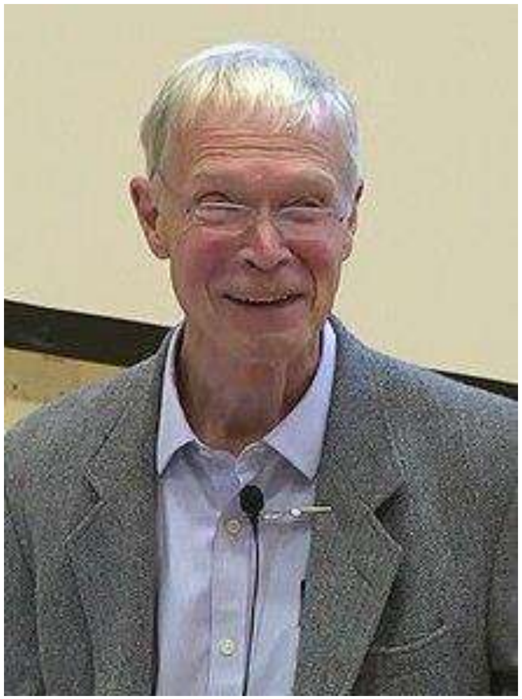
BITS F464: Machine Learning (1st Sem 2024-25)

NEURAL NETWORKS: PERCEPTRON, MLP, BACKPROPAGATION

Chittaranjan Hota, Sr. Professor
Dept. of Computer Sc. and Information Systems
hota@hyderabad.bits-pilani.ac.in

Artificial Neural Networks

Geoffrey Everest Hinton



John Joseph Hopfield

Who is Godfather of AI?

ANNs: Motivating Examples

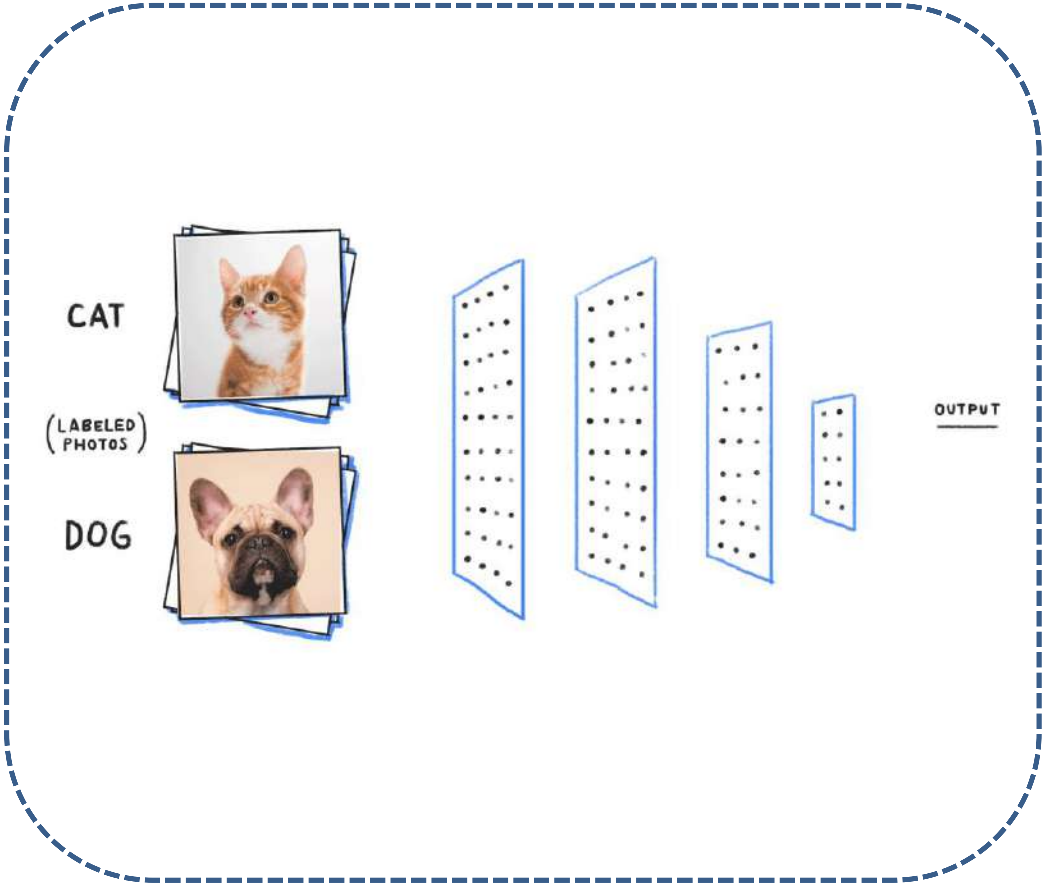
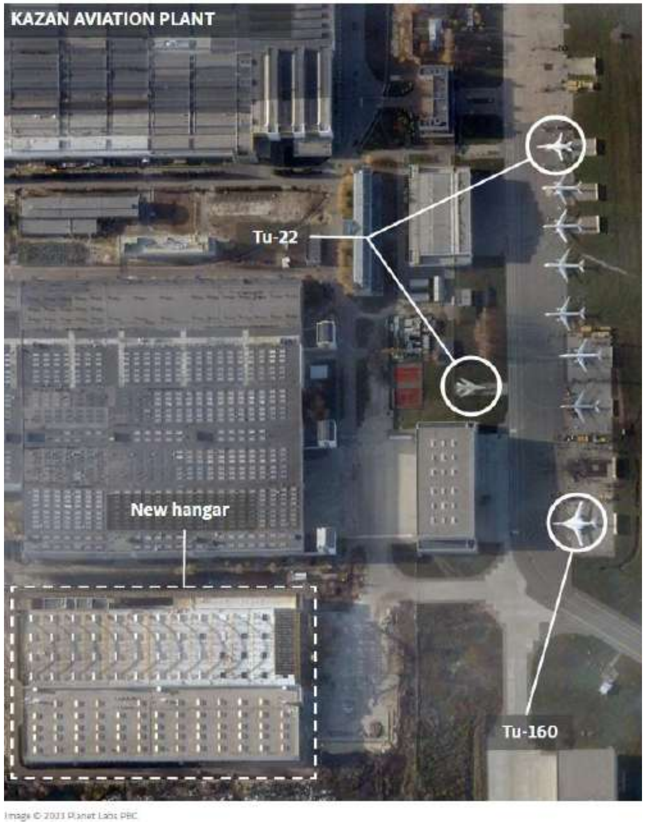


Image source: https://towardsdatascience.com/

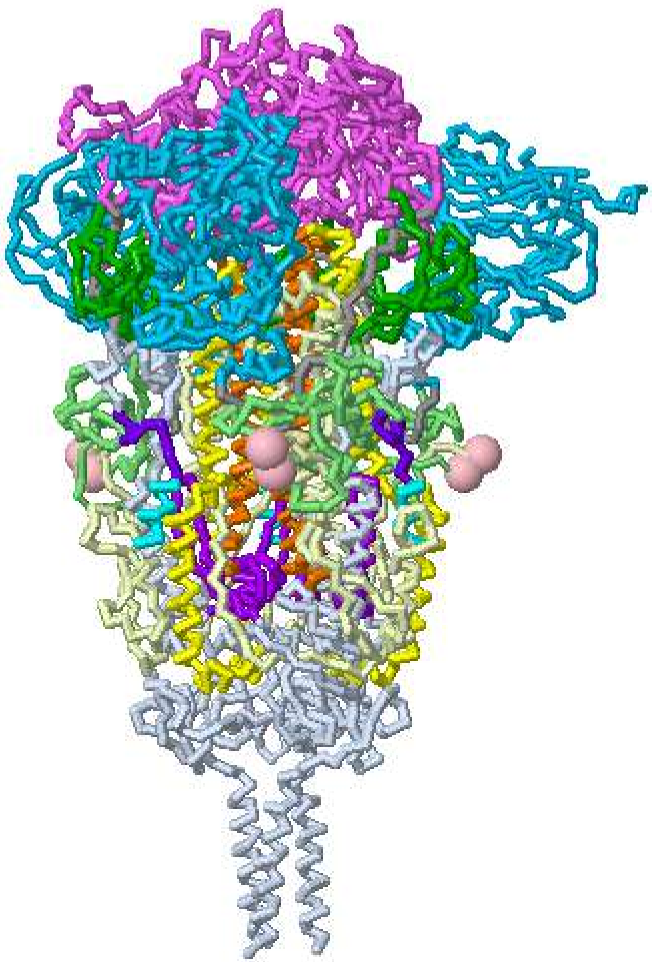


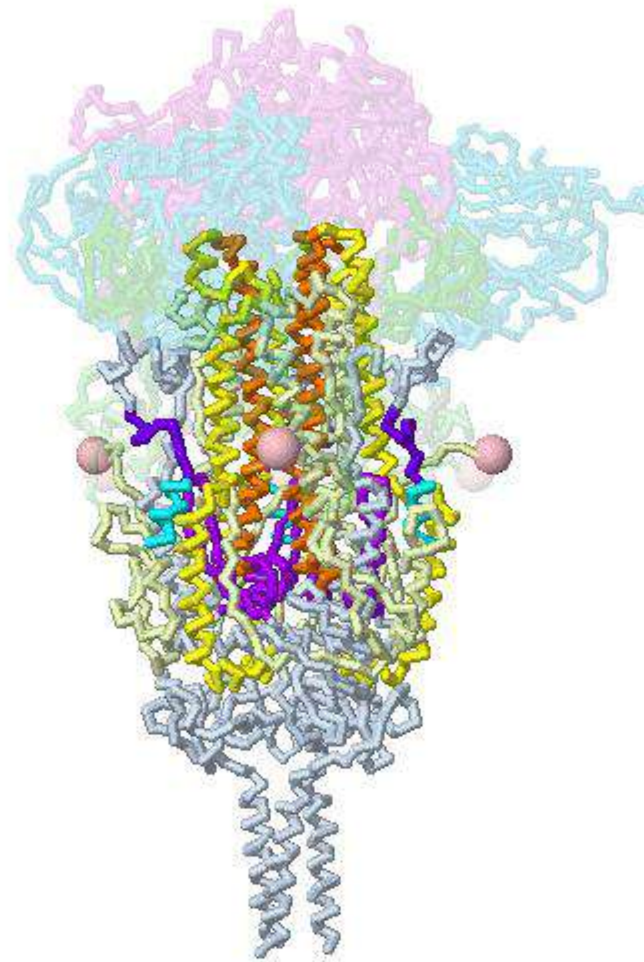
(Russia-Ukrain War: Ukrainians used ANNs to combine ground-level photos, drone video footage and satellite imagery to enhance War Intelligence)

‘The machine did it coldly’: Israel used AI to identify 37,000 Hamas targets: The Lavender, The Gospel.

Source: https://www.ri.cmu.edu/

Pre-Fusion Spike Protein





https://proteopedia.org/ SARS-CoV-2 Alphafold 2

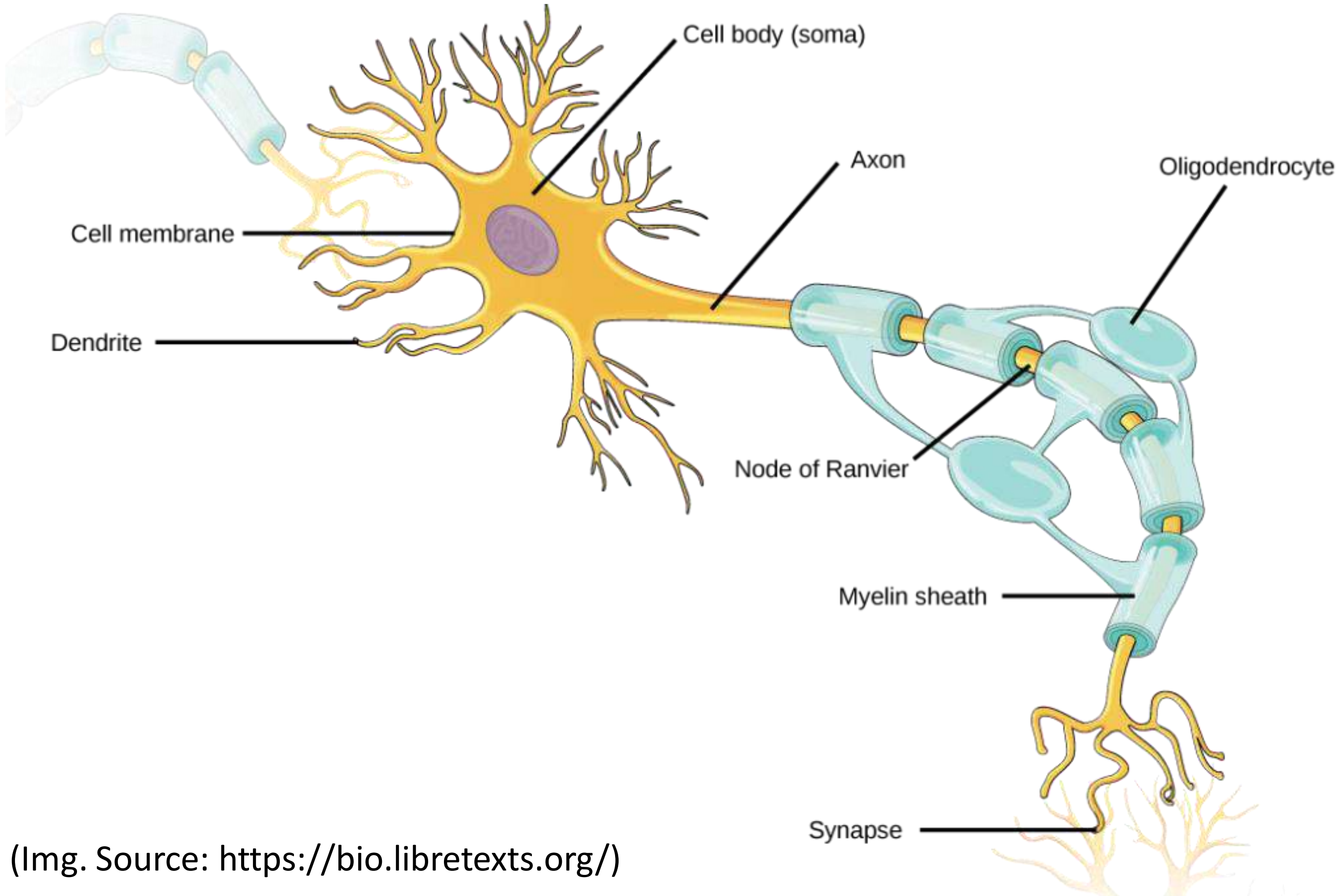
Learning Rewires the Brain

An electrical signal shooting down a nerve cell and then off to others in the brain. Learning strengthens the paths that these signals take, essentially "wiring" certain common paths through the brain.

