02.04.2024

BITS F464: Machine Learning

NEURAL NETWORKS: PERCEPTRON, BACK PROPAGATION

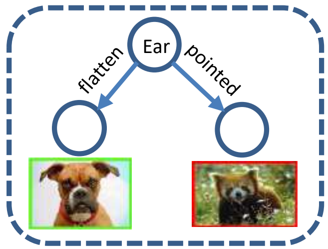
Chittaranjan Hota, Sr. Professor
Dept. of Computer Sc. and Information Systems
hota@hyderabad.bits-pilani.ac.in

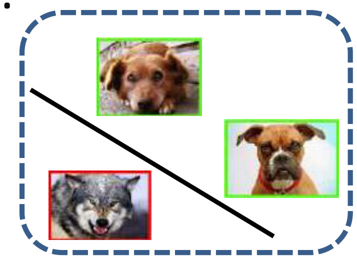
Recap:

(A) Symbolic learning:

It involves representing knowledge in a symbolic form, often using logical rules (logical structures) or mathematical functions.

$$\text{Gini Index} = 1 - \sum_{i=1}^{n} (P_i)^2$$



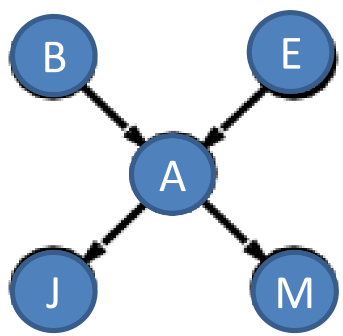


$$y(x) = w^T x + w_0 = \sum_{i=1}^{d} w_i x_i + w_0$$

(B) Probabilistic learning:

Takes a Joint probability P(x, y) where x is the input and y is the label and predicts the most possible known label $\tilde{y} \in Y$ for the unknown variable $\tilde{x}$ using the Bayes theorem.

$$P(h|D) = \frac{P(D|h).P(h)}{P(D)}$$



Naïve Bayes (MAP):

$$= \underset{h \in H}{\operatorname{argmax}} P(h) \prod_{i=1}^{n} P(x|h)$$

(C) Connectionist learning (Artificial Neural Networks) today...

ANNs: Motivating Examples

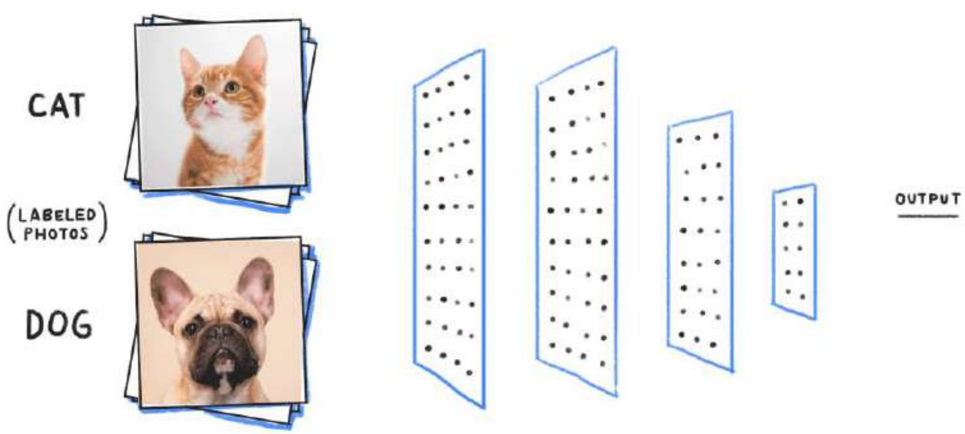
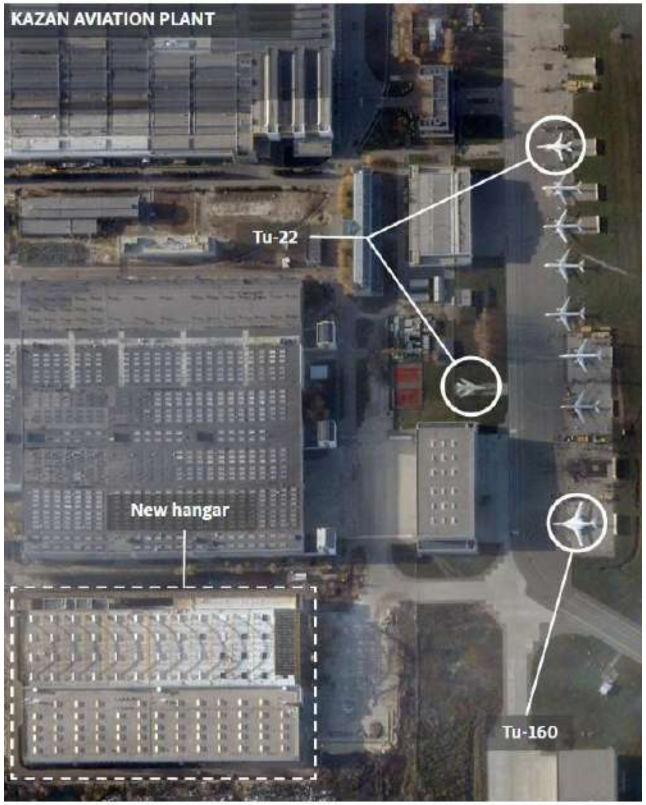


Image source: https://towardsdatascience.com/



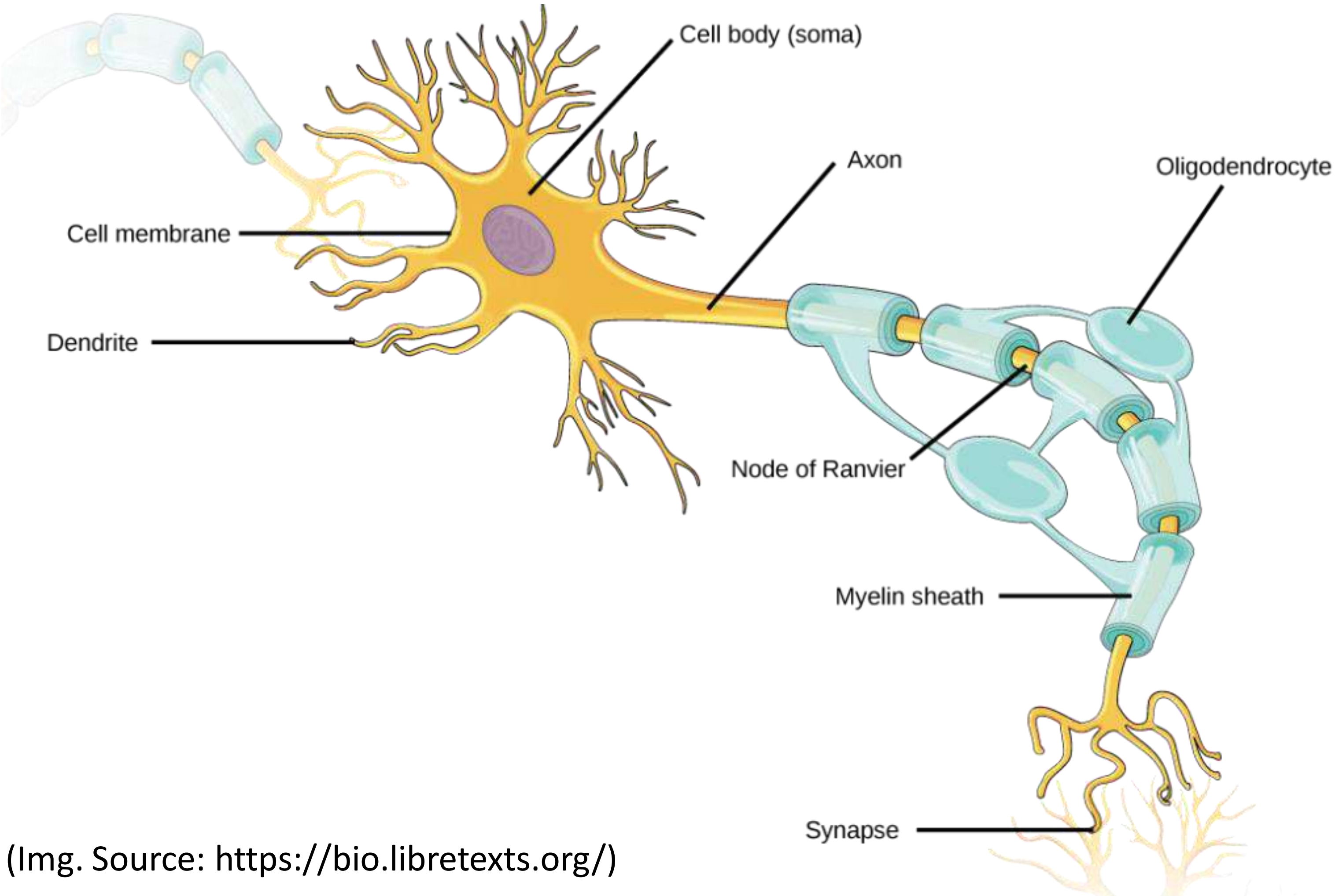
(Russia-Ukrain War: Ukrainians used ANNs to combine ground-level photos, drone video footage and satellite imagery to enhance War Intelligence)

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

A healthy human brain has around 100 billion neurons (10^{11}), and a neuron may connect to as many as 100,000 other neurons.

