POLITECNICO
MILANO 1863

POLITECNICO
MILANO 1863

Artificial Neural Networks and Deep Learning

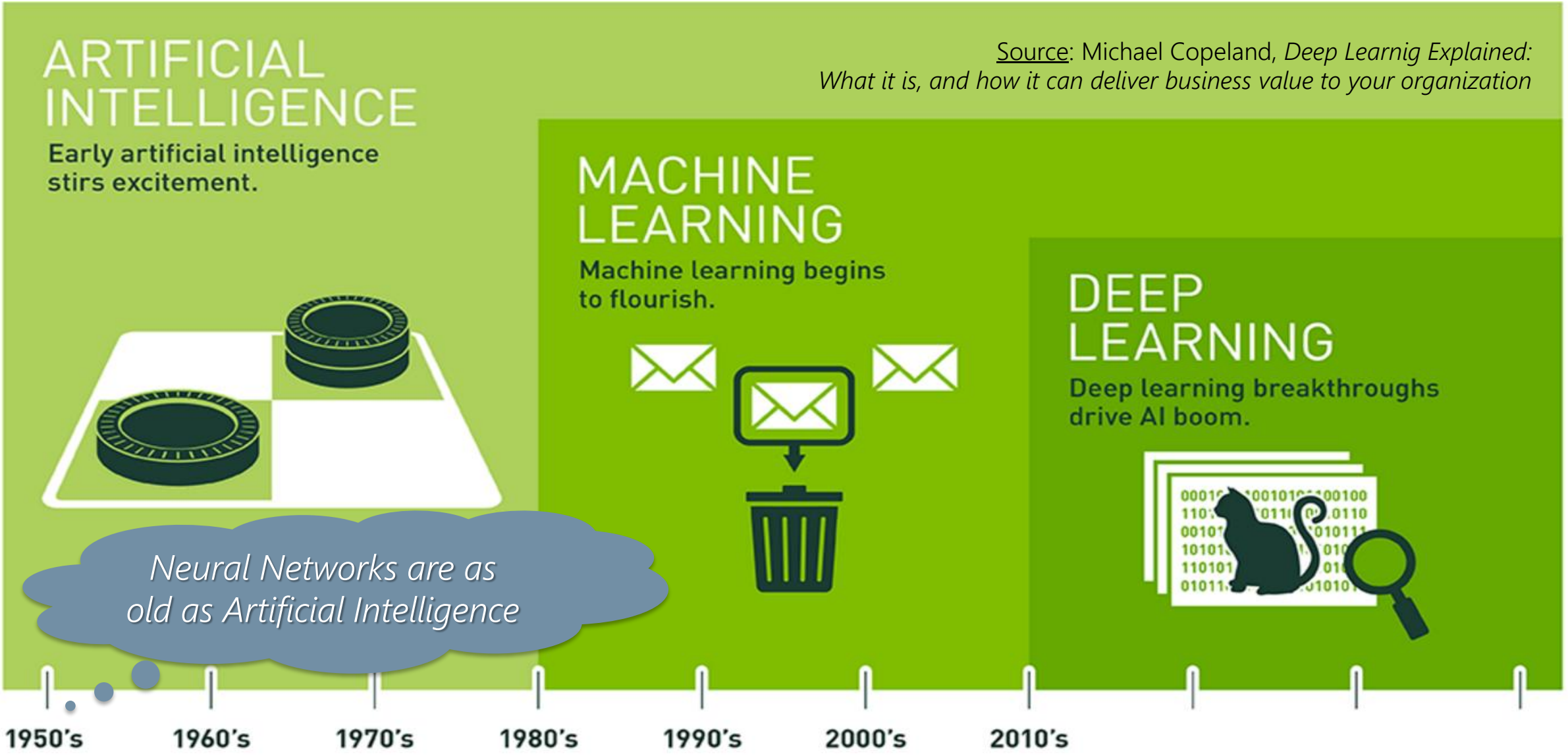
- From Perceptrons to Feed Forward Neural Networks -

Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB
ARTIFICIAL INTELLIGENCE AND ROBOTICS LAB

«Deep Learning is not AI, nor Machine Learning»

Source: Michael Copeland, *Deep Learnig Explained: What it is, and how it can deliver business value to your organization*



The inception of AI

A PROPOSAL FOR THE

DARTMOUTH SUMMER RESEARCH PROJECT

ON ARTIFICIAL INTELLIGENCE

J. McCarthy, Dartmouth College
M. L. Minsky, Harvard University
N. Rochester, I. B. M. Corp
C. E. Shannon, Bell Telepho

August 31, 1955

1) Automatic Computers

If a machine can do a job, then an automatic calculator can be programmed to simulate the machine. The speeds and memory capacities of present computers may be insufficient to simulate many of the higher functions of the human brain, but the major obstacle is not lack of machine capacity, but our inability to write programs taking full advantage of what we have.