How many Neurons in a human brain?

Image Source: https://www.snexplores.org/ (imagination)

A Nerve Cell: Neuron

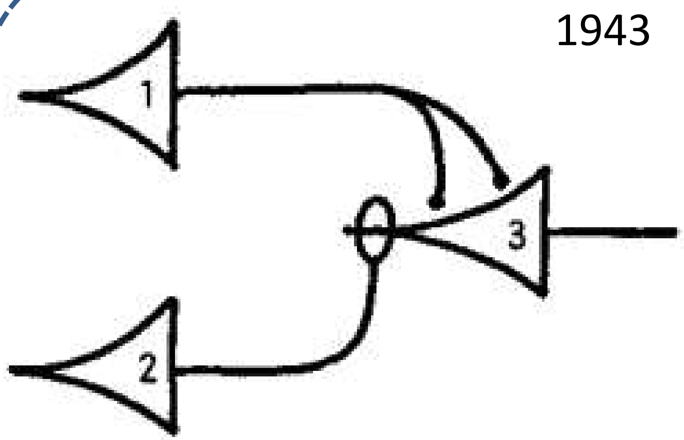


(Img. Source: https://bio.libretexts.org/)

What are their computational abstractions in an Artificial Neural Network?

Perceptron: Modelling the Nerve cell

1943



$$N_3(t) \;.\equiv.\; N_1(t-1) \,.\sim N_2(t-1)$$

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

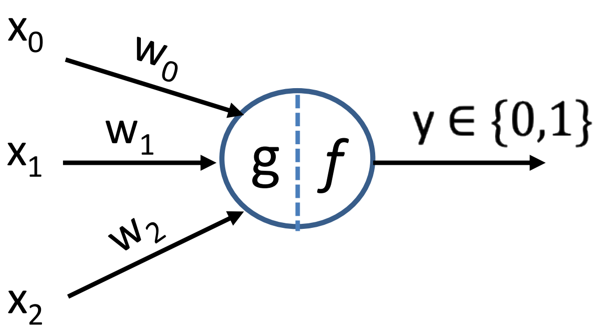
FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

Frank Rosenblatt at IBM 704 (Electronic profile analyzing computer): a precursor to the perceptron, 1958.

(Image source: https://news.cornell.edu)

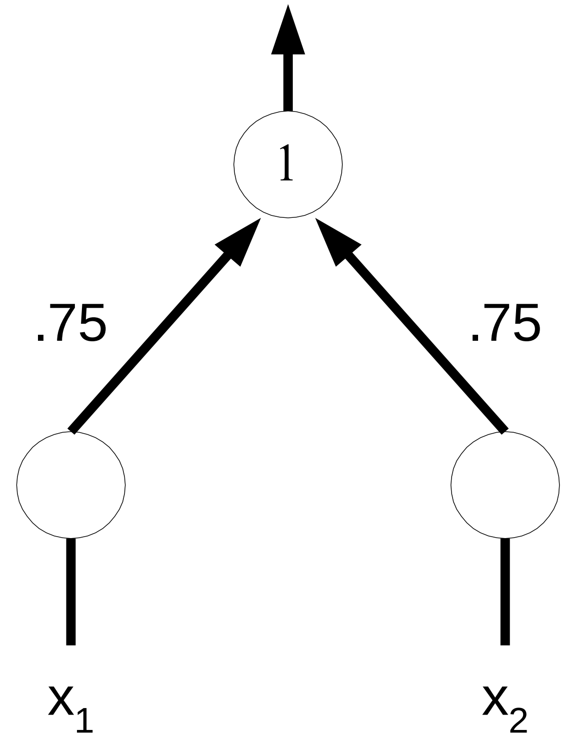
Perceptron: An Example

$$g(x_1, x_2, \ldots, x_n) = g(x) = \sum_{i=1}^{n} w_i x_i$$



$x_n \in \{0,1\}$

$$y = f(g(x)) \begin{cases} = 1, & \text{if } g(x) \geq \theta \\ = 0, & \text{if } g(x) < \theta \end{cases}$$

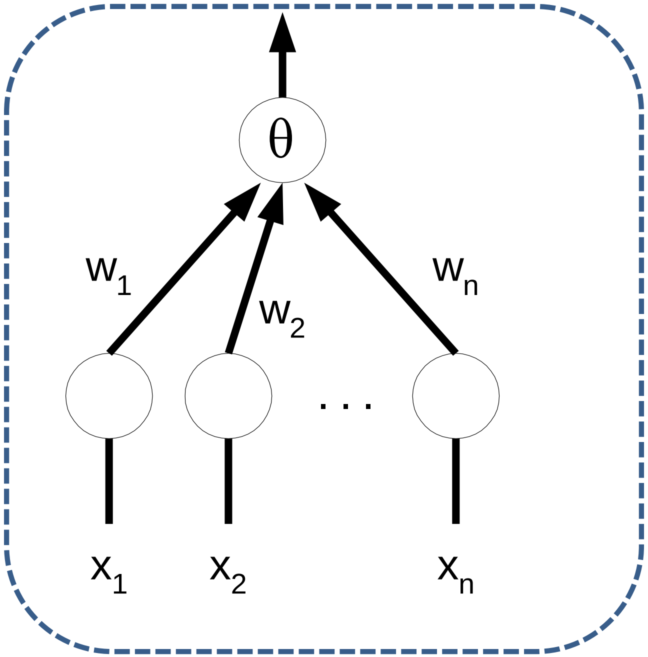


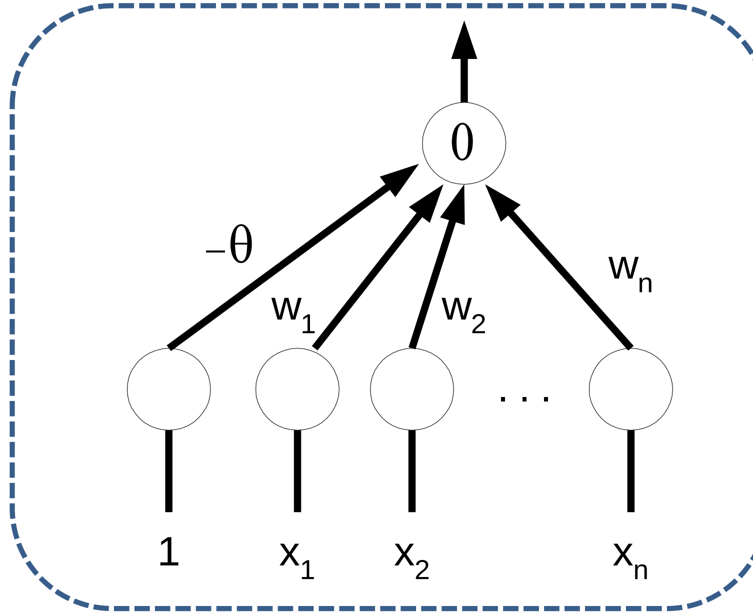
What function this neuron computes?

Normalizing thresholds

- Why do we need Normalization?

$$y(x) = w_0 + w_1x_1 + w_2x_2, \cdots, w_nx_n = w_0 + \sum_{i=1}^{n} w_ix_i$$

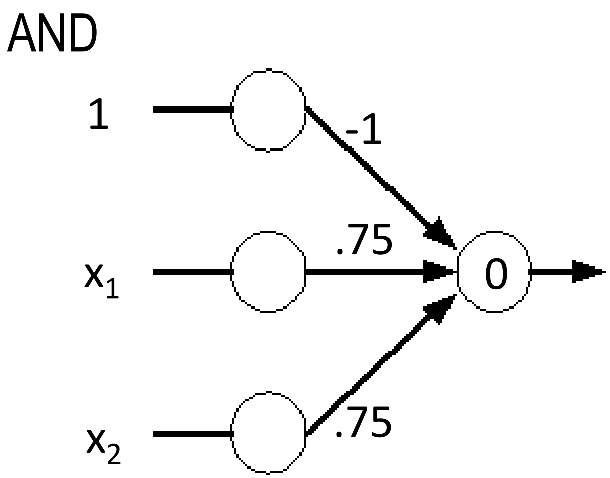


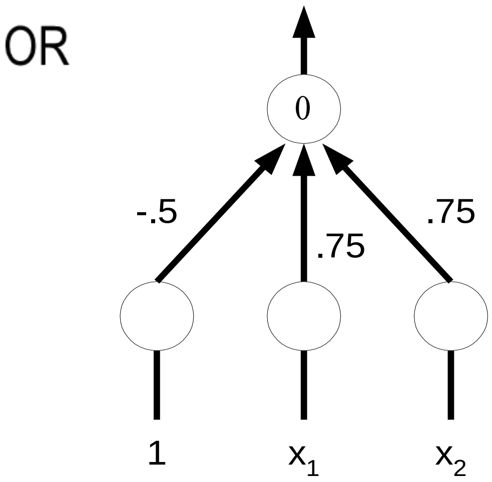


Advantage: threshold = 0 for all neurons:

$$y = f(g(x)) = 1, \quad \text{if } -\boldsymbol{\theta}X1 + \sum_{i=1}^{n} w_i x_i \geq 0$$
$$= 0, \quad \text{Otherwise}$$

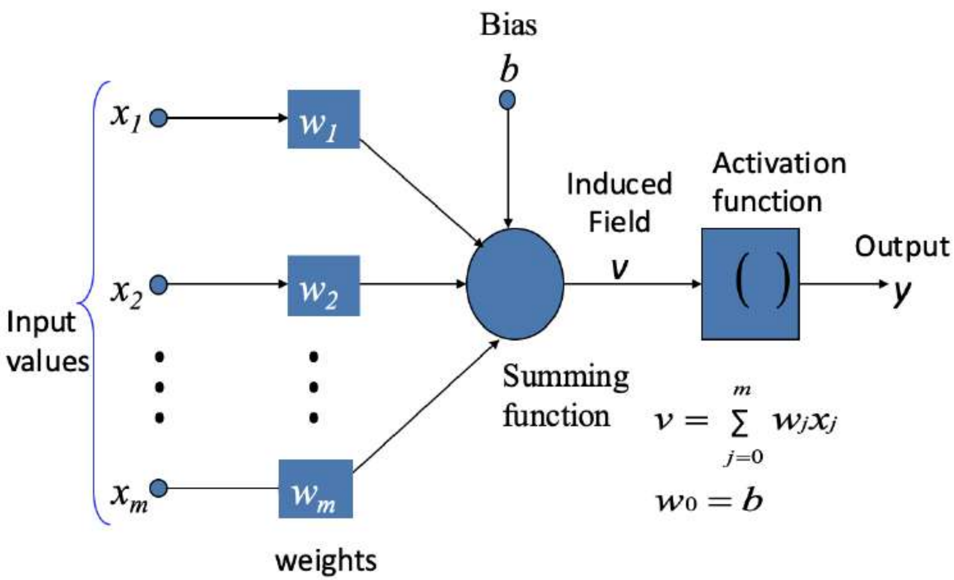
Normalized examples

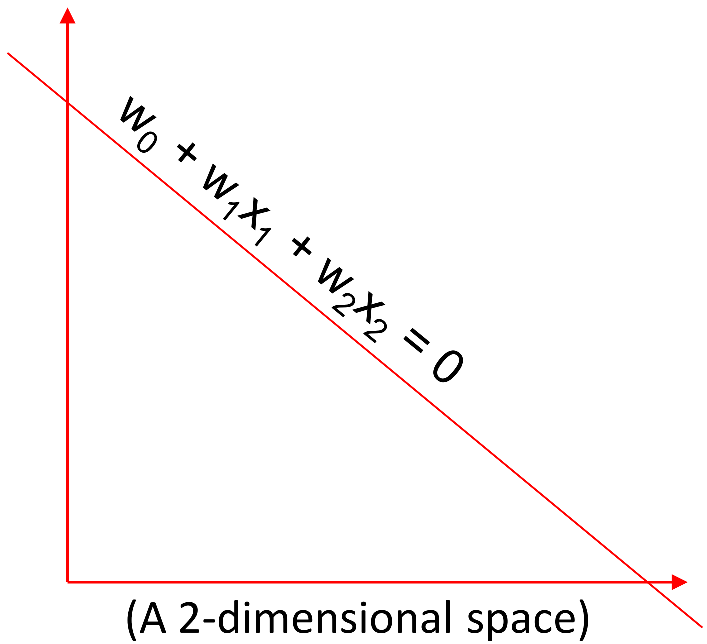




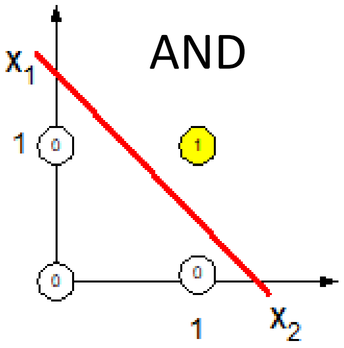
Perceptron as a Decision Surface

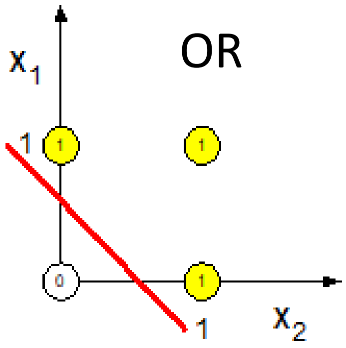
- What type of Classifier?





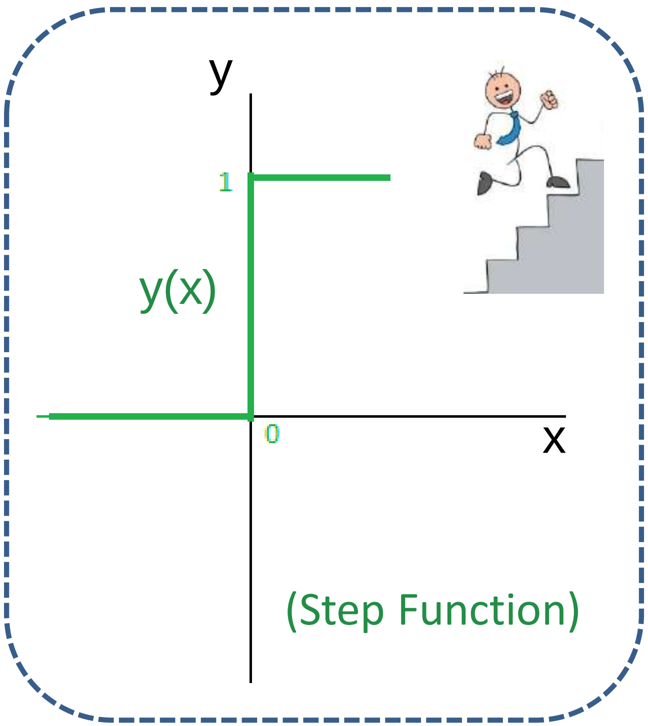
- Can it solve Non-linear classification problems?