A Nerve Cell: Neuron

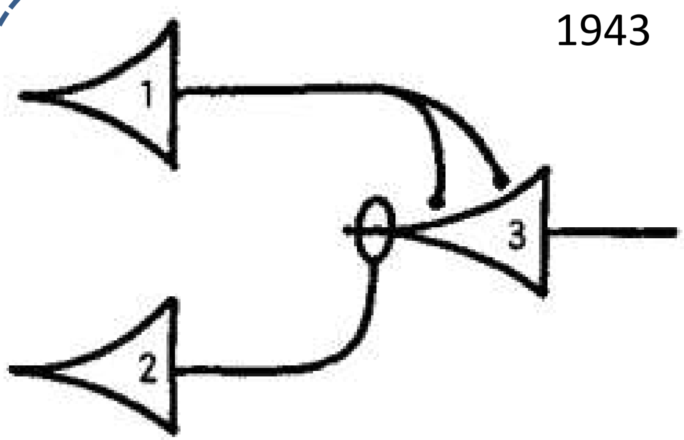


(Img. Source: https://bio.libretexts.org/)

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

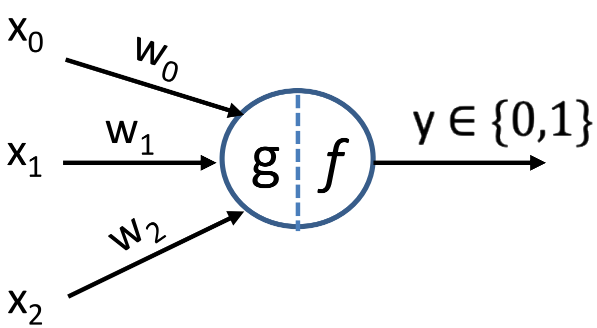
FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE, DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE, AND THE UNIVERSITY OF CHICAGO

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

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

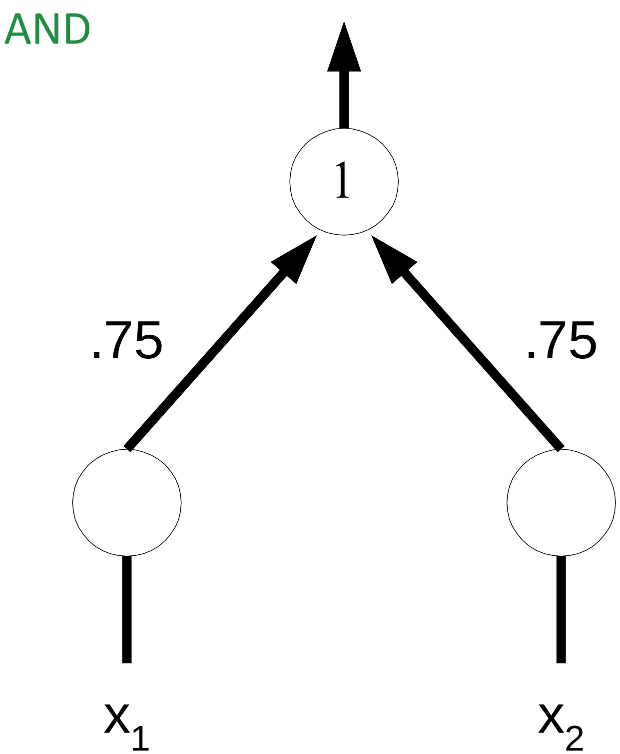
Perceptron: An Example

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



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

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

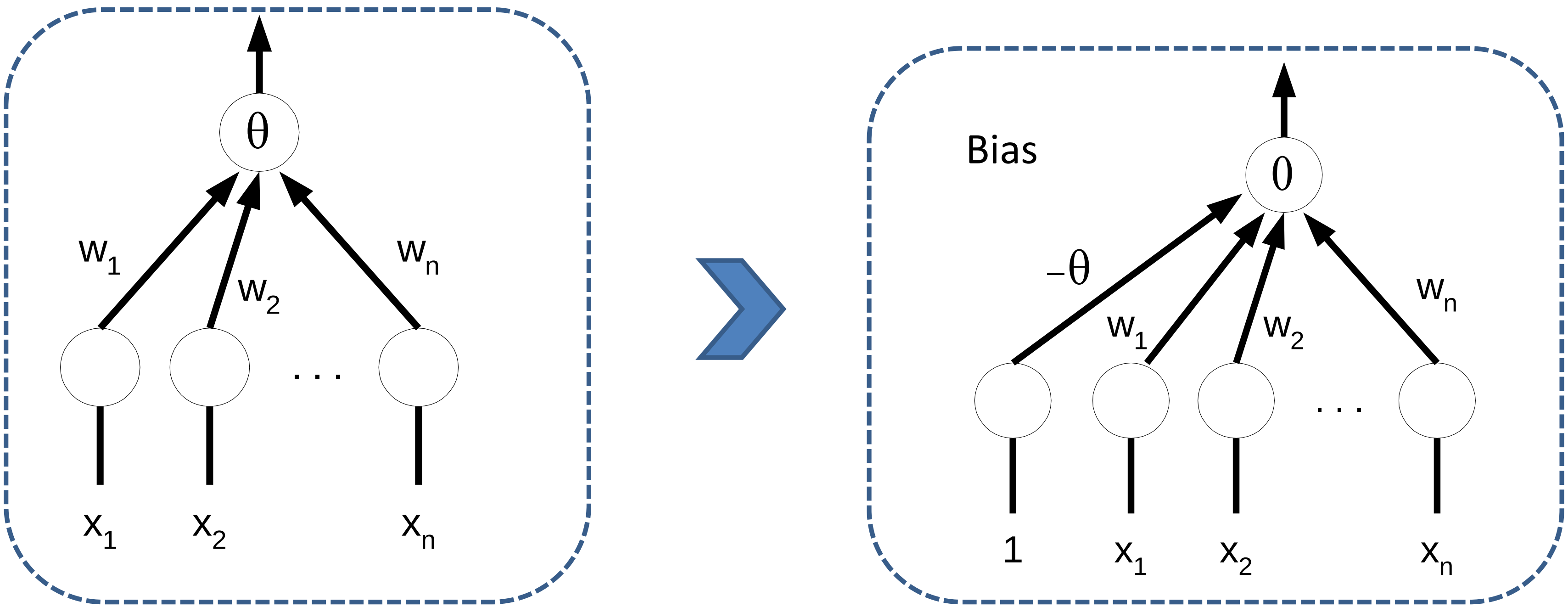


What function this neuron computes?