3. Neuron Nets

How can a set of (hypothetical) neurons be arranged so as to form concepts. Considerable theoretical and experimental work has been done on this problem by Uttley, Rashevsky and his group, Farley and Clark, Pitts and McCulloch, Minsky, Rochester and Holland, and others. Partial results have been obtained but the problem needs more theoretical work.

5) Self-Improvement

Probably a truly intelligent machine will carry out activities which may best be described as self-improvement. Some schemes for doing this have been proposed and are worth further study. It seems likely that this question can be studied abstractly as well.

A Proposal for the

DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLI

We propose that a 2 month, 10 man study of artificial intelligence carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.

The following are some aspects of the artificial intelligence problem:

6) Abstractions

A number of types of "abstraction" can be distinctly defined and several others less distinctly. A direct attempt to classify these and to describe machine methods of forming abstractions from sensory and other data would seem worthwhile.

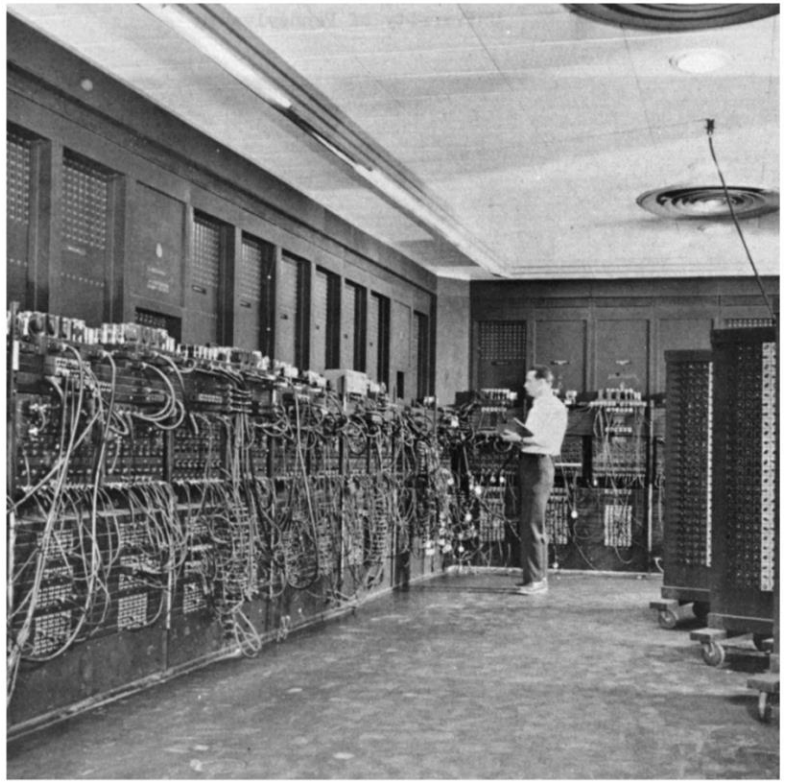
Let's go back to 1940s …

Computers were already good at

- Doing precisely what the programmer programs them to do
- Doing arithmetic very fast

However, they would have liked them to:

- Interact with noisy data or directly with the environment
- Be massively parallel and fault tolerant
- Adapt to circumstances



Researchers were seeking a computational model beyond the *Von Neumann Machine*!

The Brain Computationa Model

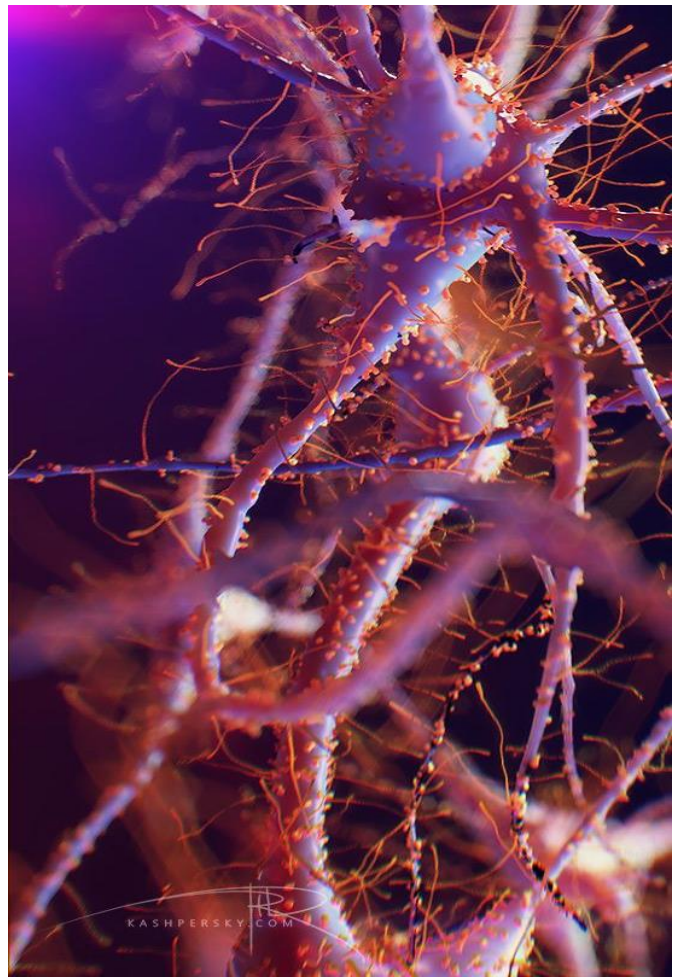
The human brain has a huge number of computing units:

- 10^{11} (one hundred billion) neurons
- 7,000 synaptic connections to other neurons
- In total from 10^{14} to 5 x 10^{14} (100 to 500 trillion) in adults to 10^{15} synapses (1 quadrillion) in a three year old child

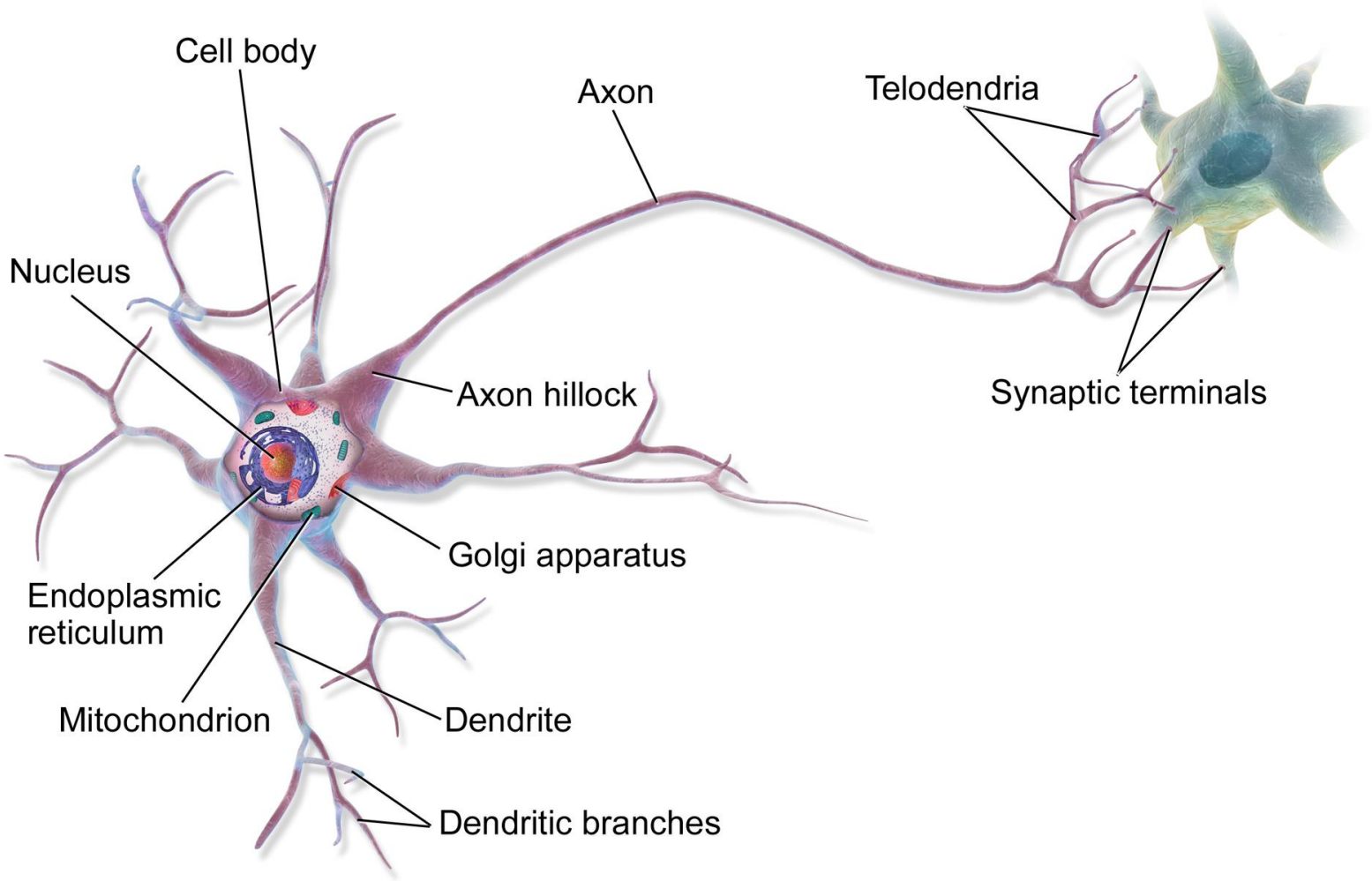
The computational model of the brain is:

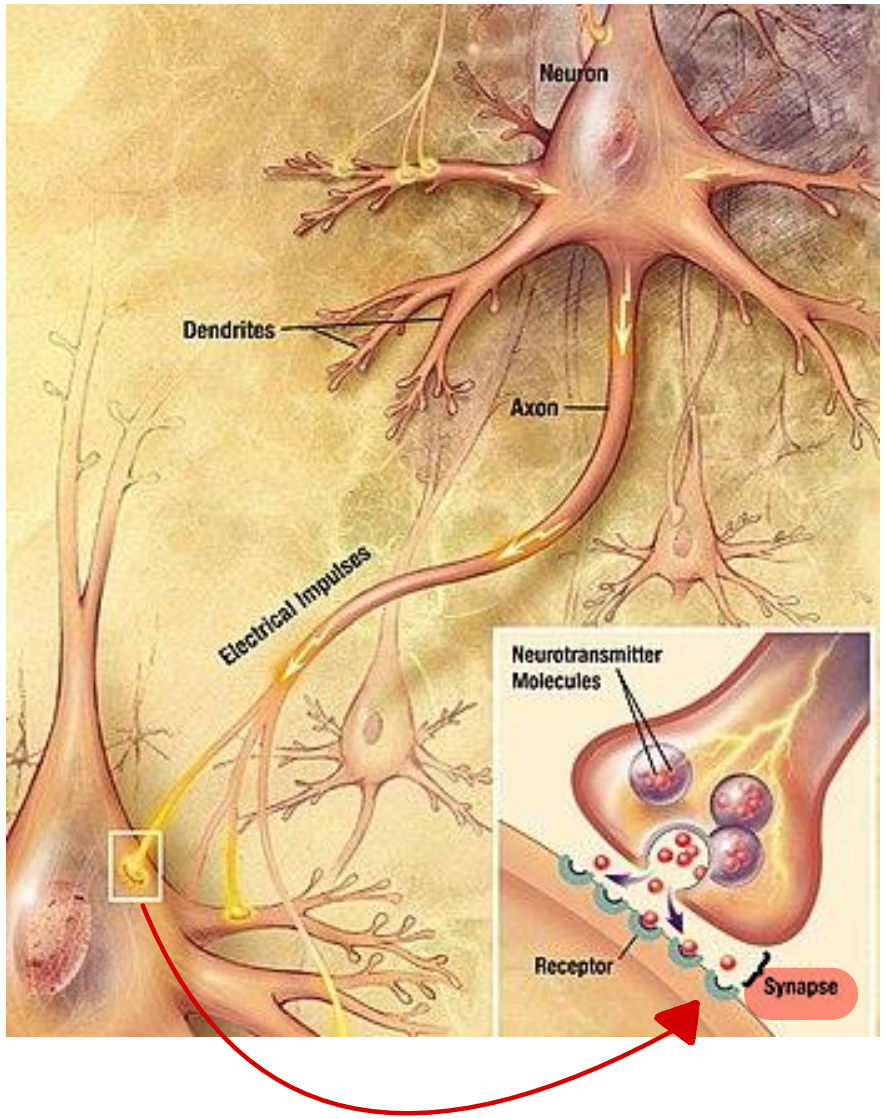
- Distributed among simple non linear units
- Redundant and thus fault tolerant
- Intrinsically parallel

Perceptron: a computational model based on the brain!