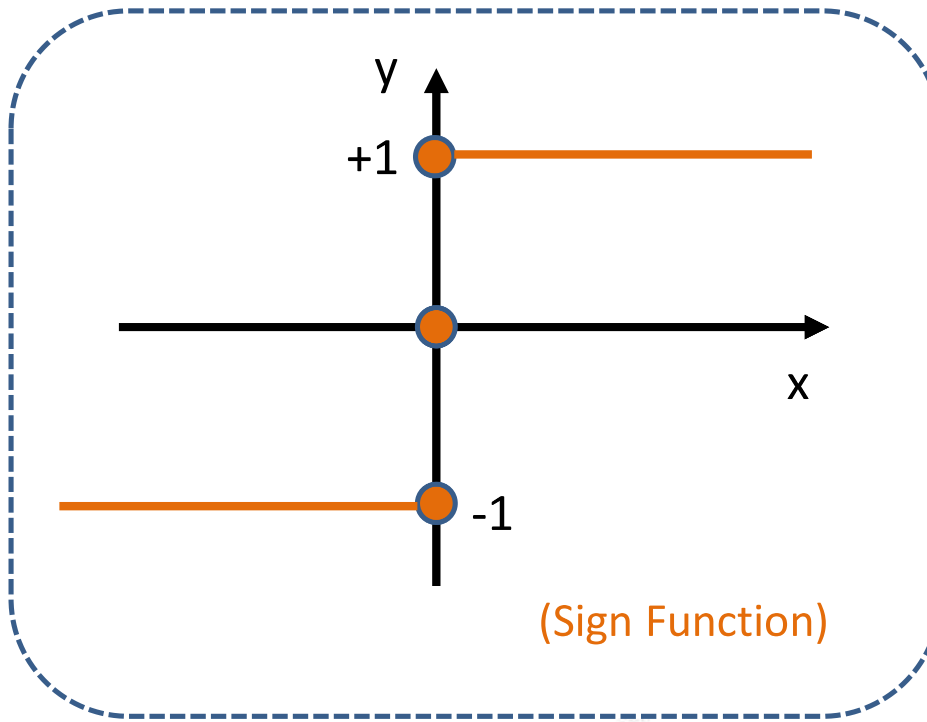




XOR?

Activation Functions in a Perceptron





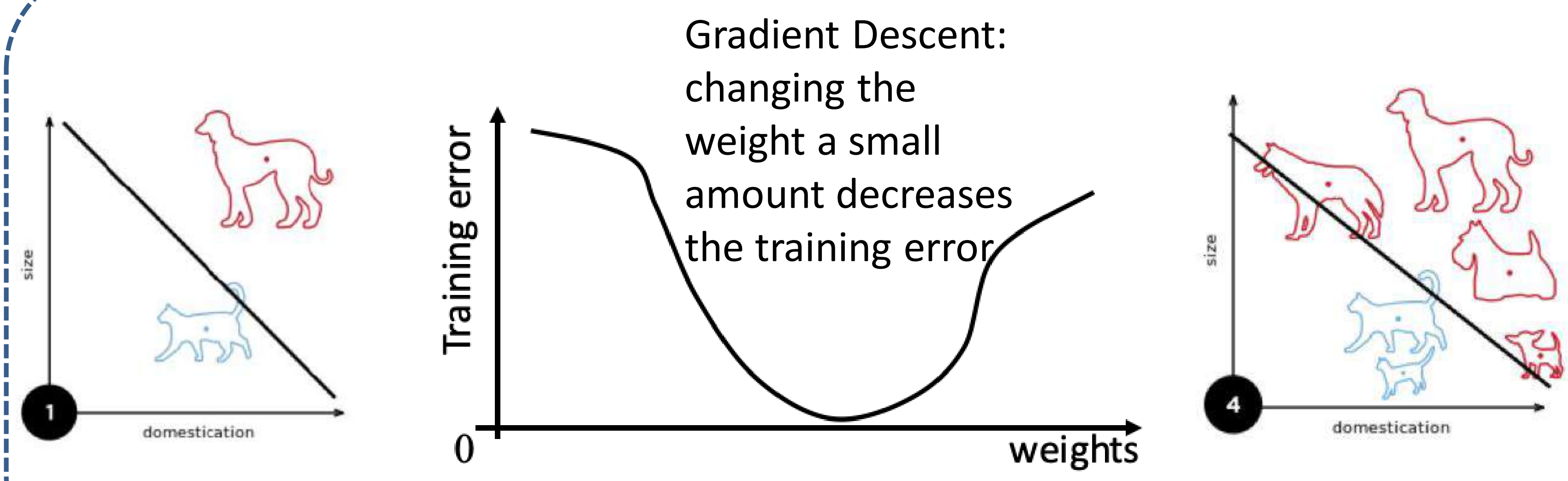
$$y(x) = 1, \quad \text{if } -\boldsymbol{\theta} X1 + \sum_{i=1}^{n} w_i x_i \geq 0$$
$$= 0, \text{ Otherwise}$$

$$y(x) \begin{cases} = 1, & \text{if } -\boldsymbol{\theta} X1 + \sum_{i=1}^{n} w_i x_i > 0 \\ = 0, & \text{for equal to } 0 \\ = -1, & \text{for less than } 0 \end{cases}$$

What about other activation functions like Sigmoid ?

Multi-layer Perceptron

Perceptron Training Example



An example: A perceptron updating its linear boundary as more training examples are added. (Image Source: Wiki)

Perceptron Training Algorithm

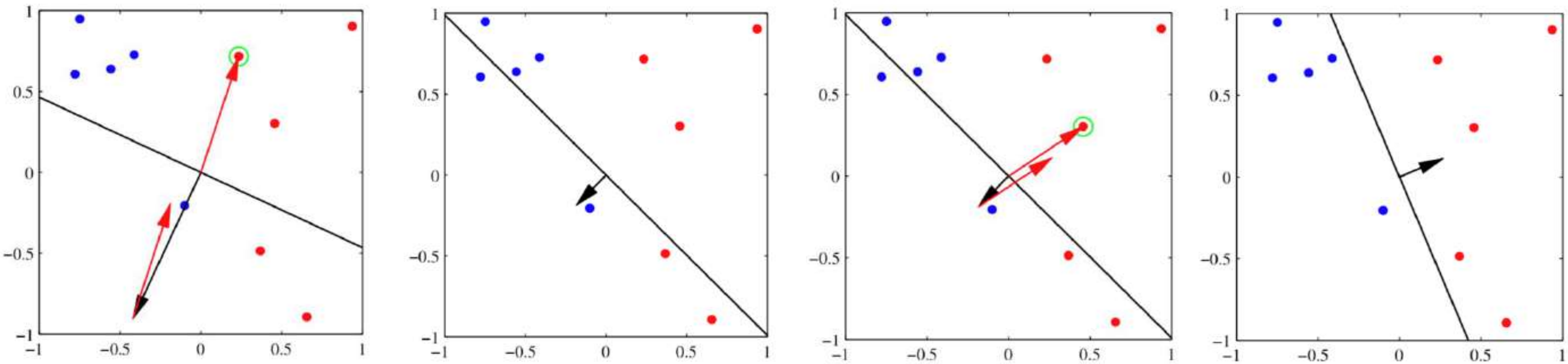


Image Source: PRML, Bishop

Algorithm: Perceptron Learning Algorithm

```
P ← inputs with label +
N ← inputs with label –
Initialize w ← Random value;
while (!convergence) do
  Pick random x ∈ P U N;
  if (x ∈ P && w.x <0) then
    w = w + x;
  endif;
  if (x ∈ N && w.x ≥ 0) then
    w = w - x;
  endif;

endwhile;

// Algorithm converges when all the inputs are classified correctly.
```

Perceptron Training Rule

$$w_i \leftarrow w_i + \boxed{\Delta w_i} \longrightarrow \eta\,(t - o)\,x_i$$

Where,

- t = target value
- o = perceptron output
- η a small constant (e.g, 0.1) called the learning rate.

Let us see this through an example:

When all (η , (t-o) and x_i) are positive, w_i will increase and vice versa:

$x_i = 0.8$, $\eta = 0.1$, $t = 1$, $o = -1$:

$\rightarrow \Delta w_i = 0.1(1-(-1))0.8 = 0.16 \uparrow$

If t = -1, o = 1, what will happen?

Why should this update rule converge toward successful weight values?

If training data is linearly separable and η is sufficiently small.

It updates the weights only when the predicted class is incorrect. The updates are based on whether the example is misclassified (binary classification).

Run through the algorithm: AND

$x_1 = 1, x_2 = 1 \rightarrow 1$
$x_1 = 1, x_2 = 0 \rightarrow 0$
$x_1 = 0, x_2 = 1 \rightarrow 0$
$x_1 = 0, x_2 = 0 \rightarrow 0$

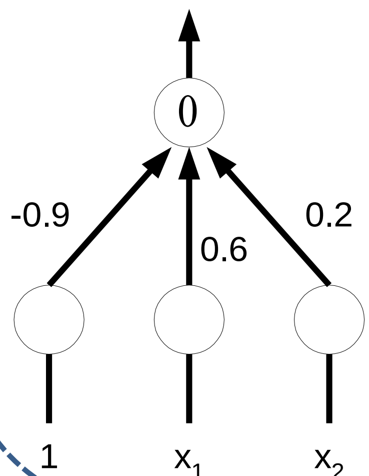
(Training Set)

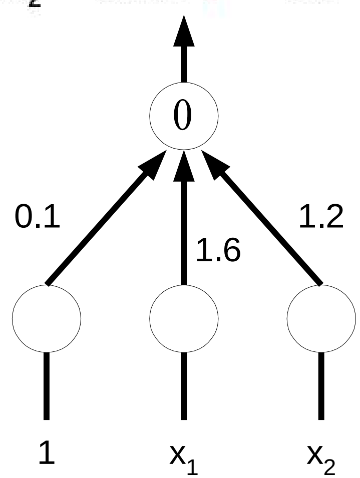
$x_1 = 1, x_2 = 1$:	?			
$x_1 = 1, x_2 = 0$:	$-0.9*1 + 0.6*1 + 0.2*0$	$= -0.3 \rightarrow 0$	OK	
$x_1 = 0, x_2 = 1$:	$-0.9*1 + 0.6*0 + 0.2*1$	$= -0.7 \rightarrow 0$	OK	
$x_1 = 0, x_2 = 0$:	$-0.9*1 + 0.6*0 + 0.2*0$	$= -0.9 \rightarrow 0$	OK	

$w_0 = -0.9 + 1 = 0.1$
$w_1 = 0.6 + 1 = 1.6$
$w_2 = 0.2 + 1 = 1.2$

New Weights

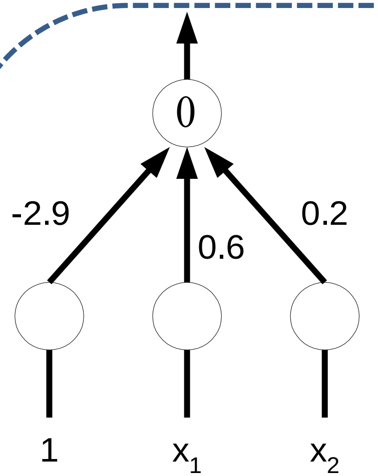
$w_0 = 0.1 - 1 - 1 - 1 = -2.9$
$w_1 = 1.6 - 1 - 0 - 0 = 0.6$
$w_2 = 1.2 - 0 - 1 - 0 = 0.2$





$x_1 = 1, x_2 = 1$:	$0.1*1 + 1.6*1 + 1.2*1$	$= 2.9 \rightarrow 1$	OK
$x_1 = 1, x_2 = 0$:	$0.1*1 + 1.6*1 + 1.2*0$	$= 1.7 \rightarrow 1$	WRONG
$x_1 = 0, x_2 = 1$:	$0.1*1 + 1.6*0 + 1.2*1$	$= 1.3 \rightarrow 1$	WRONG
$x_1 = 0, x_2 = 0$:	$0.1*1 + 1.6*0 + 1.2*0$	$= 0.1 \rightarrow 1$	WRONG

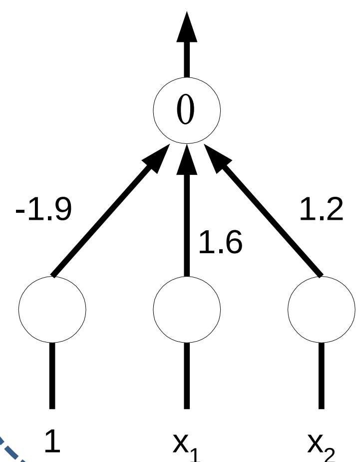
Continued…



$x_1 = 1, x_2 = 1$:	-2.9*1 + 0.6*1 + 0.2*1	= -2.1 →	0	WRONG
$x_1 = 1, x_2 = 0$:	-2.9*1 + 0.6*1 + 0.2*0	= -2.3 →	0	OK
$x_1 = 0, x_2 = 1$:	-2.9*1 + 0.6*0 + 0.2*1	= -2.7 →	0	OK
$x_1 = 0, x_2 = 0$:	-2.9*1 + 0.6*0 + 0.2*0	= -2.9 →	0	OK

$w_0 = -2.9 + 1 = -1.9$
$w_1 = 0.6 + 1 = 1.6$
$w_2 = 0.2 + 1 = 1.2$

→ New Weights



$x_1 = 1, x_2 = 1$:	-1.9*1 + 1.6*1 + 1.2*1	= 0.9 →	1	OK
$x_1 = 1, x_2 = 0$:	-1.9*1 + 1.6*1 + 1.2*0	= -0.3 →	0	OK
$x_1 = 0, x_2 = 1$:	-1.9*1 + 1.6*0 + 1.2*1	= -0.7 →	0	OK
$x_1 = 0, x_2 = 0$:	-1.9*1 + 1.6*0 + 1.2*0	= -1.9 →	0	OK

Convergence Reached. Halt! $w_0 = -1.9$, $w_1 = 1.6$, $w_2 = 1.2$

Gradient Descent and the Delta Rule

- If the training examples are NOT linearly separable (which the Perceptron rule cannot handle), the Delta rule converges towards a best-fit approximation to the target concept.

- The key-idea behind the delta rule is to use Gradient descent, a basis for Back-propagation algorithm.

- Delta rule is best understood by considering an un-thresholded Perceptron, i.e. a linear unit without threshold (or activation function).