Normalizing thresholds

- Why do we need Normalization?

$$y(x) = w_0 + w_1 x_1 + w_2 x_2, \cdots, w_n x_n = w_0 + \sum_{i=1}^{n} w_i x_i$$

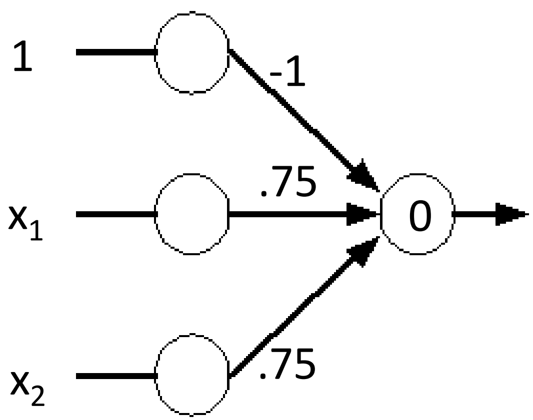


Advantage: threshold = 0 for all neurons:

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

Normalized examples

AND



INPUT: $x_1 = 1, x_2 = 1$

$1*-1 + .75*1 + .75*1 = .5 >= 0$ ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$

$1*-1 +.75*1 + .75*0 = -.25 < 1$ ➔ OUTPUT: 0

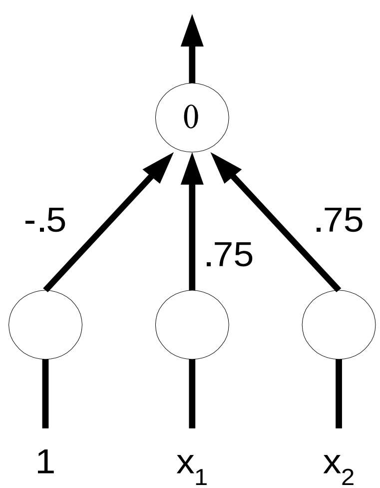
INPUT: $x_1 = 0, x_2 = 1$

$1*-1 +.75*0 + .75*1 = -.25 < 1$ ➔ OUTPUT: 0

INPUT: $x_1 = 0, x_2 = 0$

$1*-1 +.75*0 + .75*0 = -1 < 1$ ➔ OUTPUT: 0

OR



INPUT: $x_1 = 1, x_2 = 1$

$1*-.5 + .75*1 + .75*1 = 1 >= 0$ ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$

$1*-.5 +.75*1 + .75*0 = .25 > 1$ ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 1$

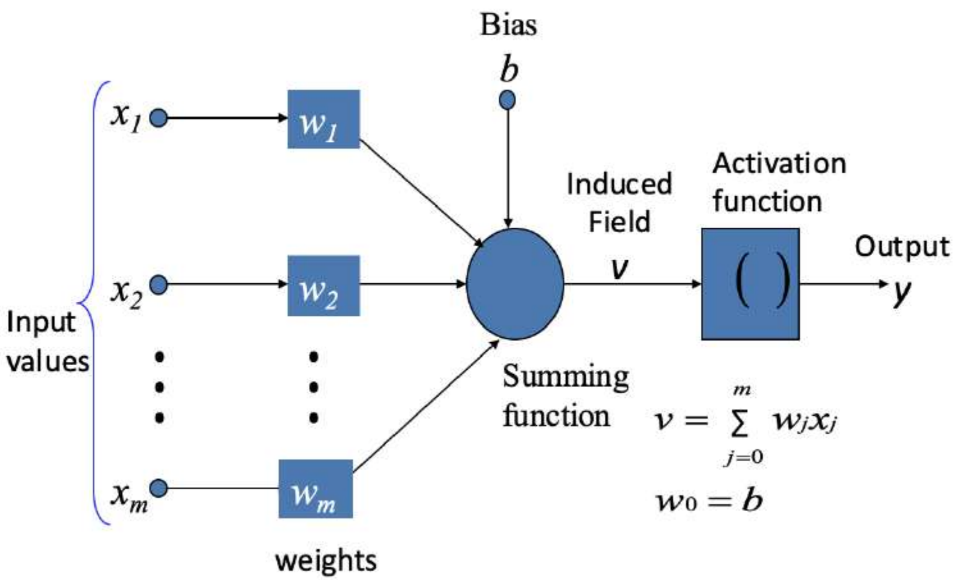
$1*-.5 +.75*0 + .75*1 = .25 < 1$ ➔ OUTPUT: 1

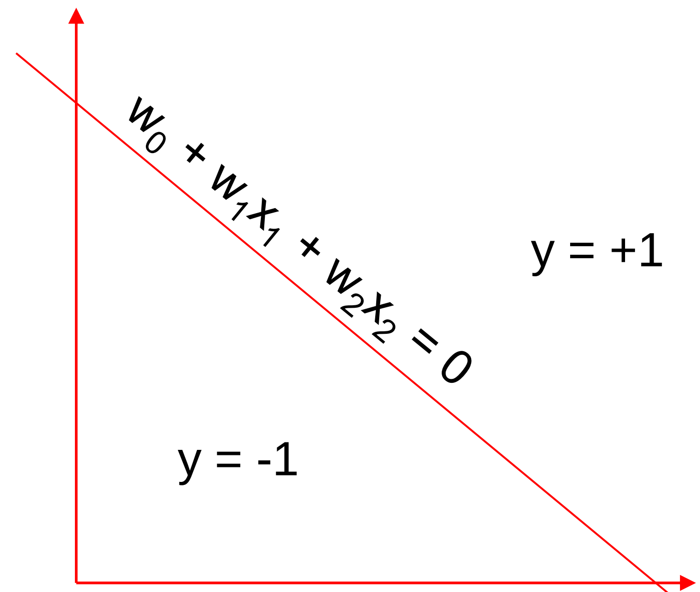
INPUT: $x_1 = 0, x_2 = 0$

$1*-.5 +.75*0 + .75*0 = -.5 < 1$ ➔ OUTPUT: 0

Perceptron as a Decision Surface

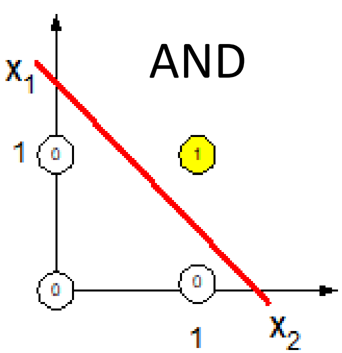
- Perceptron is a Linear Binary Classifier

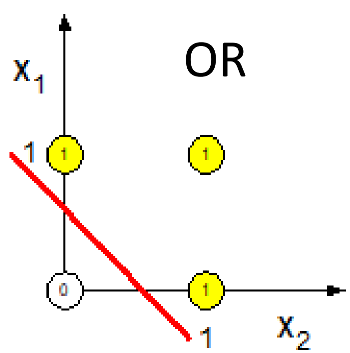




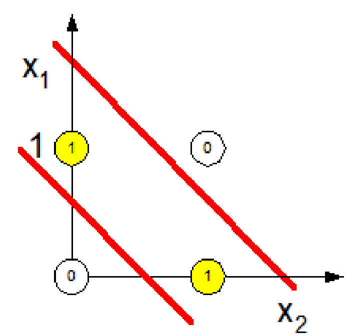
(A 2-dimensional space)

- A perceptron can only solve linearly separable classification problems.