Computation in Biological Neurons

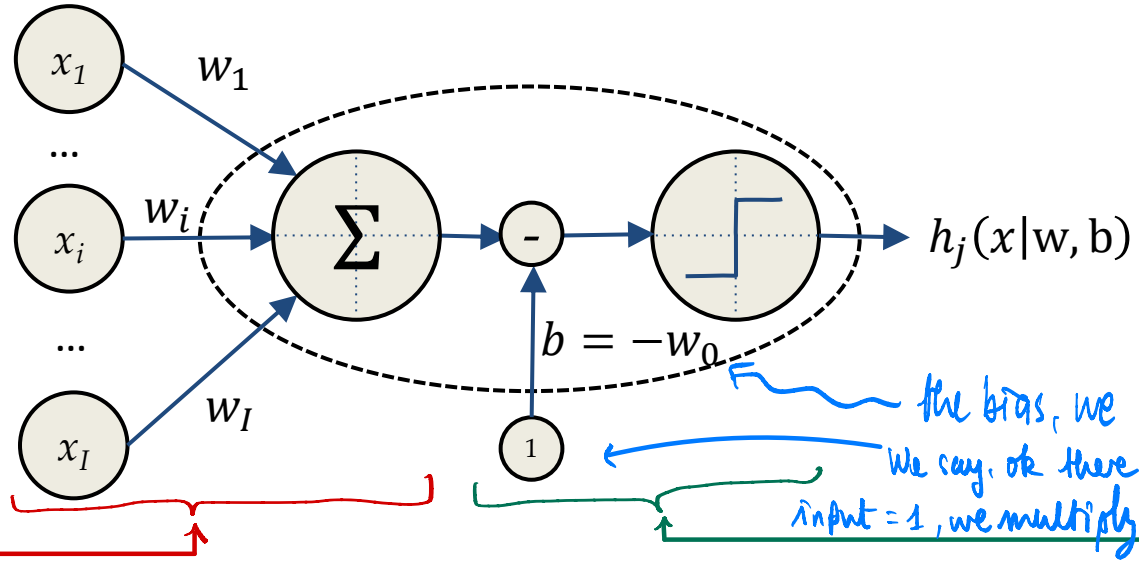




Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

- Dendrites collect charges from synapses, both Inhibitory and Excitatory
- Cumulates charge is released (neuron fires) once a Threshold is passed



the bias, we call it $-w_0$.

We say, ok there is always an input = 1, we multiply it by $-w_0$: it gives us the BIAS.

We will always write it like this, but don't forget the bias!!

$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

nonlinear function

if >0: charge is released.

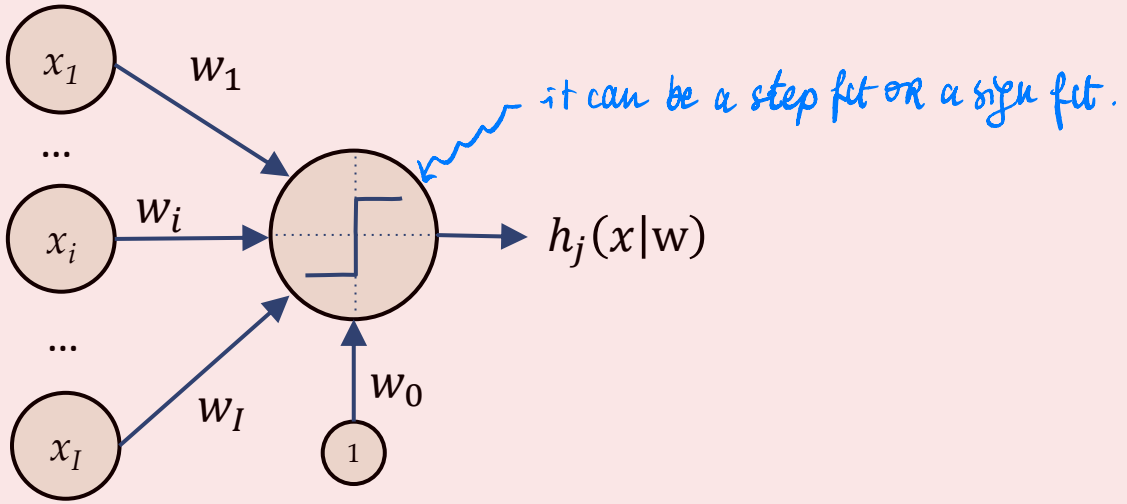
threshold = "bias" = $b = -w_0$

So using the notation $b = -w_0$, we can write it in a simpler form.

Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

- Dendrites collect charges from synapses, both Inhibitory and Excitatory
- Cumulates charge is released (neuron fires) once a Threshold is passed