- Let the linear unit be characterized by: $o = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$

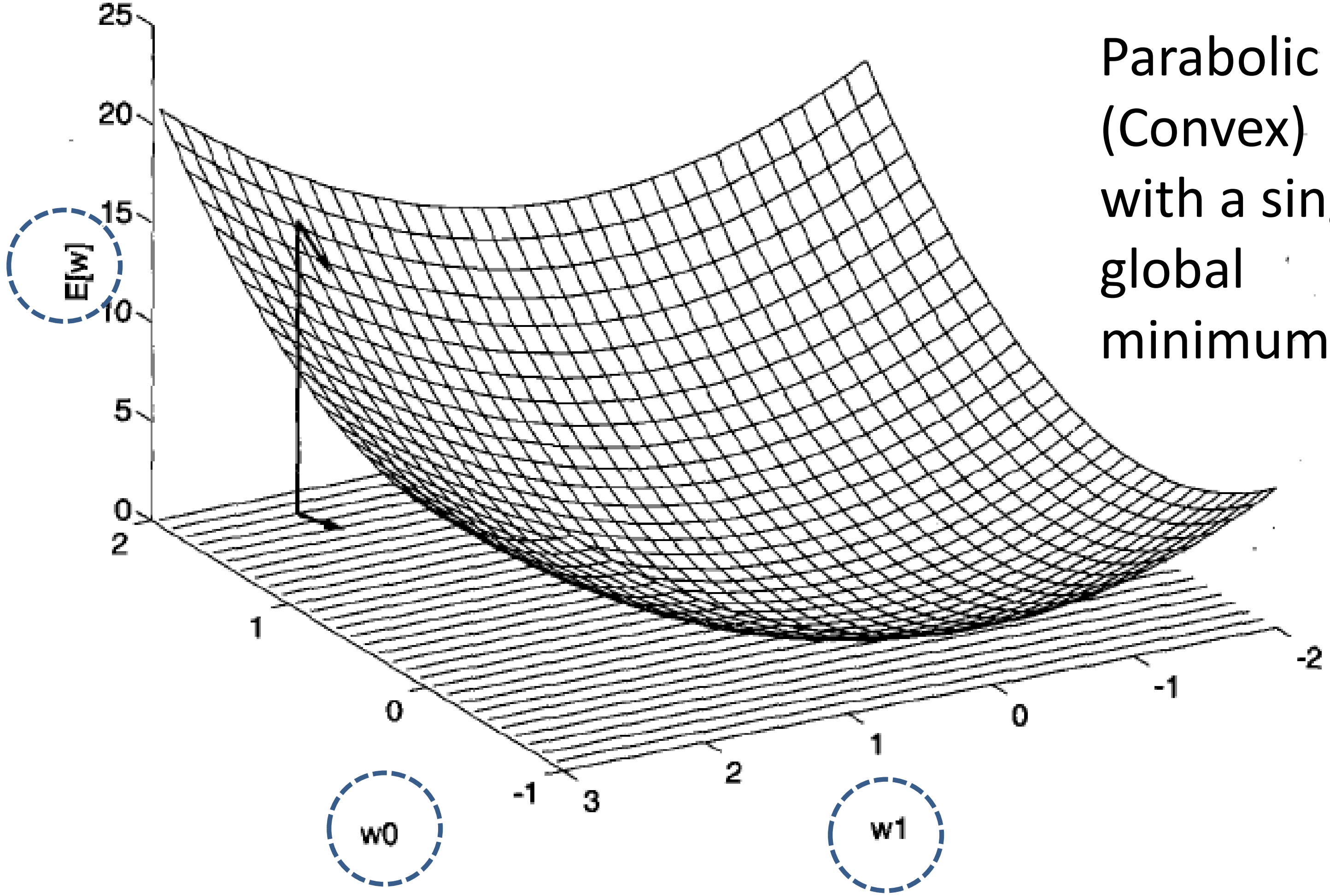
- Let us learn w_i's that minimize the squared error: $E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$

ADALINE: adaptive linear neural network based on MSE. Or Least mean square (LMS) Widrow Hoff

---

It updates the weights even if the predicted value is close to the correct one but not exactly right. The adjustment is proportional to the error, making it suitable for continuous outputs.

Visualizing Gradient Descent: Recap



Parabolic (Convex) with a single global minimum.

w_0 and w_1: The two weights of a linear unit and E is the error.

Derivation of Gradient Descent: Recap

- How can we calculate the direction of s
surface?
- Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

- When interpreted as a vector in weig
the direction that produces the steepes
- The negative of this vector therefore
decrease.
- The training rule: $\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$ Where:

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}] \quad \Delta w_i = -\eta\frac{\partial E}{\partial w_i} \quad (1)$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \quad (2)
\end{aligned}$$

Substituting (2) in (1):

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)\, x_{id}$$

Gradient Descent & Stochastic Gradient Descent

1. Initialize each w_i to some small random value
2. Until the termination condition is met {
    1. Initialize each Δw_i to 0
    2. For each training example do {
        1. Input the instance to the Linear unit and compute output ‘o’
        2. For each Linear unit weight w_i do

           $$\boxed{\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i} \rightarrow w_i \leftarrow w_i + \eta(t - o)x_i$$
        3. }
    3. For each Linear unit weight w_i do {

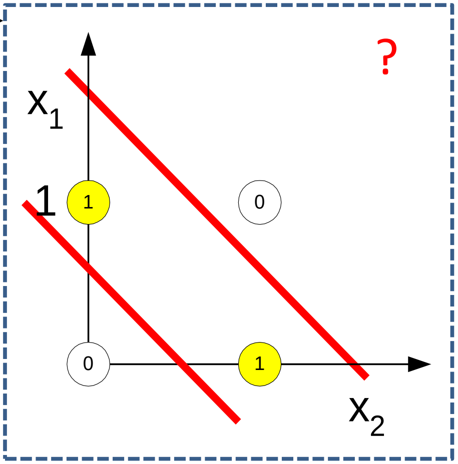
       $$w_i \leftarrow w_i + \Delta w_i$$
    4. }

    X
3. } Alternatively, SGD computes ‘E’ for each training ex: $E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$

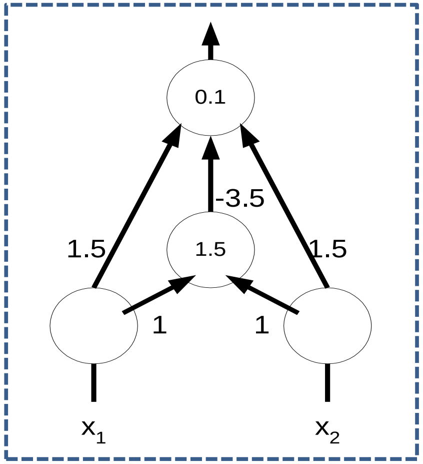
GD: the error is summed over all examples before updating weights. It might miss global minima when multiple local minima are present. In SGD/Incremental GD, weights are updated upon examining each training example.

Inadequacy of Perceptron

- Many simple problems are NOT linearly separable.



- Output is in the form of binary (0 or 1), NOT in the form of continuous values or probabilities.
- No memory and hence treat each input independently. Hence, limited ability to understand sequential or temporal patterns in data.



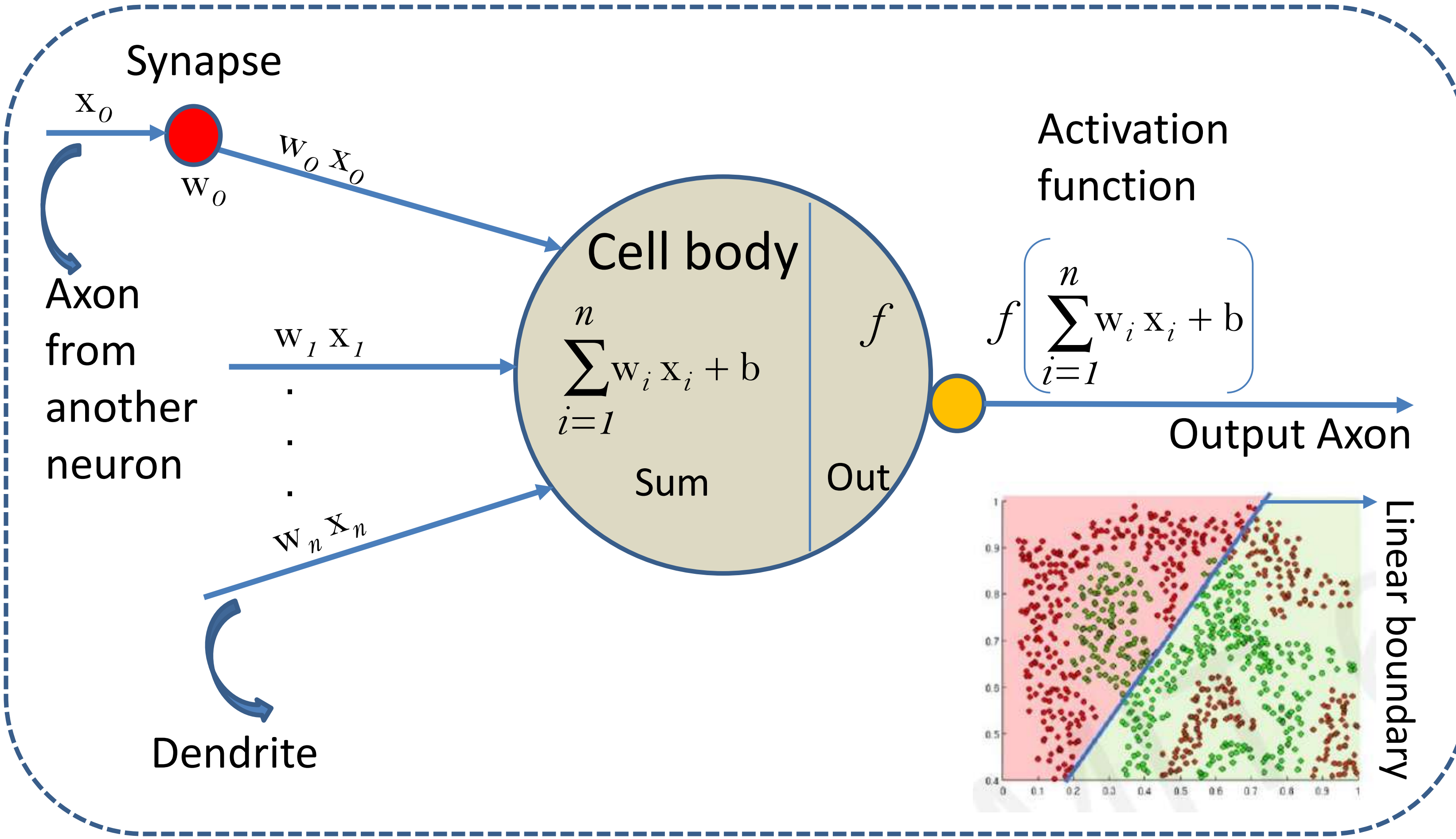
However, you can compute XOR by introducing a new, hidden unit as shown in the left.

Every classification problem has a Perceptron solution if enough hidden layers are used.

How to build such a multi-layer network?

Minsky & Papert's paper: Pretty much killed ANN research in 1970. Rebirth in 1980: faster parallel computers, newer algorithms (BPN,…), newer architectures (Hopfield nets).

Recap: A Perceptron

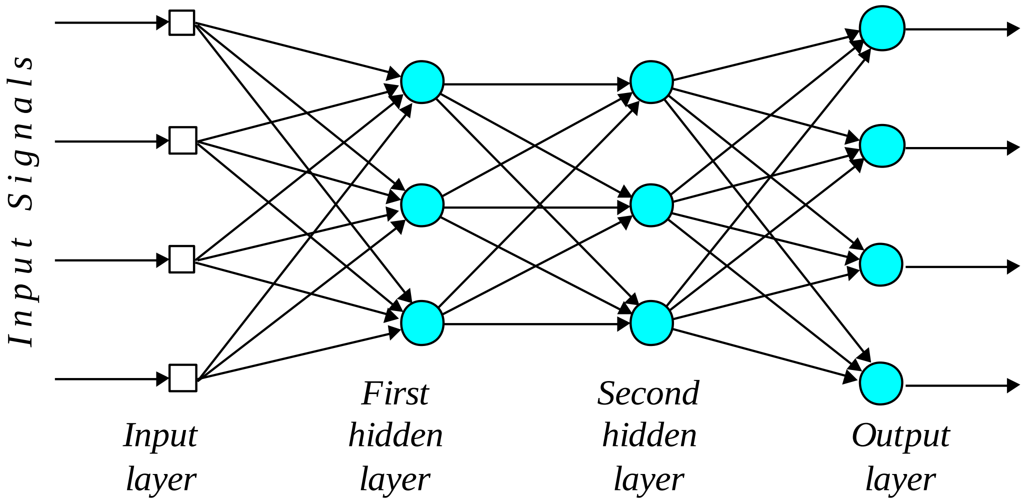


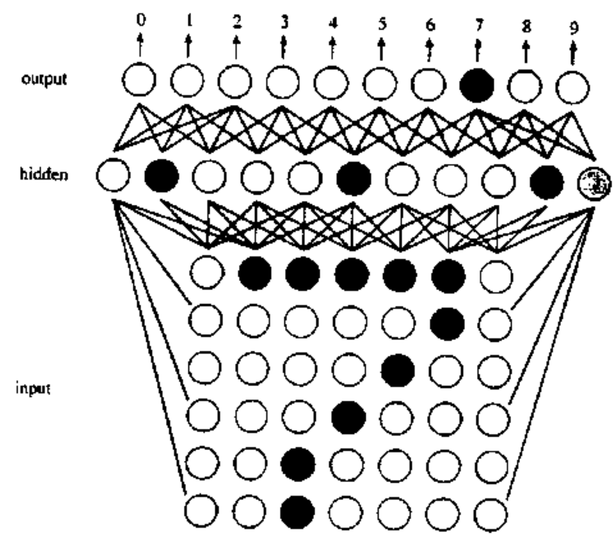
Modelling Mathematically a Neuron

Hidden units in a Multi-layer Perceptron (MLP)

- The addition of hidden units allows the network to develop complex feature detectors (i.e., internal representations)