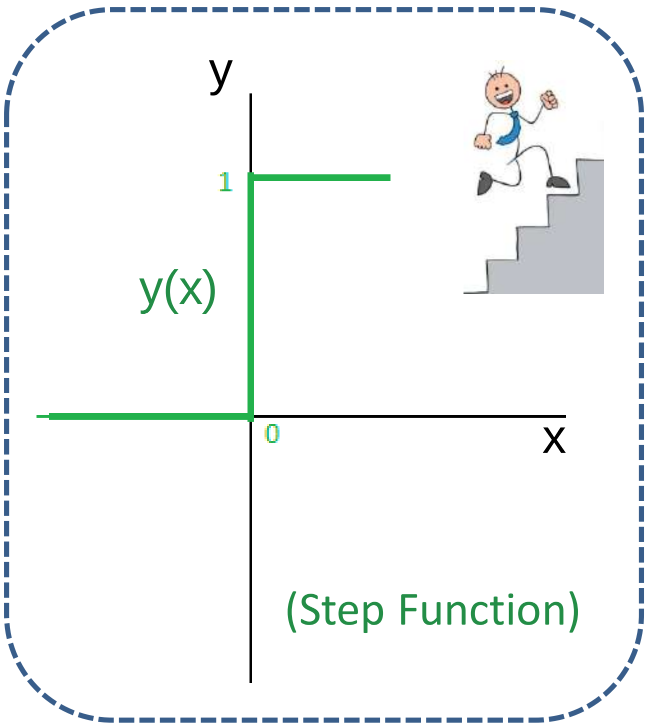




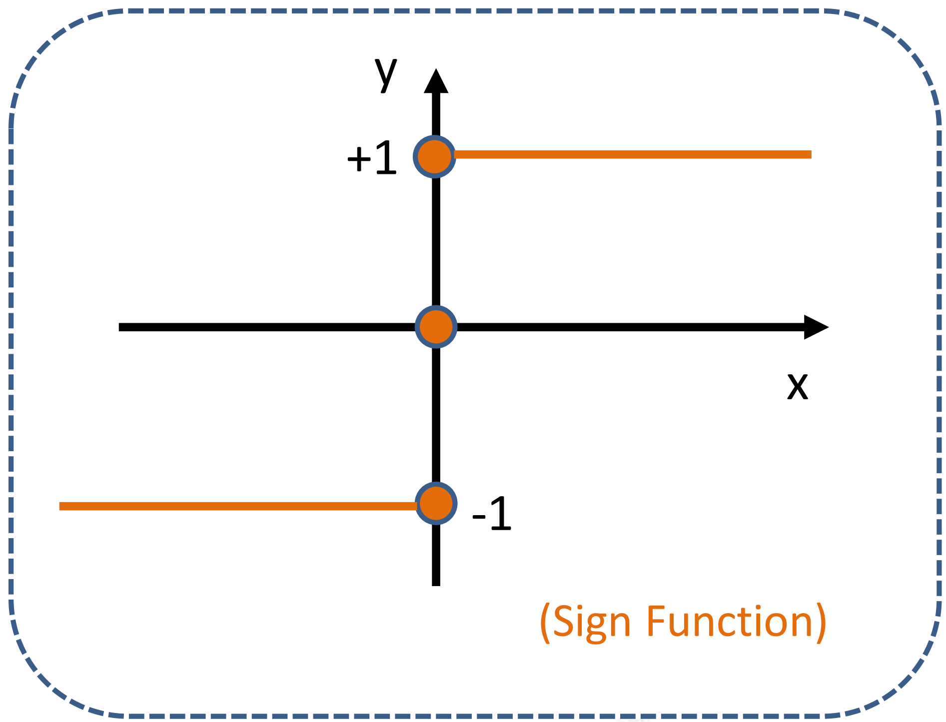
How about XOR?



Activation Functions in a Perceptron



(Step Function)



(Sign Function)

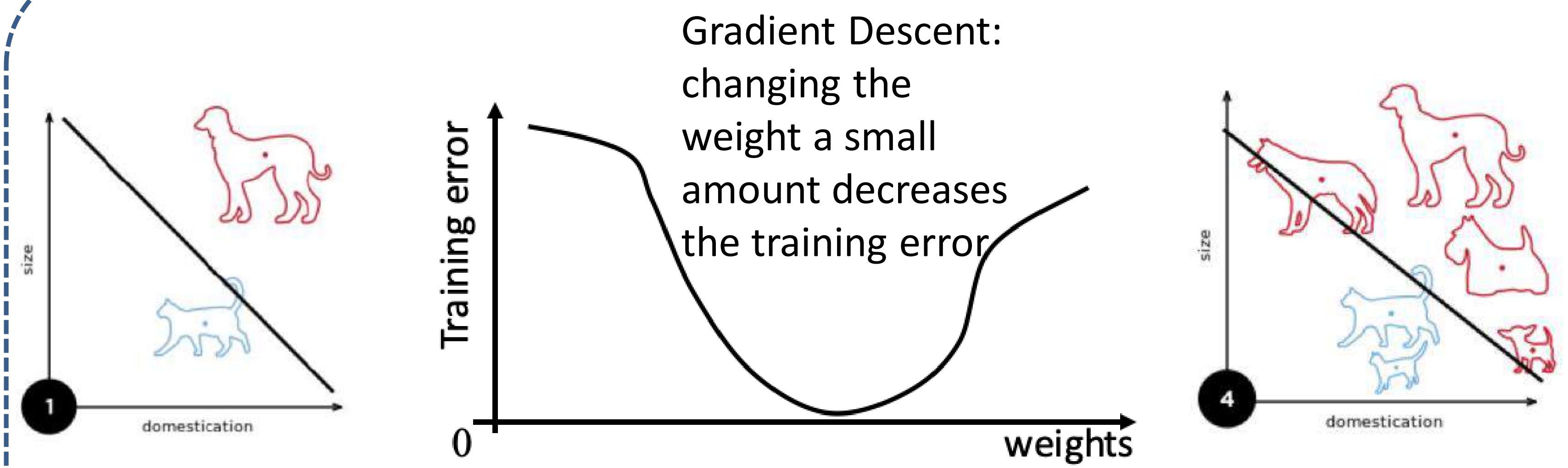
y (x) = 1, if $-\boldsymbol{\theta}X1 + \sum_{i=1}^{n} w_i\, x_i \geq 0$

= 0, 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, Gaussian etc.? 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