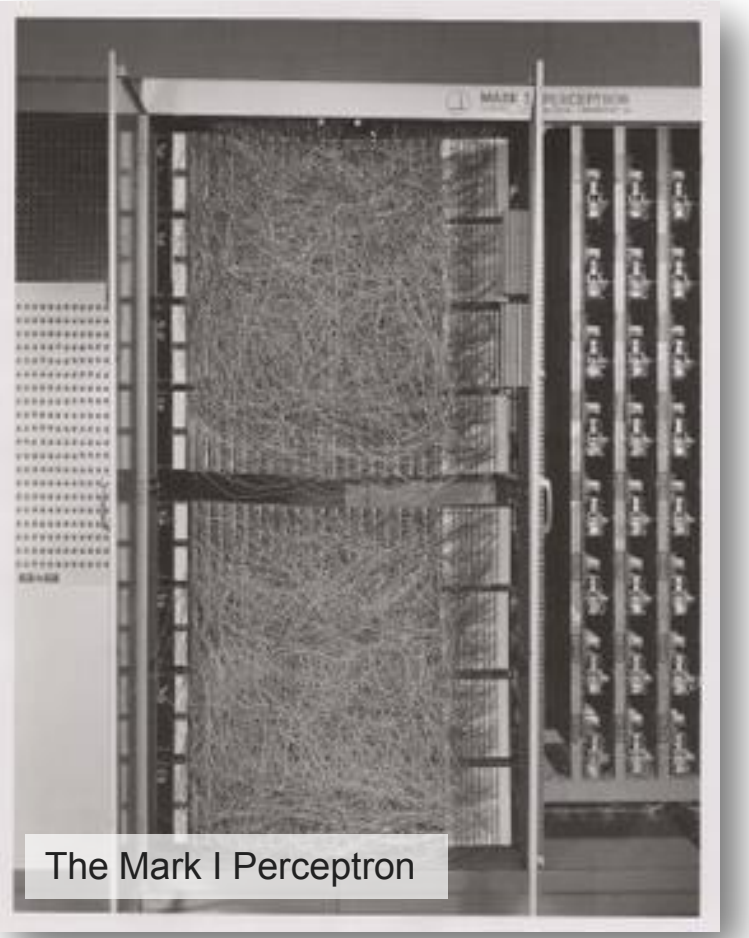
$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

VERY IMPORTANT NOTATION

Who did it first?

Several researchers were investigating models for the brain

- In 1943, Warren McCullog and Walter Harry Pitts proposed the Treshold Logic Unit or Linear Unit, the activation function was a threshold unit equivalent to the Heaviside step function
- In 1957, Frank Rosemblatt developed the first Perceptron. Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors
- In 1960, Bernard Widrow introduced the idea of representing the threshold value as a bias term in the ADALINE (Adaptive Linear Neuron or later Adaptive Linear Element)

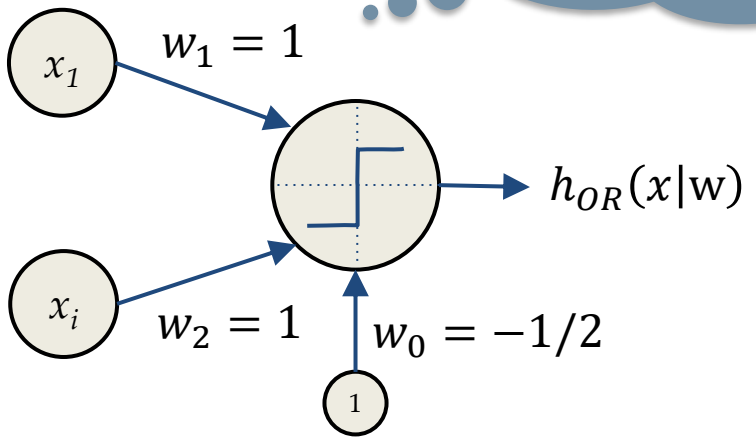


The Mark I Perceptron

What can you do with it?

Perceptron as Logical OR

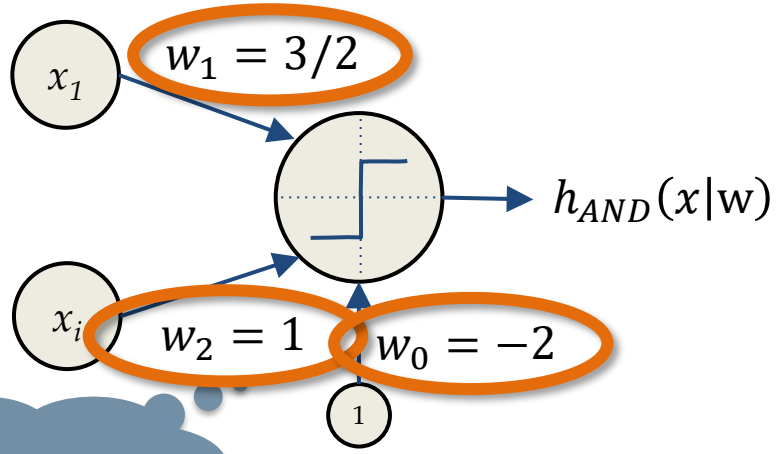
x_0	x_1	x_2	OR
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



$$h_{OR}(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2) = h_{OR}\left(-\frac{1}{2} + x_1 + x_2\right) = \begin{cases} 1, & if\ \left(-\frac{1}{2} + x_1 + x_2\right) > 0 \\ 0, & otherwise \end{cases}$$

x_0	x_1	x_2	AND
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

input always = 1



$$h_{AND}(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2) = h_{AND}\left(-2 + \frac{3}{2}\, x_1 + x_2\right) = \begin{cases} 1, & if\ \left(-2 + \frac{3}{2} x_1 + x_2\right) > 0 \\ 0, & otherwise \end{cases}$$