- e.g., OCR

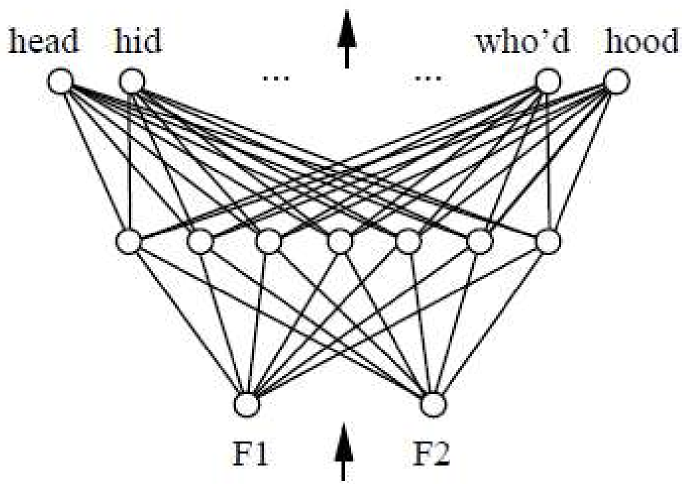


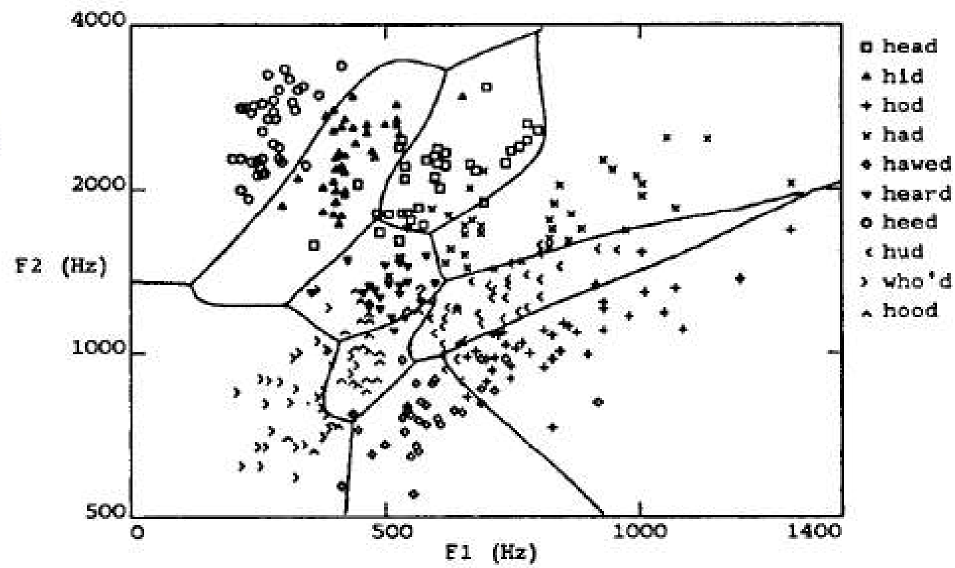


What does a hidden layer hide?

No. of nodes in a layer and no. of layers? Nodes too few: can't learn, Too many: poor generalization Expt. & tuning.

Decision Surface in a Multilayer Network: An Ex.



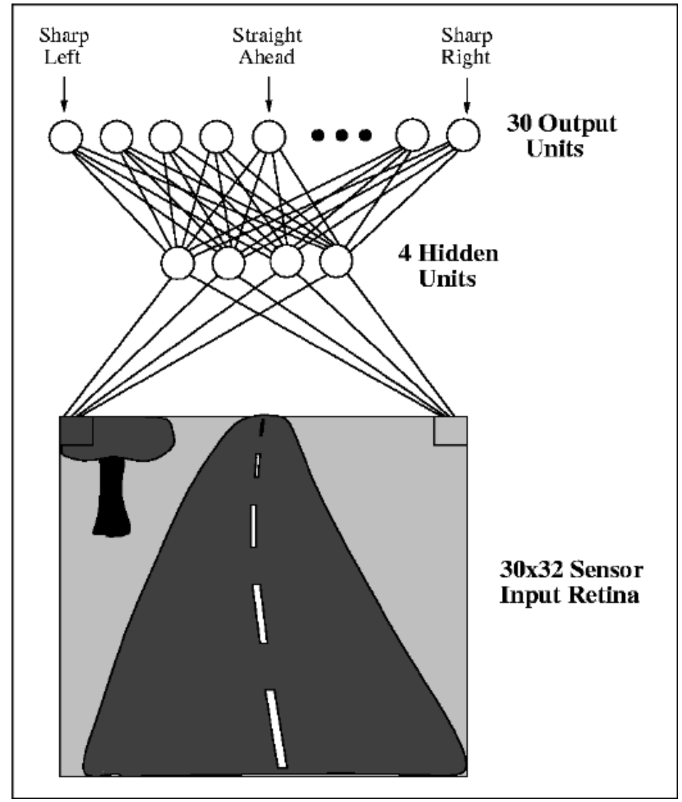


Input to the Network: two features from spectral analysis of a spoken sound

Output: vowel sound occurring in the context "h__d"

Image source: Tom Mitchell's Text

ALVINN: An Autonomous Land Vehicle In a Neural Network

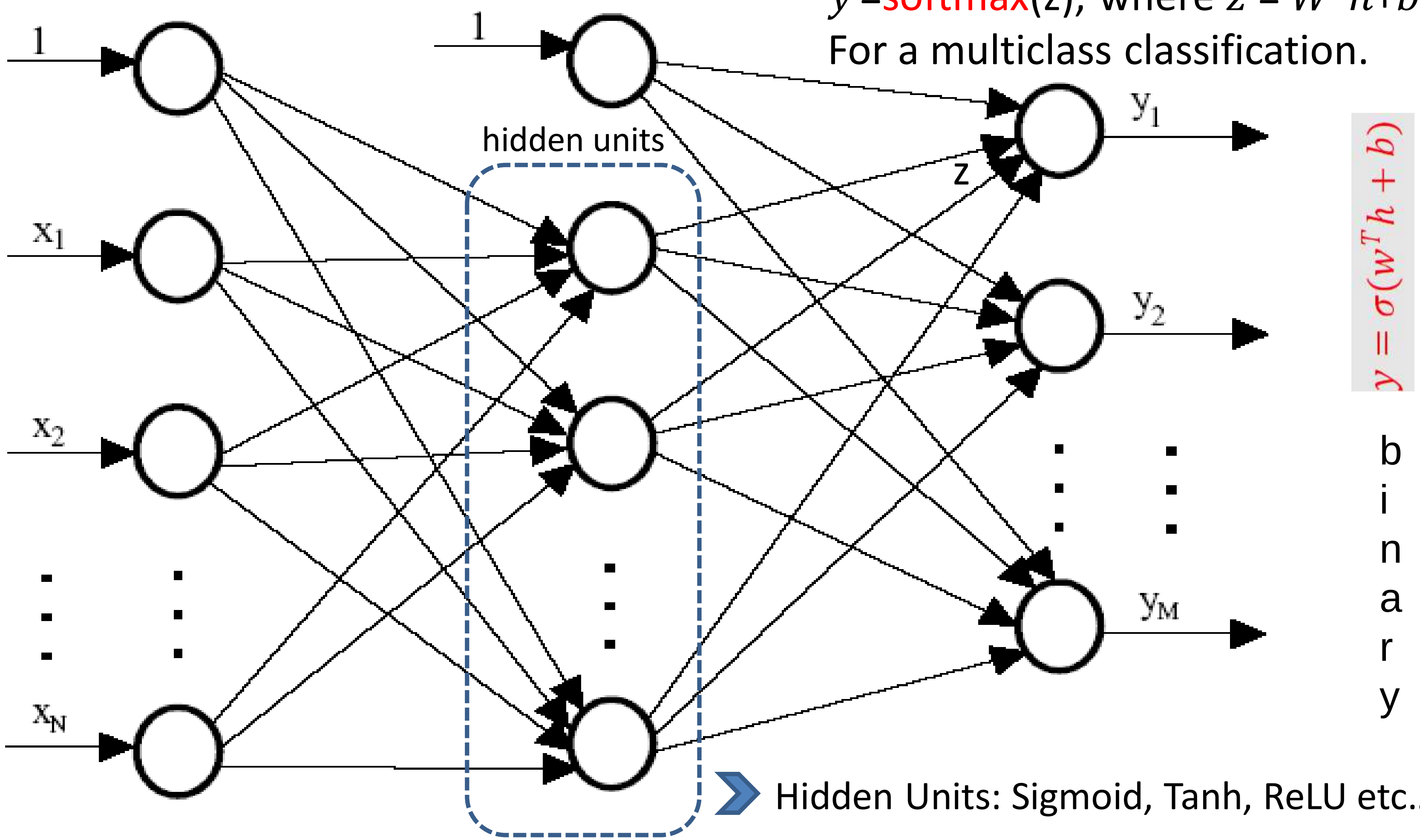


Source: https://www.ri.cmu.edu/ (1989: 3-layer Network)

An application of a Backpropagation Neural Network in smart driving

An Example 3-layer Perceptron

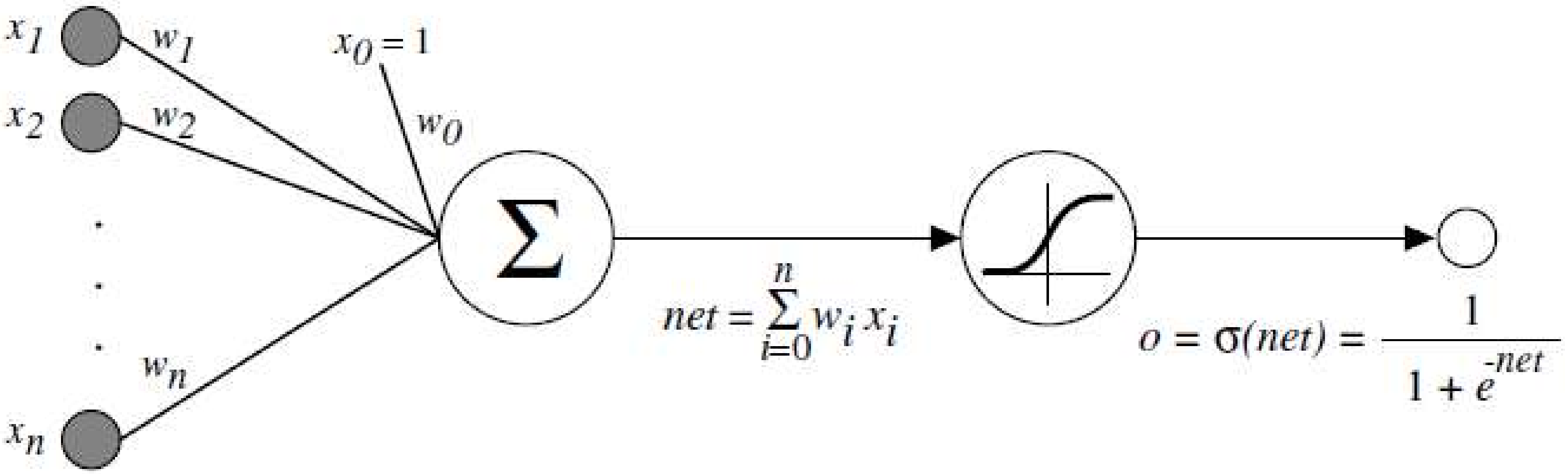
y =softmax(z), where z = $W^{\top}h$+b
For a multiclass classification.



Hidden Units: Sigmoid, Tanh, ReLU etc…

What is Sparse Connectivity and what are its’ Pros and Cons? Leaving out some links.

Activation Function: Sigmoid



$\sigma(x)$ is the $\boxed{\text{sigmoid function}}$

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

During backpropagation, the network might experience:

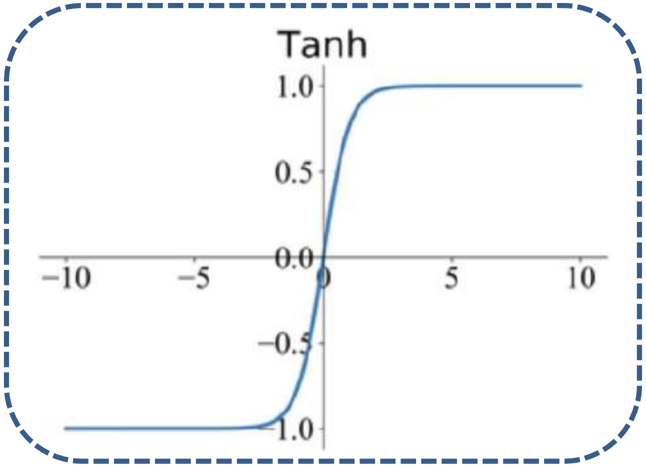
Vanishing Gradient

Where should you worry much? Shallow or Deep NNs?

Activation Function: Tanh

- The hyperbolic tangent (tanh) activation function is another commonly used non-linear activation function in neural networks.
- The tanh function squashes the input values to the range [-1, 1]. It is similar to the sigmoid function, but its output is zero-centered, meaning that its output is centered around zero, unlike the sigmoid function which outputs values between 0 and 1.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Used in RNNs, and LSTMs…

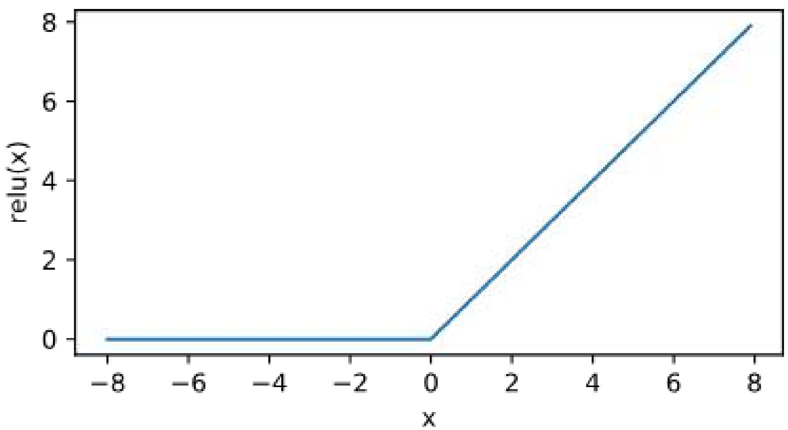
Does it suffer from Vanishing gradient problem?

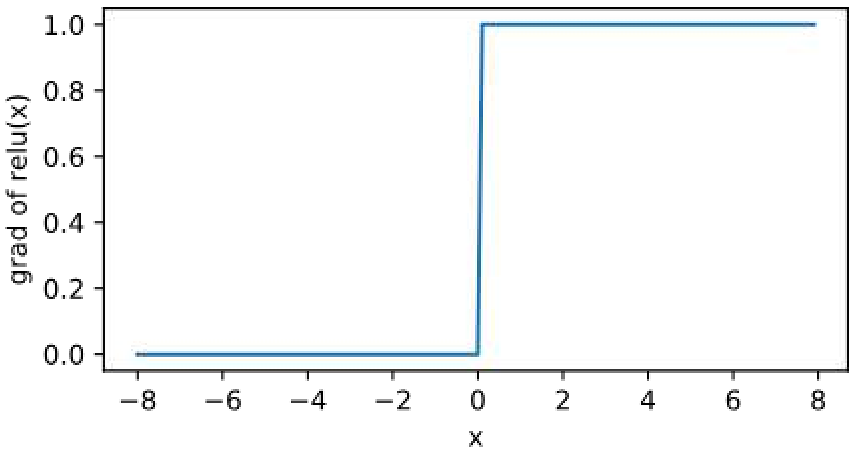
Activation Functions: ReLU

Rectified linear unit function (ReLU) provides a very simple nonlinear transformation:

$$\text{ReLU}(x)=\begin{cases} x, if\ x > 0 \\ o, if\ x \leq 0 \end{cases}$$

What is ReLU'(x)?