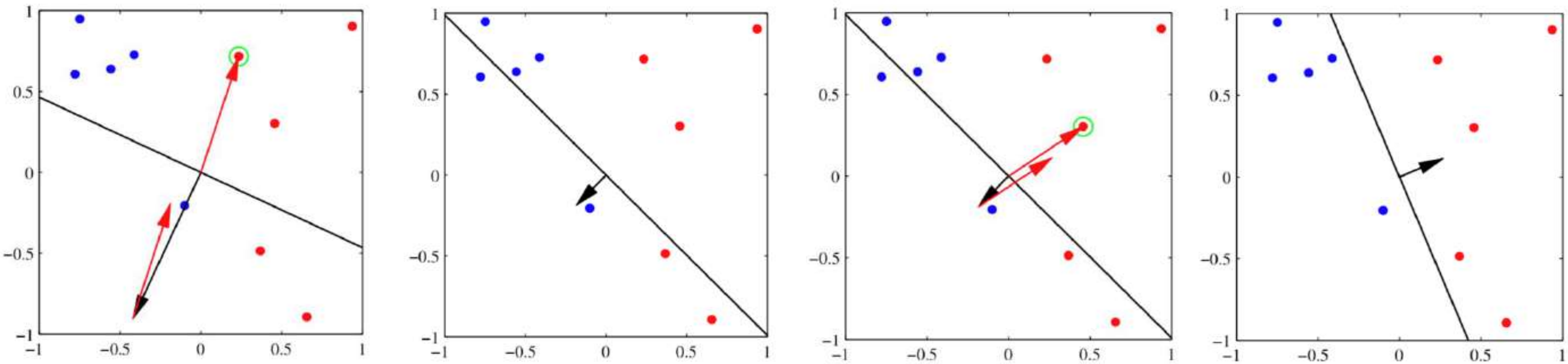


Image Source: PRML, Bishop

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.
```

Run through the algorithm: AND

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

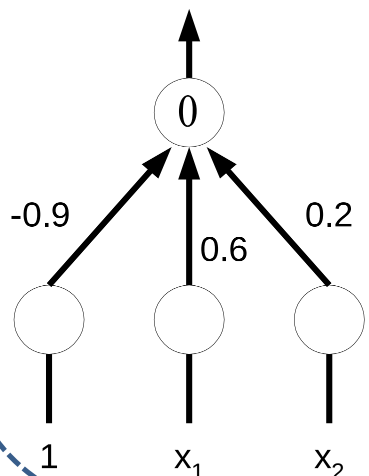
(Training Set)

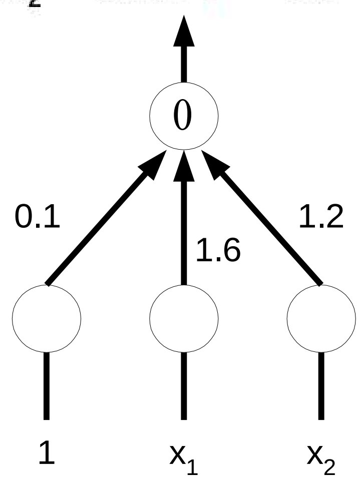
$x_1 = 1, x_2 = 1$:	$-0.9*1 + 0.6*1 + 0.2*1$	$= -0.1 \rightarrow 0$	WRONG
$x_1 = 1, x_2 = 0$:	$-0.9*1 + 0.6*1 + 0.2*0$	$= -0.3 \rightarrow 0$	OK
$x_1 = 0, x_2 = 1$:	$-0.9*1 + 0.6*0 + 0.2*1$	$= -0.7 \rightarrow 0$	OK
$x_1 = 0, x_2 = 0$:	$-0.9*1 + 0.6*0 + 0.2*0$	$= -0.9 \rightarrow 0$	OK

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

New Weights

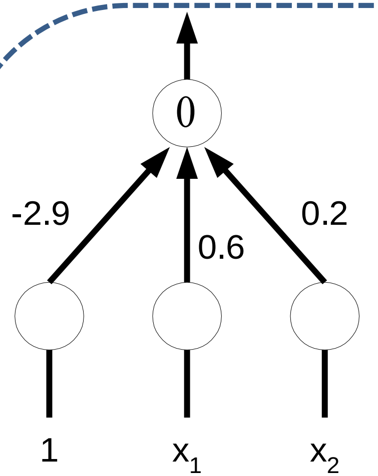
$w_0 = 0.1 - 1 - 1 - 1 = -2.9$
$w_1 = 1.6 - 1 - 0 - 0 = 0.6$
$w_2 = 1.2 - 0 - 1 - 0 = 0.2$





$x_1 = 1, x_2 = 1$:	$0.1*1 + 1.6*1 + 1.2*1$	$= 2.9 \rightarrow 1$	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

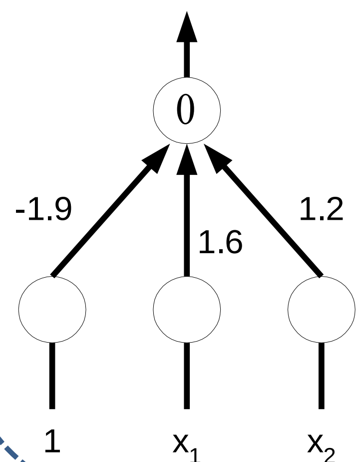
Continued…



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

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

$\longrightarrow$ New Weights



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

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

Perceptron Training Rule: Recap

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

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 ↑

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

Where,

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

Why should this update rule converge toward successful weight values?