Perceptron as Logical AND

Hebbian Learning

"The strength of a synapse increases according to the simultaneous activation of the relative input and the desired target"

(Donald Hebb, The Organization of Behavior, 1949)

Hebbian learning can be summarized by the follow

$$w_i^{k+1} = w_i^k + \Delta w_i^k$$

$$\Delta w_i^k = \eta \cdot x_i^k \cdot t^k$$

Start from a random initialization

*Fix the weights one sample at the time (**online**), and only if the sample is not correctly predicted*

Where we have:

- η: learning rate
- x_i^k: the i^{th} perceptron input at time k
- t^k: the desired output at time k

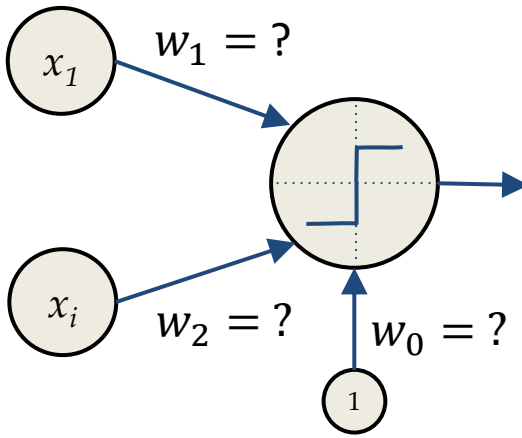
Perceptron Example

Life is easier with Sign(.)

So $0 \longleftrightarrow -1$.

Learn the weights to implement the OR operator

- Start from random weights, e.g.,
  $$w = [0 \;\; 0 \;\; 0]$$
- Chose a learning rate, e.g.,
  $$\eta = 0.5$$
- Cycle through the records by fixing those which are not correct
- End once all the records are correctly predicted



$$h(w^T x) = \begin{cases} 1 \; if w^T x > 0 \\ 0 \; if w^T x = 0 \\ -1 \; if w^T x < 0 \end{cases}$$

BIAS

x_0	x_1	x_2	OR
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1

These are the records

Inputs

desired output

Does the procedure always converge?

YES, if ... see slide n°13.

Does it always converge to the same sets of weights?

NO.

The weights depend on the order of records, η, & initialisation!

Let's do record #1: $\begin{cases} x_0 = 1 \\ x_1 = -1 \\ x_2 = -1 \end{cases}$. With the random weights: $w = [0, 0, 0]$. so we get:

$h(0+0+0) = h(0) = 0$. ⟶ we wanted -1 → so we have to fix it.

↑ according to the def of h (of previous slide).

Fixing by hebbian learning:

- $w_0 = w_0 + \Delta w_0 = w_0 + \eta x_0 t = 0 + \frac{1}{2} \cdot 1 \cdot (-1) = -1/2$

  (η: learning rate; x_0: input (always 1); t: target (or desired output))
- $w_1 = w_1 + \Delta w_1 = w_1 + \eta x_1 t$
  $= 0 + \frac{1}{2}(-1)(-1) = +1/2$
- $w_2 = w_2 + \Delta w_2 = w_2 + \eta x_2 t = 0 + \frac{1}{2}(-1)\cdot(-1) = +1/2$

So AFTER the 1st record: $w \leftarrow [-1/2 \;\; 1/2 \;\; 1/2]$.

(*) to check that the updates we did with records 2, 3, 4... did not screw up the answer on record 1. And again, & again...

Many epochs ⟶ here if we do it, after epoch 2 it's finished!

AND then we repeat the procedure, with the new w & the 3 next records. At the end we get: $w = [\ldots \;\; \ldots \;\; \ldots]$: 1 EPOCH

A pass through the dataset

We have to pass again (*).

Perceptron Math

don't forget that we don't write the bias.

A perceptron computes a weighted sum, returns its Sign (Thresholding)

$$h_j(x|\mathrm{w}) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = Sign(w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I)$$