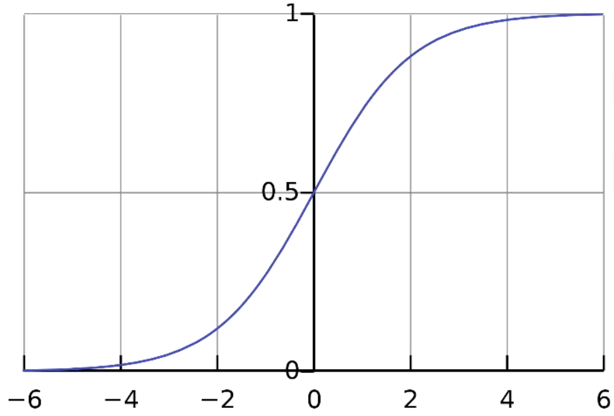




Does it suffer from Vanishing gradient problem?

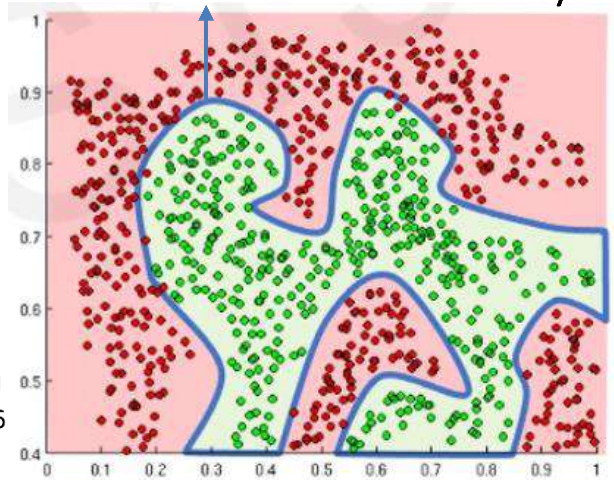
Vanishing Gradient in MLPs/BPNs

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



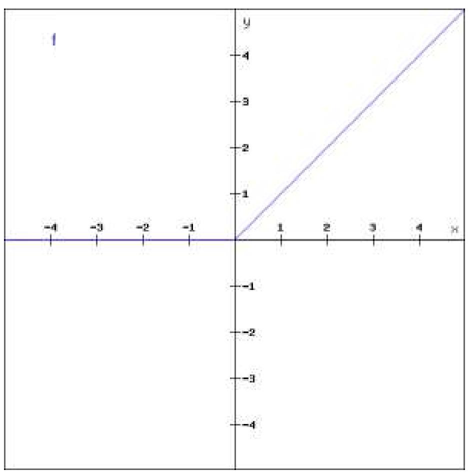
Sigmoid function/ Logistic Curve

Non-linear boundary



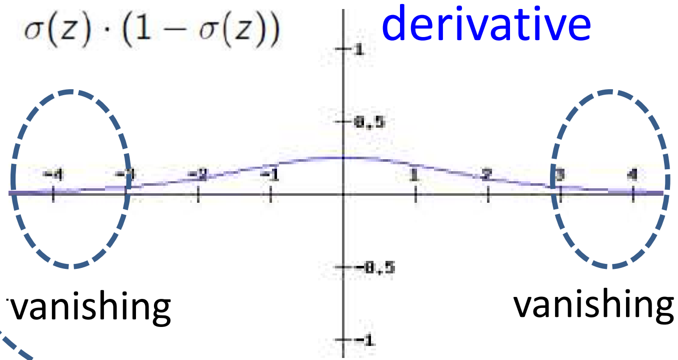
Approx. Arbitrary Complex fun

$\mathrm{ReLU}(z) = \max(0, z)$

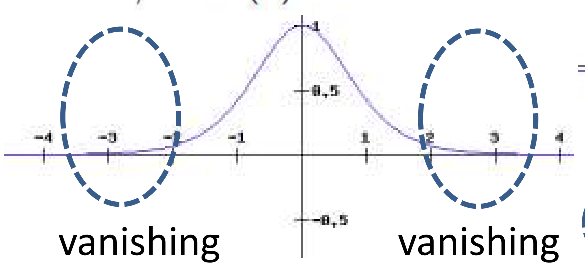


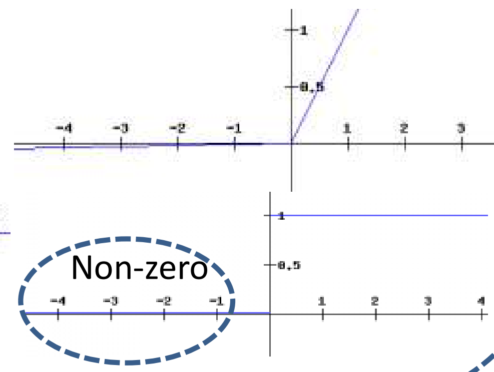
Leaky ReLU(z) = 0.01z, z < 0
= z, z >= 0

$\sigma(z) \cdot (1 - \sigma(z))$ derivative



$1/\cosh^2(z)$





Gradient Descent for Sigmoid Unit

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}
$$

But we know:

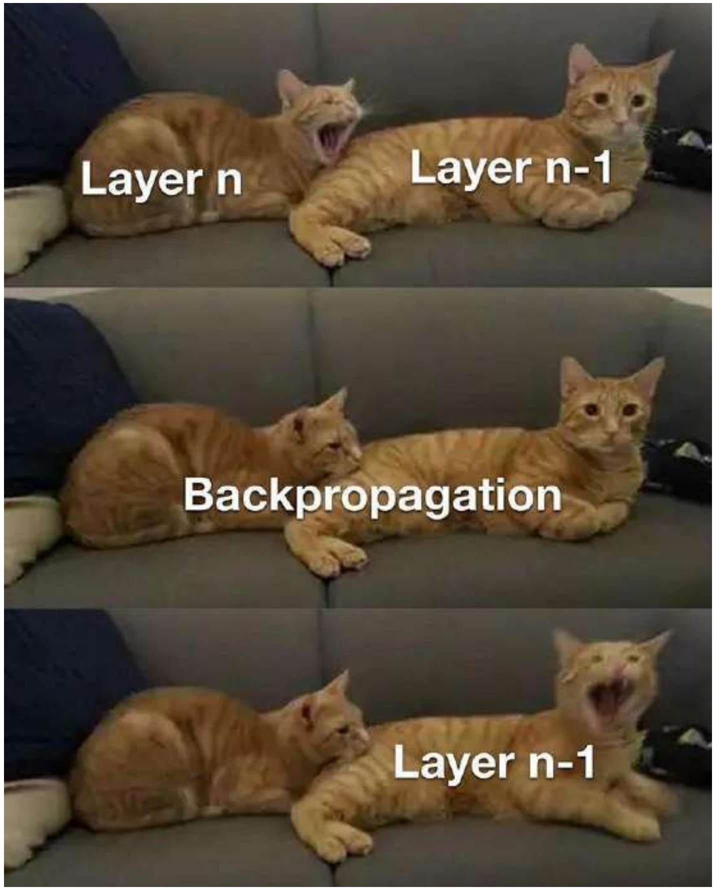
$$
\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)
$$

$$
\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}
$$

$$
\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}
$$

Backpropagation Training Algorithm (BPN)

- Initialize weights (typically random!)
- Keep doing epochs
  - For each example 'e' in the training set do
    - forward pass to compute
      - O = neural-net-output (network, e)
      - miss = (T-O) at each output unit
    - backward pass to calculate deltas to weights
    - update all weights
  - end
- until tuning set error stops improving



Error Backpropagation

- First calculate error of output units and use this to change the top layer of weights.

Current output: o_j=0.2

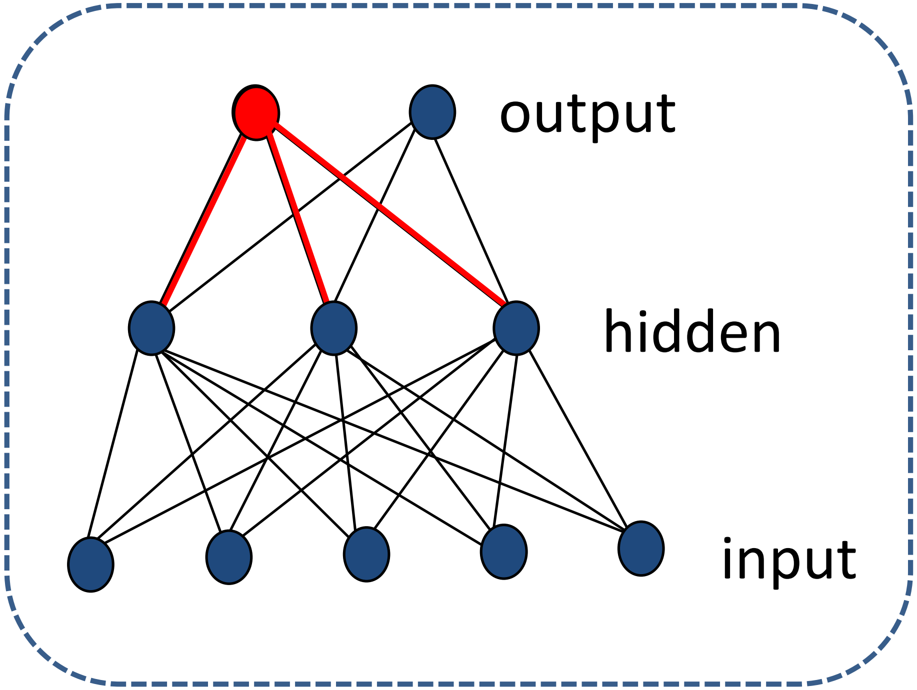
Correct output: t_j=1.0

Error $\delta_j = o_j(1–o_j)(t_j–o_j)$

0.2(1–0.2)(1–0.2)=0.128

Update weights into j

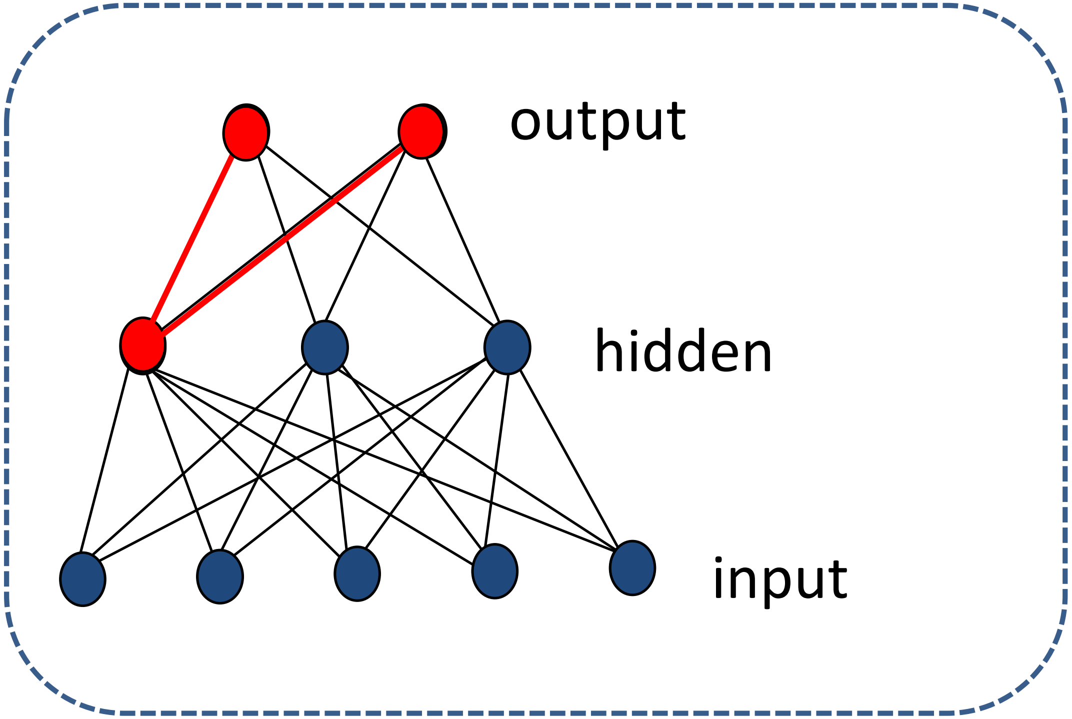
$$\Delta w_{ji} = \eta \delta_j o_i$$



Error Backpropagation continued…

- Next calculate error for hidden units based on errors on the output units it feeds into.

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj}$$



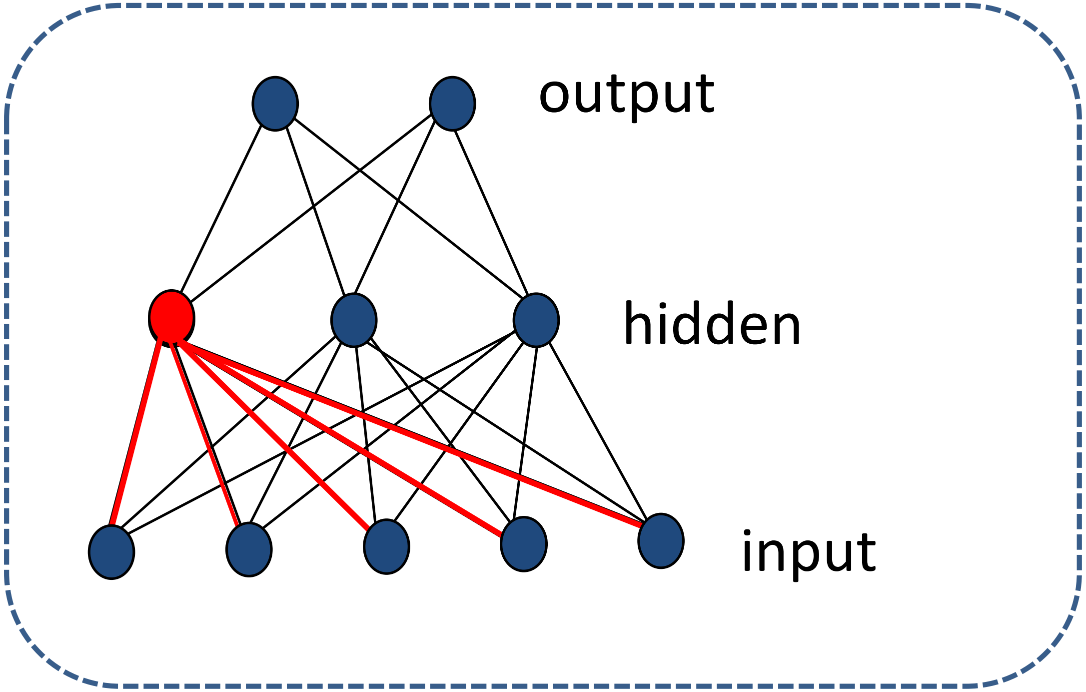
Error Backpropagation continued...