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

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.

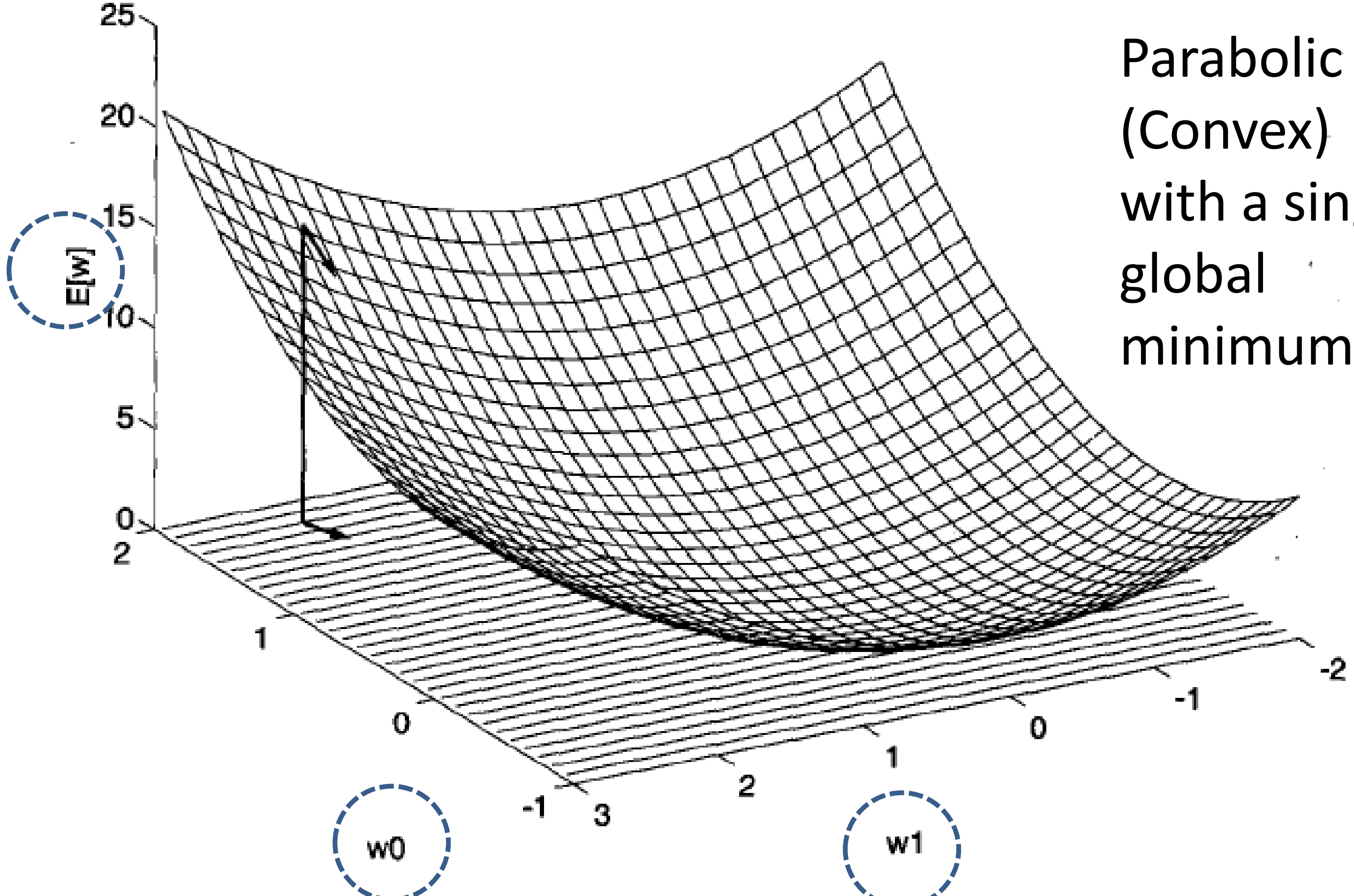
- Delta rule is best understood by considering an un-thresholded Perceptron, i.e. a linear unit without threshold (or activation function).

- 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

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:

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

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

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 {

       $$w_i \leftarrow w_i + \Delta w_i$$

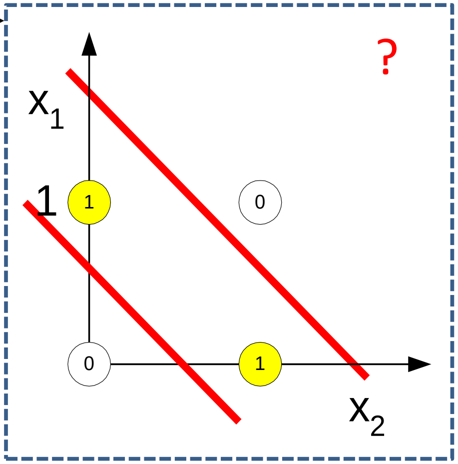
    4. }

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

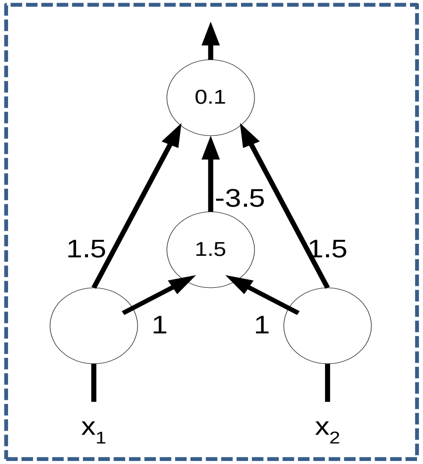
GD: the error is summed over all examples before updating weights. It might miss global minima when multiple local minima are present. In SGD/Incremental GD, weights are updated upon examining each training example.

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.

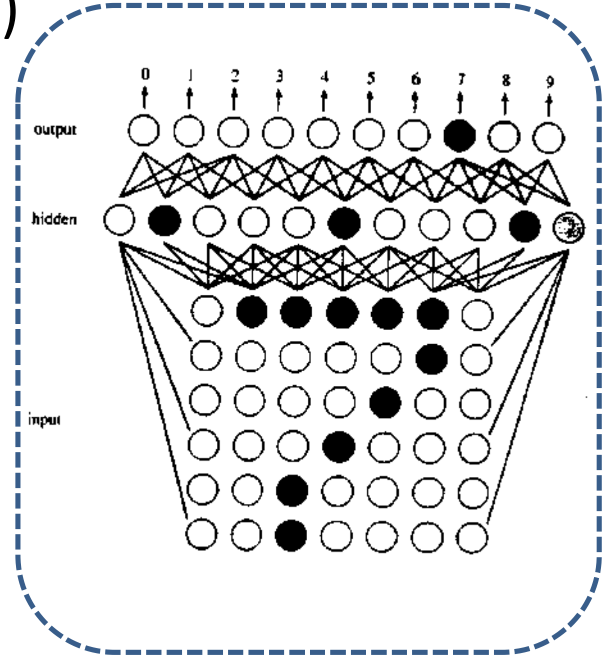
How to build such a multi-layer network?

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

Hidden units in a Multi-layer Perceptron (MLP)

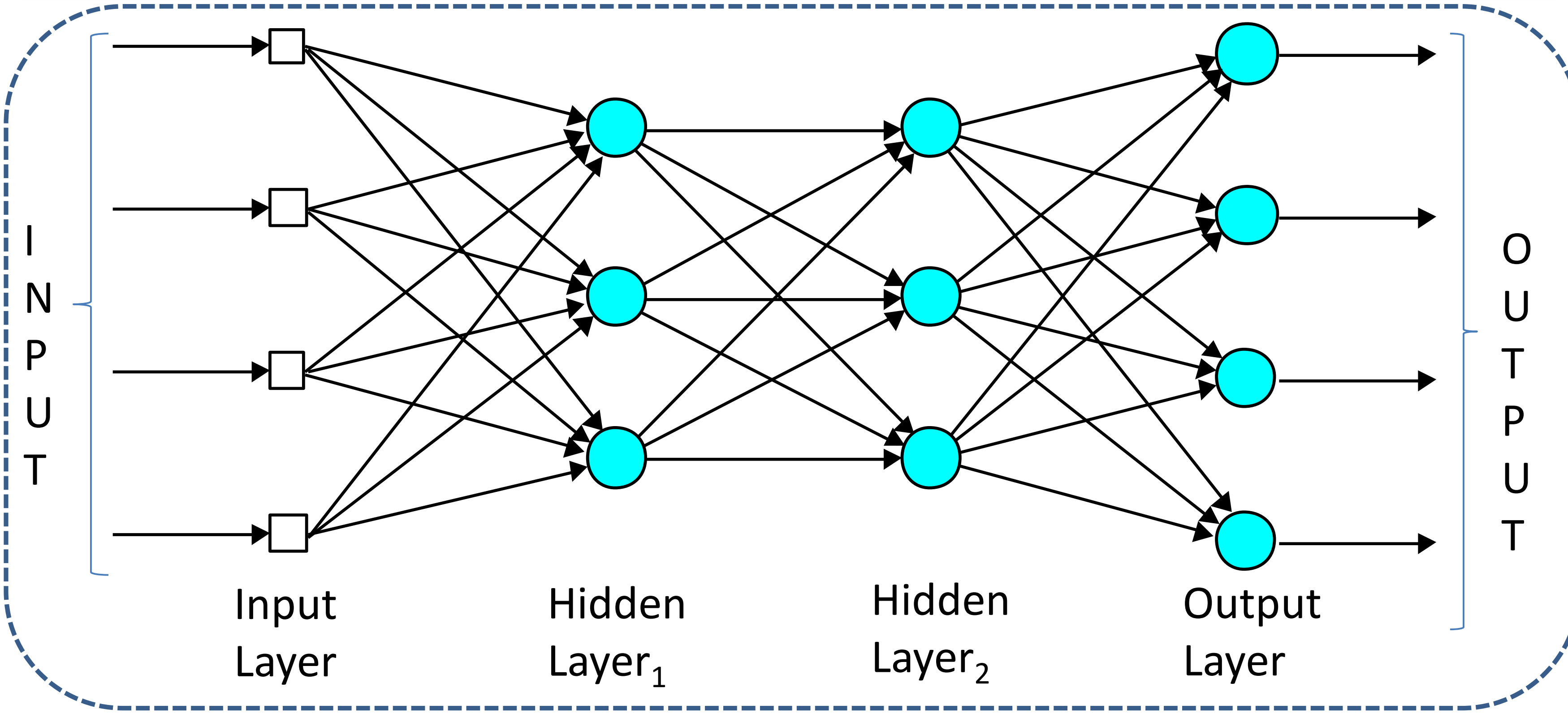
- The addition of hidden units allows the network to develop complex feature detectors (i.e., internal representations)

- e.g., Optical Character Recognition (OCR)
    - perhaps one hidden unit "looks for" a horizontal bar
    - another hidden unit "looks for" a diagonal