- Finally update bottom layer of weights based on errors calculated for hidden units.

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj}$$

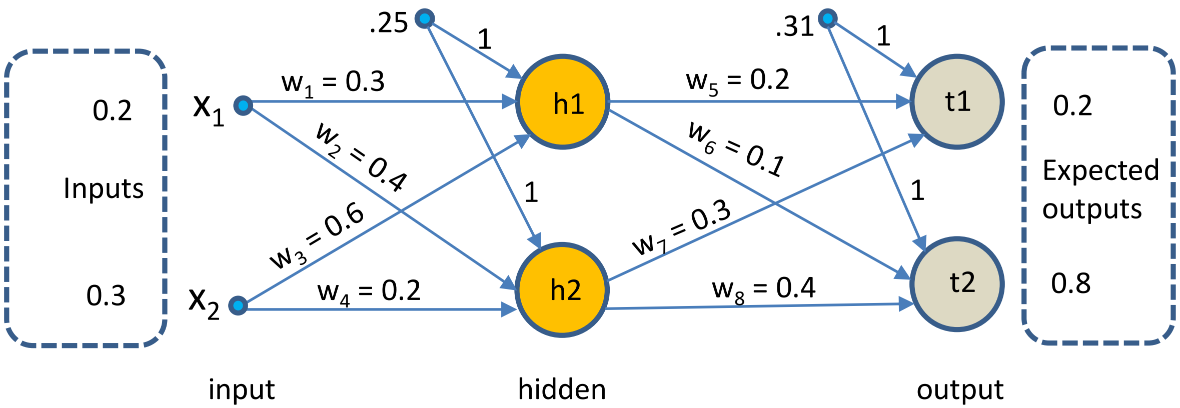
Update weights into *j*

$$\Delta w_{ji} = \eta \delta_j o_i$$



Example Backpropagation Neural Networks

Learning rate: η = 0.4



Total error:

$E_{total} = ½\sum(\text{target-output})^2$

$E1 = ½(0.2 - .6486)^2 = .1006$

$E_2 = ½(0.8 - .6480)^2 = .0116$

$E_{total} = .1006 + .0116 = 0.1122$

Forward Pass:

For h1:

Sum = .25x1 + .3x.2 + .3x.6 = 0.49

Output = $1/1+e^{-.49}$ = 0.6201

For h2:

Sum = .25x1 + .2x.4 + .3x.2 = 0.39

Output = $1/1+e^{-.39}$ = 0.5963

For t1:

Sum = .31x1 + .6201x.2 + .5963x.3 = .6129

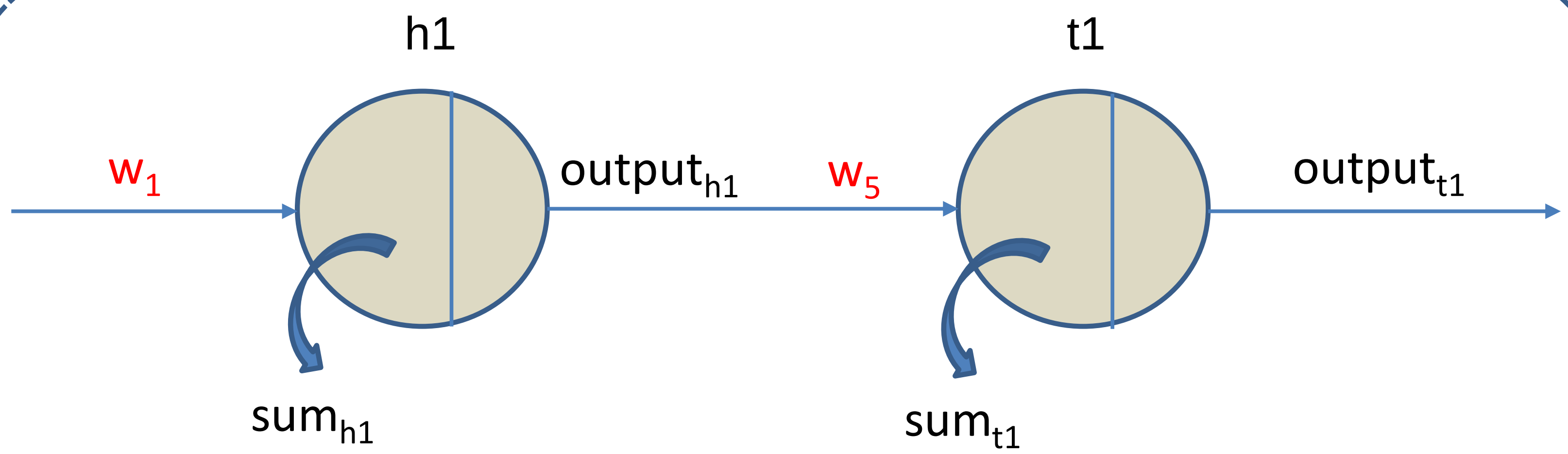
Output = $1/1+e^{-.6129}$ = 0.6486

For t2:

Sum = .31x1 + .6201x.1 + .5963x.4 = .6105

Output = $1/1+e^{-.6105}$ = 0.6480

Example Chain Rule



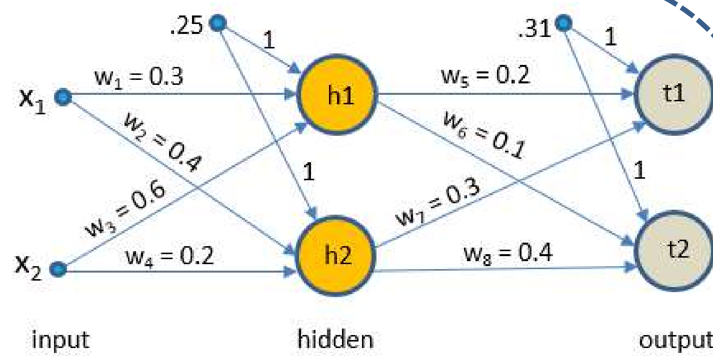
Chain Rule:

E is affected by → $output_{t1}$ → sum_{t1} → $output_{h1}$ → sum_{h1} → w_1

Backward Pass

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial output_{t1}} \times \frac{\partial output_{t1}}{\partial sum_{t1}} \times \frac{\partial sum_{t1}}{\partial w_5}$$

... Eq. (1)



$$E_{total} = \tfrac{1}{2} \sum (\text{actual} - \text{observed})^2$$

$$= \tfrac{1}{2}\left\{(\text{actual}_{t1} - \text{output}_{t1})^2 + \underline{(\text{actual}_{t2} - \text{output}_{t2})^2}\right\}$$

$$\frac{\partial output_{t1}}{\partial sum_{t1}} = ?$$

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

$$d/dx(\sigma(x)) = \sigma(x)\,(1-\sigma(x))$$

$$\frac{\partial E_{total}}{\partial output_{t1}} = 2 \times \tfrac{1}{2}\,(\text{actual}_{t1} - \text{output}_{t1}) \times -1 + 0$$

$$= (\text{output}_{t1} - \text{actual}_{t1})$$

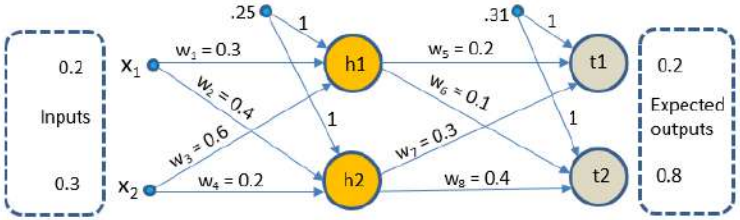
$$= 0.6486 - 0.2 = 0.4486$$ ... Eq. (2)

$$\frac{\partial output_{t1}}{\partial sum_{t1}} = \text{output}_{t1}(1-\text{output}_{t1})$$

$$= .6486(1-.6486) = .2279$$ ... Eq. (3)

$$\frac{\partial sum_{t1}}{\partial w_5} = \frac{\partial(\text{output}_{h1} x w_5 + \text{output}_{h2} x w_7)}{\partial w_5} = \text{output}_{h1} = 0.6201$$ ... Eq. (4)

Continued…



Eq. (1): $\frac{\partial E_{total}}{\partial w_5}$ = Eq.(1) x Eq.(2) x Eq.(3) = .4486 x .2279 x .6201 = .0634

$$w_5 = w_5 - \eta \frac{\partial E_{total}}{\partial w_5} = .2 - .4 \times .0634 = \boxed{.1746}$$

$$\frac{\partial E_{total}}{\partial w_6} = \frac{\partial E_{total}}{\partial output_{t2}} \text{ X } \frac{\partial output_{t2}}{\partial sum_{t2}} \text{ X } \frac{\partial sum_{t2}}{\partial w_6} = (.6480-.8)x(.6480x(1-.6480))x.6201$$

$$=-.152x.2281x.6201 = -.0215$$

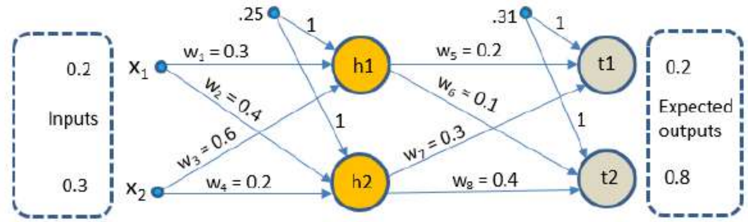
$$w_6 = w_6 - \eta \frac{\partial E_{total}}{\partial w_6} = .1-(.4x-.0215) = \boxed{.1086}$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial output_{t1}} \text{ X } \frac{\partial output_{t1}}{\partial sum_{t1}} \text{ X } \frac{\partial sum_{t1}}{\partial w_7} = \text{Eq.(2) x Eq.(3) x } output_{h2}$$

$$= 0.4486 \times 0.2279 \times 0.5963 = 0.0609$$

$$w_7 = w_7 - \eta \frac{\partial E_{total}}{\partial w_7} = .3 - .4 \times .0609 = 0.3 - 0.02436 = \boxed{0.2756}$$

Continued…



$$\frac{\partial E_{total}}{\partial w_8} = \frac{\partial E_{total}}{\partial output_{t2}} \text{ X } \frac{\partial output_{t2}}{\partial sum_{t2}} \text{ X } \frac{\partial sum_{t2}}{\partial w_8} = (-.152)\text{x } .2281\text{x}\overbrace{output_{h2}}^{.5963} = -0.0207$$

$$w_8 = w_8 - \eta \frac{\partial E_{total}}{\partial w_8} = 0.4 + 0.4 \text{ x } 0.0207 = 0.4 + 0.0083 = \boxed{.4083}$$

Similarly, w_2=.4007 … for you

Now Compute Weights in the Hidden Layer (w_1, w_2, w_3, and w_4): Chain becomes longer or shorter?

For w_1:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_1}{\partial w_1} + \frac{\partial E_2}{\partial w_1}$$

$$w_1 = w_1 + \eta\left(\partial E_{total} / \partial w_1\right)$$
$$= 0.3 - 0.4 \text{ x } 0.0008 = \boxed{.2997}$$

Where,

$$\frac{\partial E_1}{\partial w_1} = \frac{\partial E_1}{\partial output_{t1}} \text{ x } \frac{\partial output_{t1}}{\partial sum_{t1}} \text{ x } \frac{\partial sum_{t1}}{\partial output_{h1}} \text{ x } \frac{\partial output_{h1}}{\partial sum_{h1}} \text{ x } \frac{\partial sum_{h1}}{\partial w_1}$$

$$\frac{\partial E_1}{\partial w_1} = .4486 \text{ x } .2279 \text{ x } w5 \text{ x } (output_{h1} \text{ x } (1 - output_{h1})) \text{ x } 0.2 = 0.00096$$

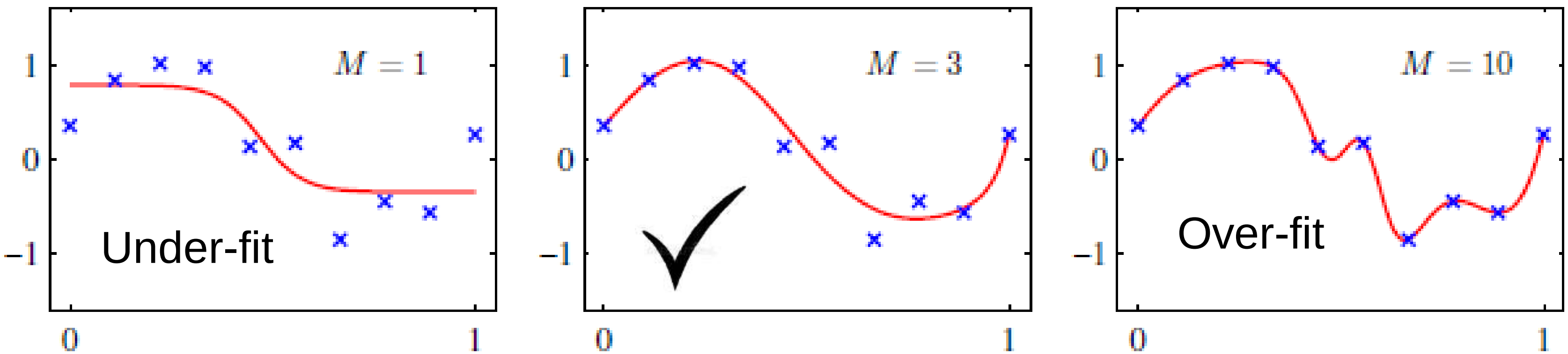
Now,

$$\frac{\partial E_2}{\partial w_1} = \frac{\partial E_2}{\partial output_{t2}} \text{ x } \frac{\partial output_{t2}}{\partial sum_{t2}} \text{ x } \frac{\partial sum_{t2}}{\partial output_{h1}} \text{ x } \frac{\partial output_{h1}}{\partial sum_{h1}} \text{ x } \frac{\partial sum_{h1}}{\partial w_1}$$

$= -.1520 \text{ x } .2281 \text{ x } w6 \text{ x } .2356 \text{ x } .2 = -.00016$ $\partial E_{total}/\partial w_1 = .00096 - .00016 = .0008$

Regularization in Neural Networks

- Which one is a free parameter in a Neural network?
  - Input Output or Number of units in the hidden layer (M)
- Why Regularization is needed in Neural Networks?
  - To improve the generalization/ learning outcome. To control impact of noise and fluctuations on the dataset. Alternatively, to avoid over-fitting.