    - another looks for the vertical base



- The combination of specific hidden units indicates a 7.

Multiple Hidden Layers

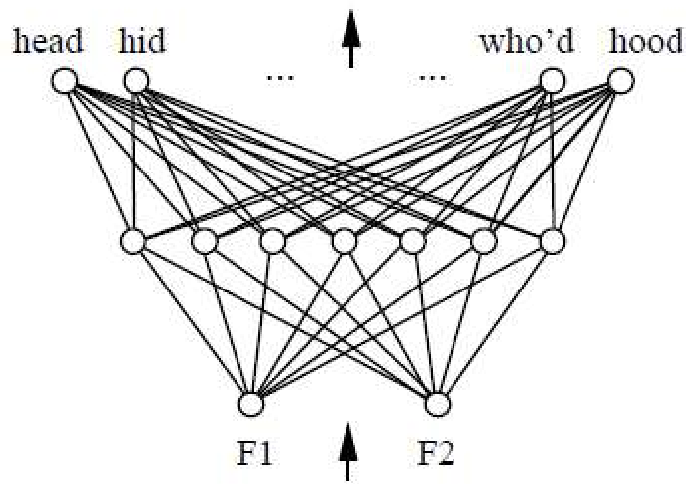


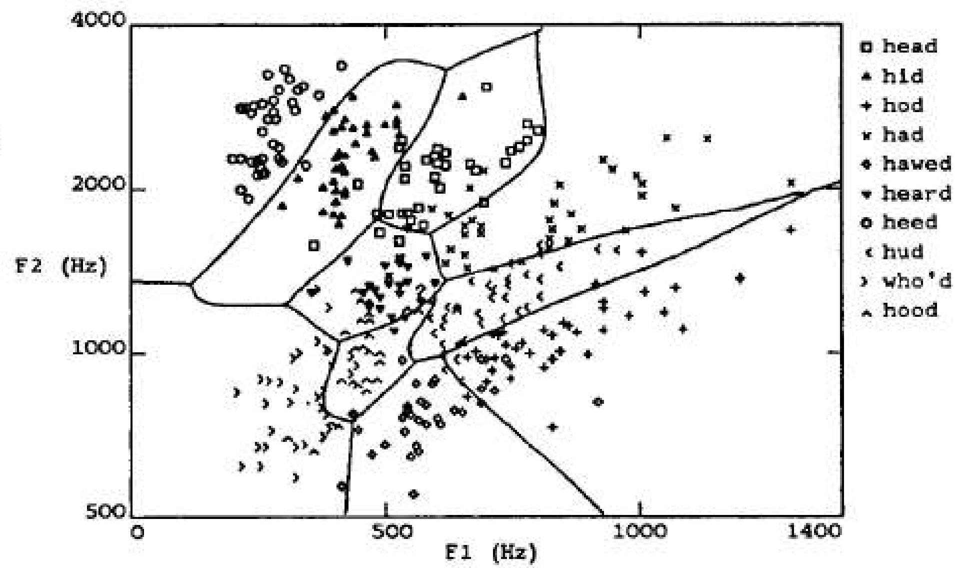
What does a hidden layer hide?

Hides it's desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network.

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

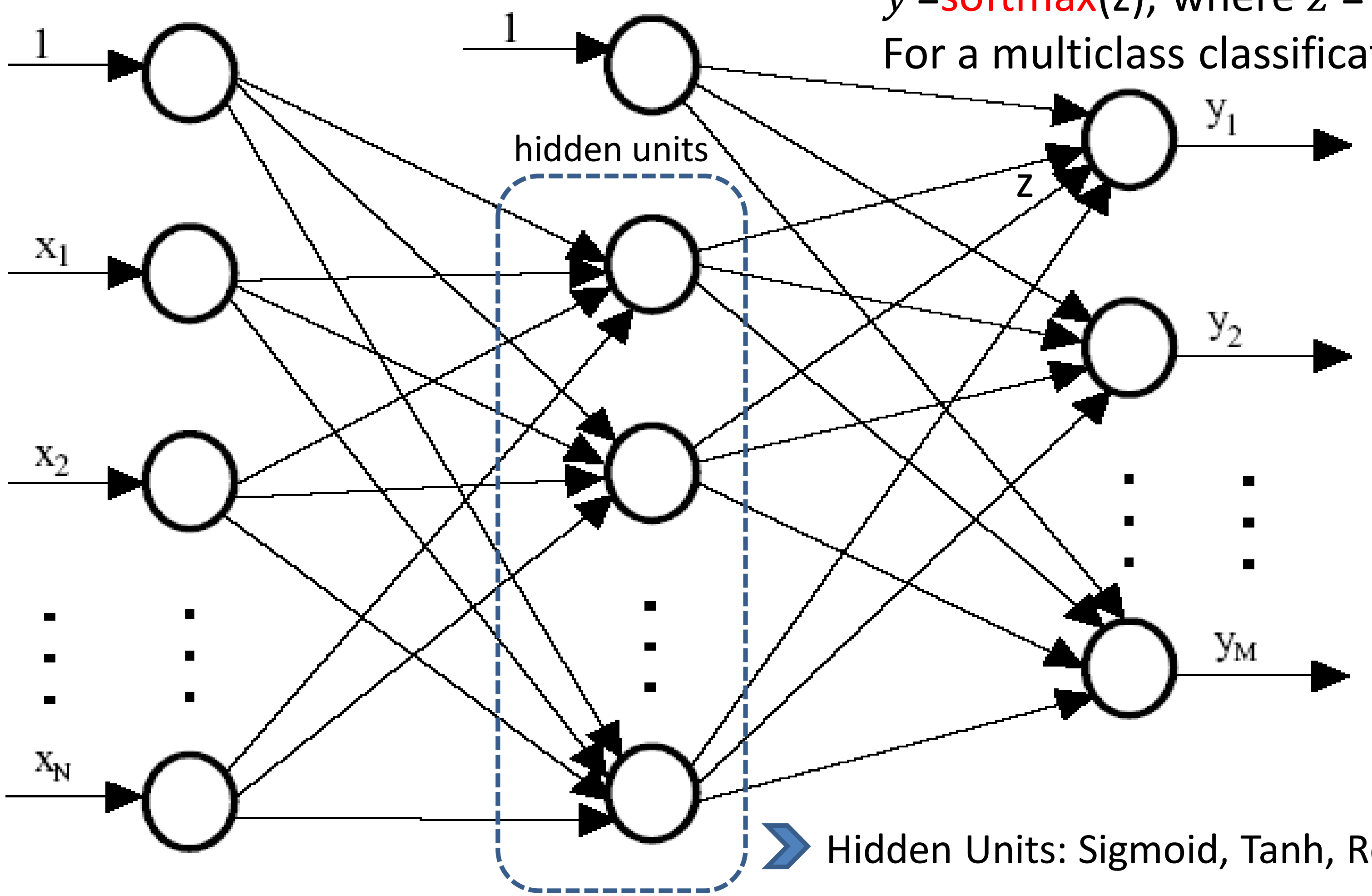
Output: vowel sound occurring in the context "h__d"

Image source: Tom Mitchell's Text

An Example 3-layer Perceptron

y =softmax(z), where $z = W^{\mathsf{T}}h + b$

For a multiclass classification.



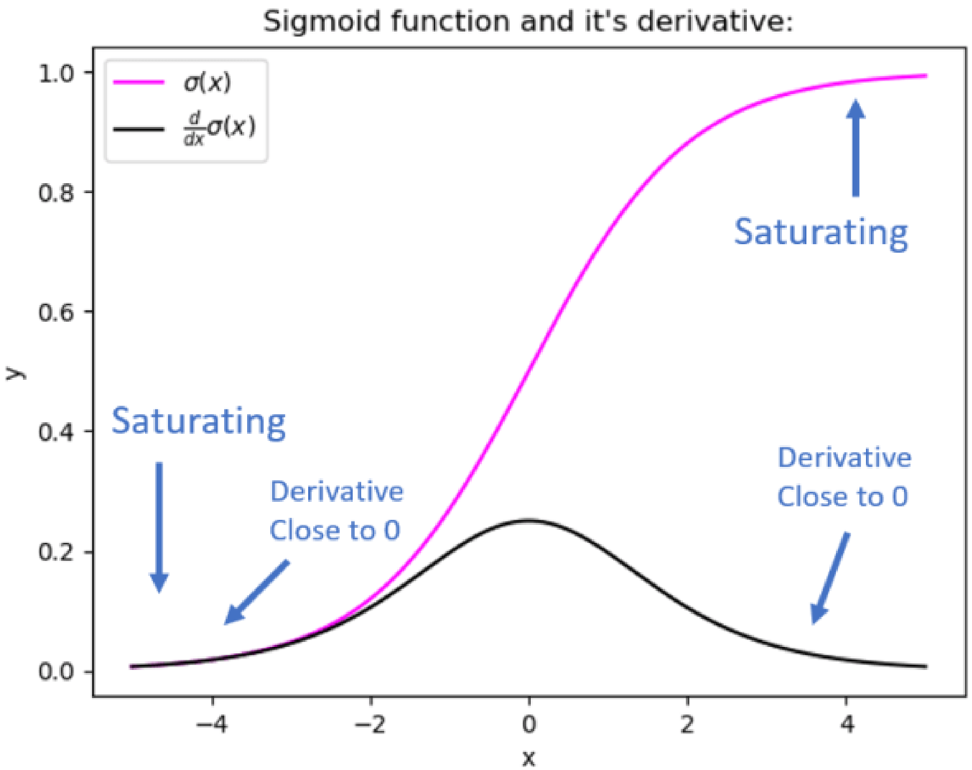
$y = \sigma(w^T h + b)$

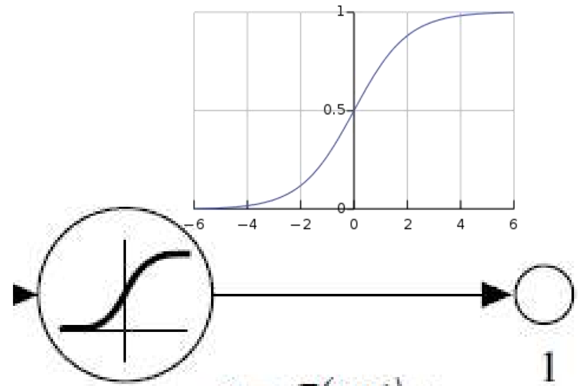
b
i
n
a
r
y

- Hidden Units: Sigmoid, Tanh, ReLU etc…

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

Activation Function: Sigmoid





$$o = \sigma(net) = \frac{1}{1 + e^{-net}}$$

During backpropagation, the gradients of the weights in the early layers of the network (closer to the input) can become very small as they are multiplied by small gradients of later layers using the sigmoid function.

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

Vanishing Gradient

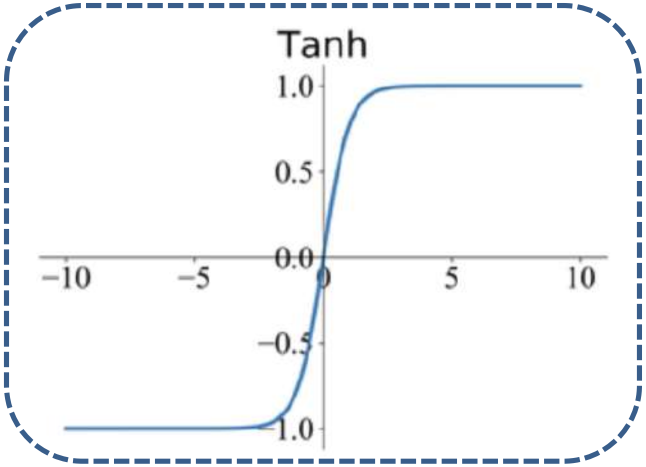
Where should you worry much? Shallow or Deep NNs?

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.

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



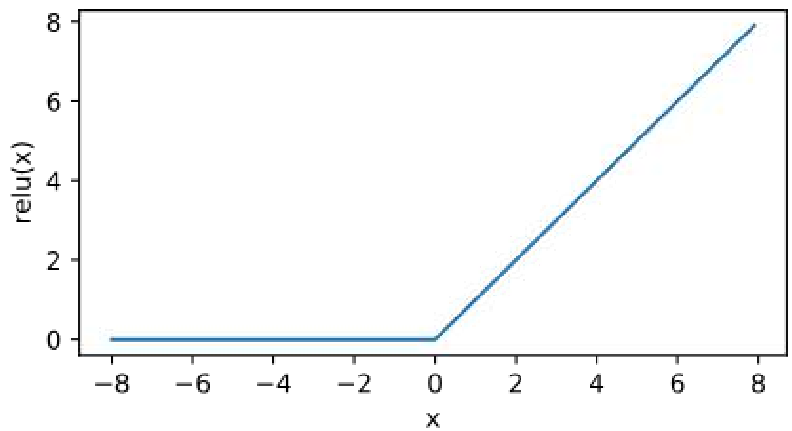
Used in RNNs, and LSTMs…

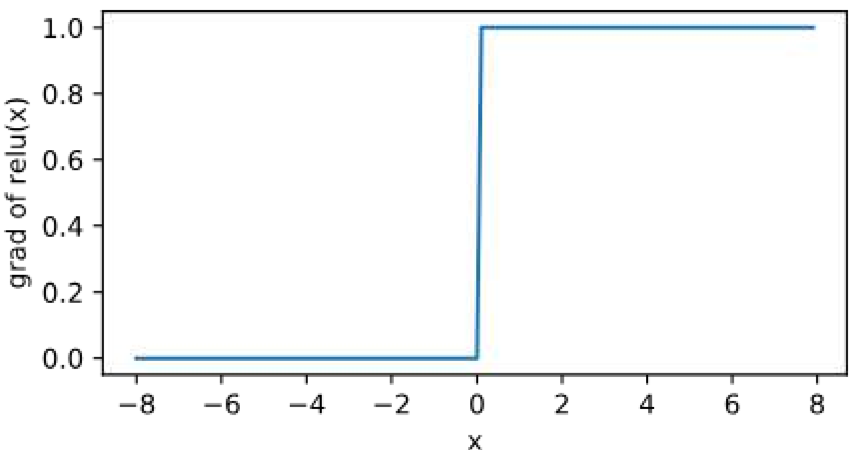
Suffers from same Vanishing gradient problem.

Activation Functions: ReLU

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

ReLU(x)=max(x,0) → Retains only positive elements and discards negative ones.





The reason for using the ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it reduces the issue of the vanishing gradient problem.

Unlike sigmoid or tanh which saturate in certain regions (i.e., the gradients become very close to zero), ReLU does not saturate in the positive region → for positive inputs, the derivative of ReLU is always 1. Hence, during backpropagation, gradients do not vanish for positive values, allowing for faster and more effective learning.

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:

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

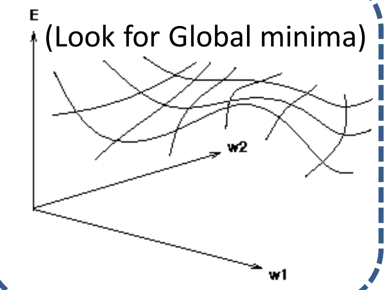
$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

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

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

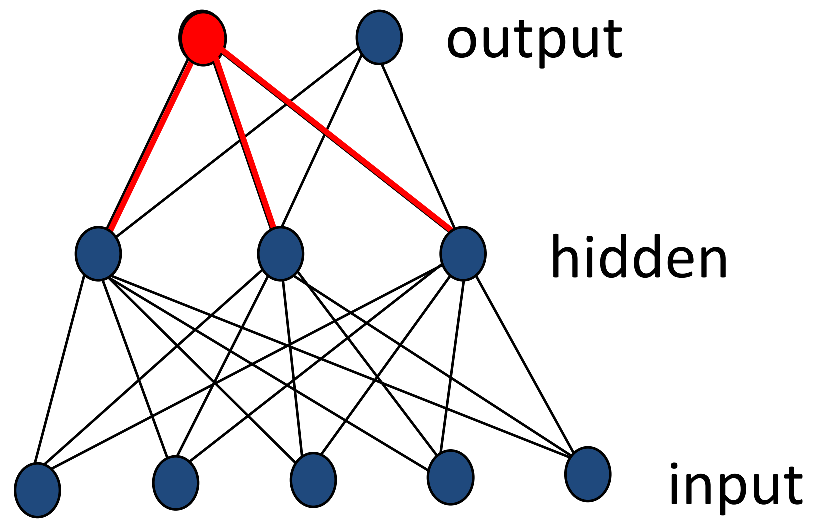
Correct output: t_j=1.0

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

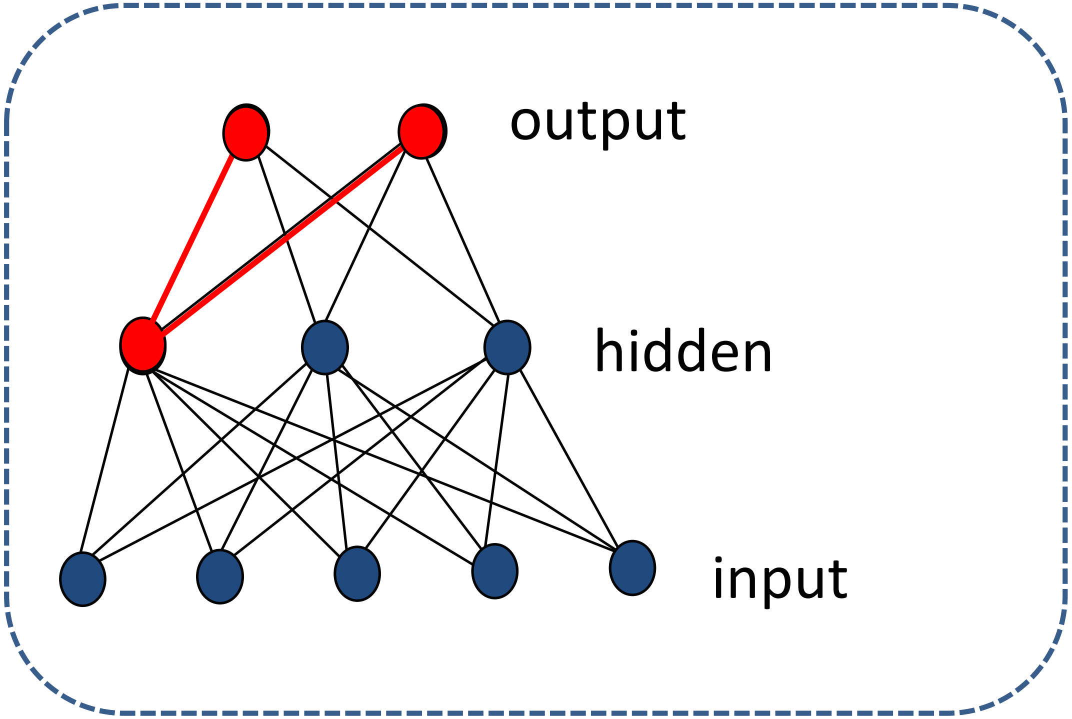
$$\Delta w_{ji} = \eta \delta_j o_i$$



Error Backpropagation continued…

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

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



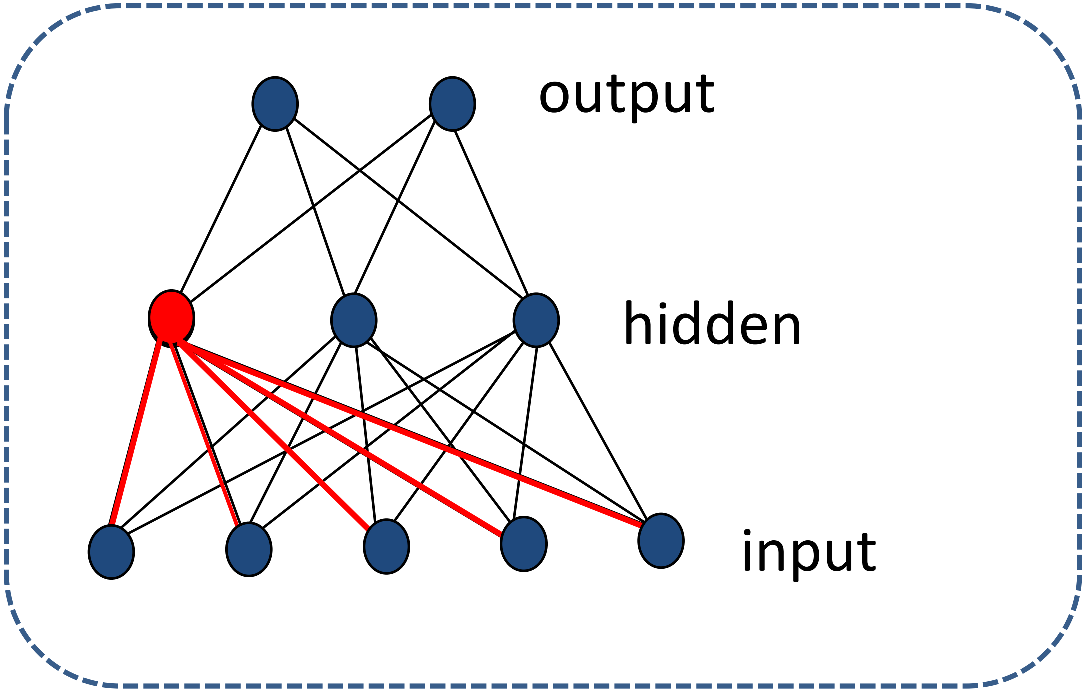
Error Backpropagation continued…

- 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*

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



Assignment 5: 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!