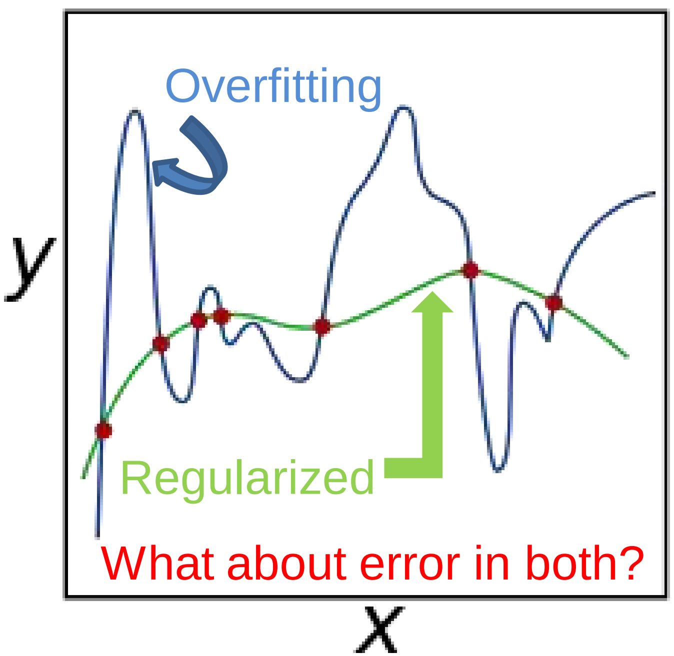
(Fitting a Sinusoidal dataset with different number of hidden units and Sum-of-Squares error function optimized by Gradient descent)

Img. Source: Bishop Text

Regularization: Weight Decay (L1/L2)

- Control model complexity by the addition of a regularization term to the error function.

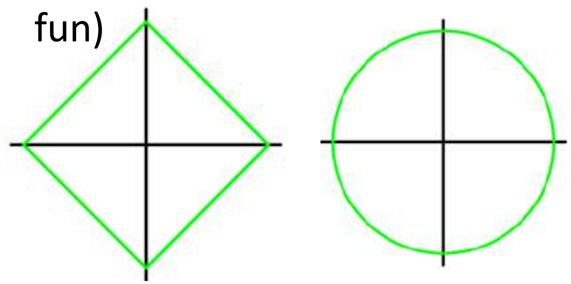
Loss function: $\underbrace{\boxed{E_D(\mathbf{w})}}_{\text{Data term}} + \underbrace{\boxed{\lambda E_W(\mathbf{w})}}_{\text{Regularization term}}$



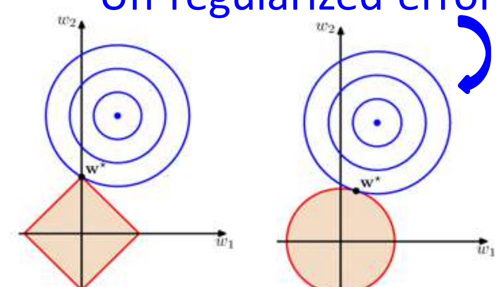
With Sum-of-squares error and a quadratic regularizer: $$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n)\}^2 + \boxed{\frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}}$$

L2 regularization (Ridge)

Contour plots (behaviour of fun)



Un-regularized error

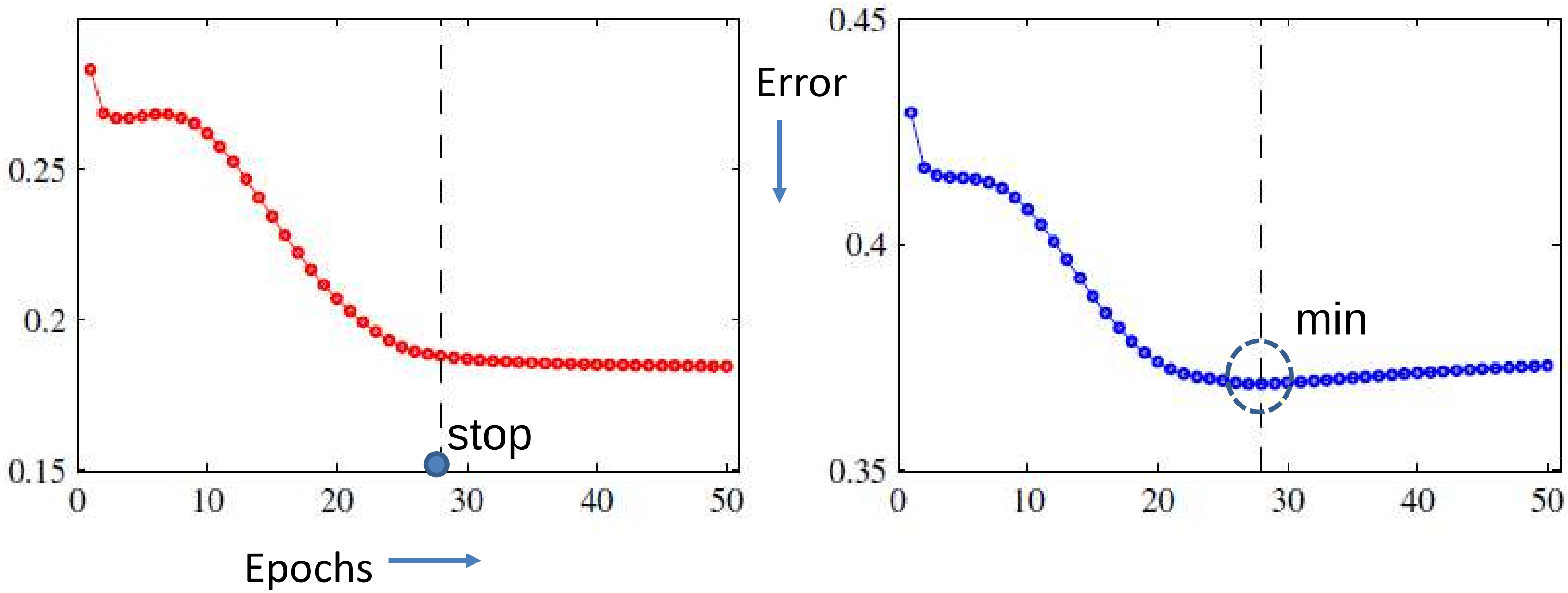


In L1 regularization (Lasso), the additional term added to the loss function is the sum of the absolute values of the weights. This encourages sparsity in the weights, effectively shrinking some of them to zero. $J_{L1}(\theta) = J(\theta) + \lambda \sum_{i=1}^{n} |\theta_i|$

Where, λ as the regularization parameter, ϑ as the vector of weights of the Network.

Regularization: Early Stopping

- Early stopping monitors the performance of the model on a validation set and stops training when the performance starts to degrade, thus preventing the model from overfitting to the training data.
- Example sinusoidal dataset



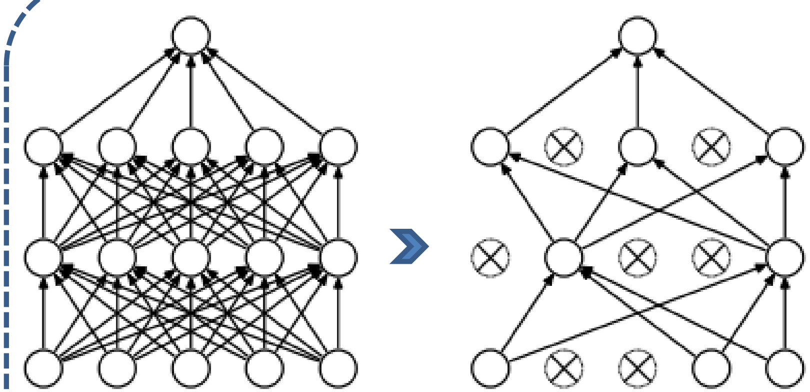
(Training set error)

(Validation set error)

Img. Source: Bishop text

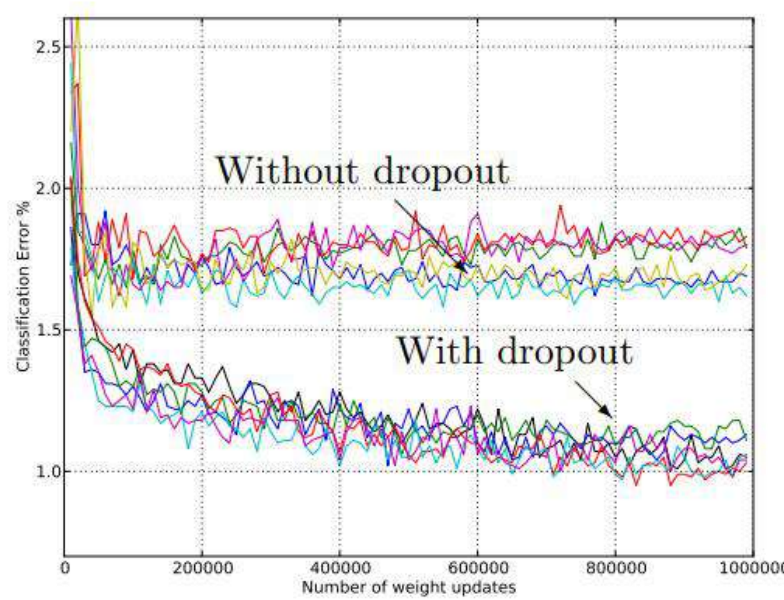
Stochastic Regularization: Dropout

- Drop out each individual unit with some probability ρ (usually ρ = 1/2) by setting its activation to ‘0’.
- The key idea behind dropout is to prevent overfitting by adding noise to the network during training.



(Standard Neural Network) (After applying dropout)

(Img. Source: Nitish Srivastava, et al., Journal of Machine Learning Research, 15, 2014)



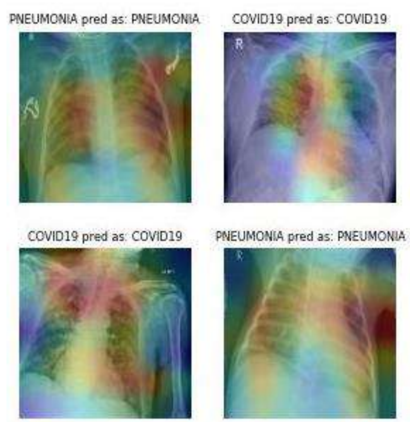
During inference (testing or prediction), dropout is typically turned off, and the full network is used. However, the weights are usually scaled by ‘1-ρ’ during inference to account for the fact that more units were active during training.

Regularization: Data Augmentation

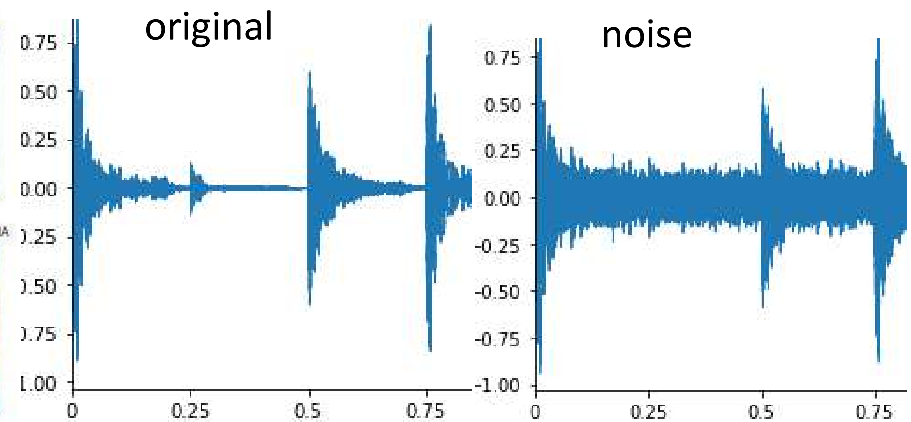
- Data augmentation acts as a form of regularization by introducing additional variations and diversity into the training dataset.
- A technique used to artificially increase the size of a training dataset by applying various transformations to the existing data samples.
- Random rotation, Random scaling, Random cropping, Horizontal or vertical flipping, Adding noise (e.g., Gaussian noise), Changing brightness, contrast, or saturation

Applications

(Self driving cars)



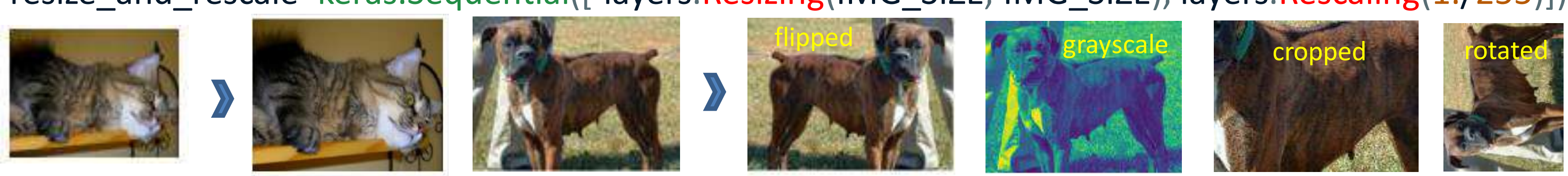
(Healthcare)



(Automatic Speech Recognition)

Img. Source: https://www.datacamp.com/

```
resize_and_rescale=keras.Sequential([ layers.Resizing(IMG_SIZE, IMG_SIZE), layers.Rescaling(1./255)])
```



PyTorch Ex: BPNs for Predicting Age of Abalones

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim
```

	Length	Diameter	Height	Whole_weight	Shucked_weight	Viscera_weight	Shell_weight	Rings
0	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Label (no of rings will decide the age)

```python
class BPN(nn.Module):
    def __init__(self):
        super(BPN, self).__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```python
train(model, criterion, optimizer, X_train_tensor, y_train_tensor)

def evaluate(model, X_test, y_test):
    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
        mse = nn.MSELoss()
        loss = mse(outputs, y_test)
        print(f"Test Loss: {loss.item()}")

evaluate(model, X_test_tensor, y_test_tensor)
```

```
Epoch 20, Loss: 1110.1275
Epoch 30, Loss: 1110.1275
Epoch 40, Loss: 1110.1275
Epoch 50, Loss: 1110.1275
Epoch 60, Loss: 1110.1275
Epoch 70, Loss: 1110.1275
Epoch 80, Loss: 1110.1275
Epoch 90, Loss: 1110.1275
Epoch 100, Loss: 1110.127
Test Loss: 13.29509258270
```

Thank You!