It is a linear classifier for which the decision boundary is the hyperplane

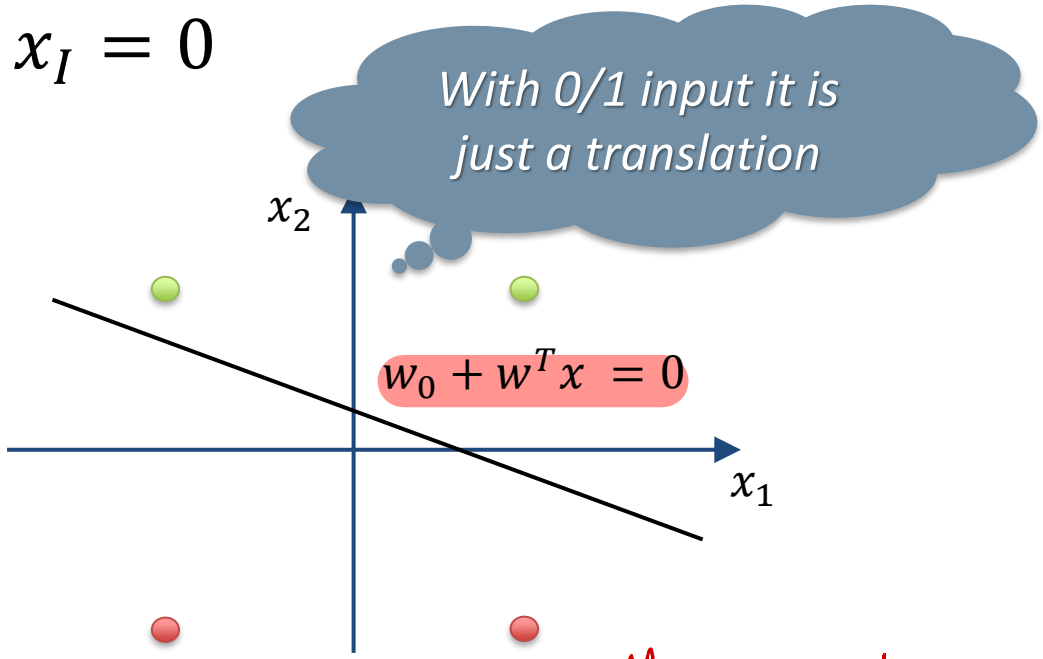
$$w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I = 0$$

In 2D, this turns into

$$w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$

$$w_2 \cdot x_2 = -w_0 - w_1 \cdot x_1$$

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} \cdot x_1$$: it's a line.



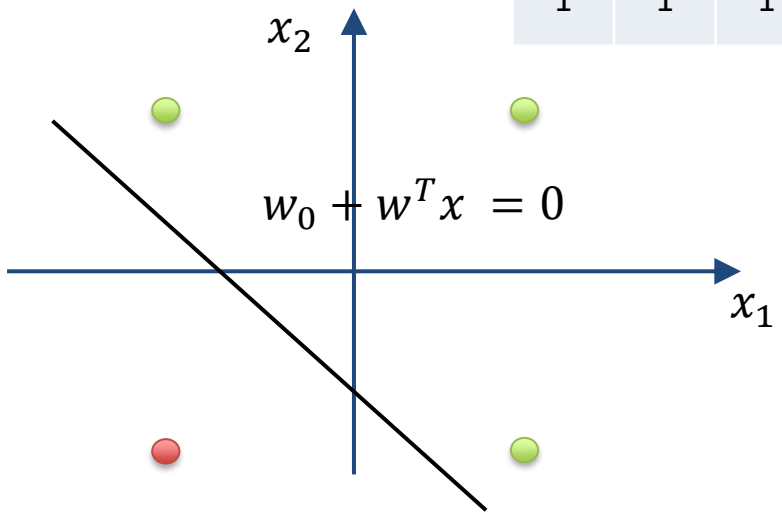
So the perceptron finds a line that keeps all the points >0 : above; & all <0 : below

Boolean Operators are Linear Boundaries

What's about it? We had already Boolean operators

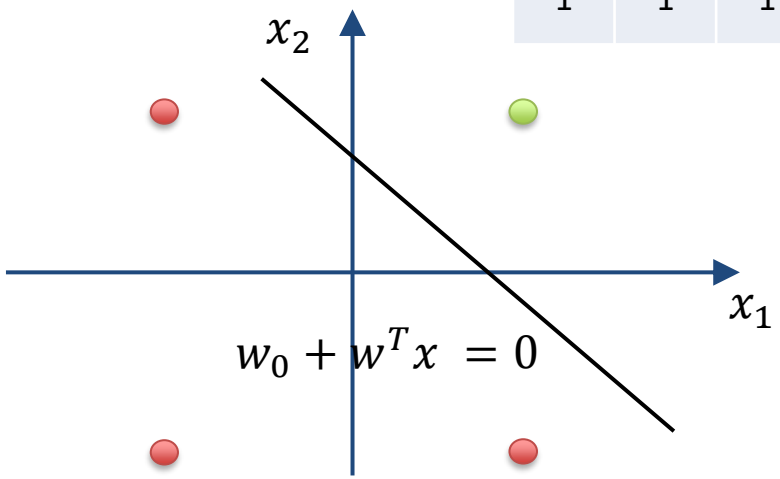
Linear boundary explains how Perceptron implements Boolean operators

x_0	x_1	x_2	OR
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1



Yes! But this ia a single trainable HW!

x_0	x_1	x_2	AND
1	-1	-1	-1
1	-1	1	-1
1	1	-1	-1
1	1	1	1

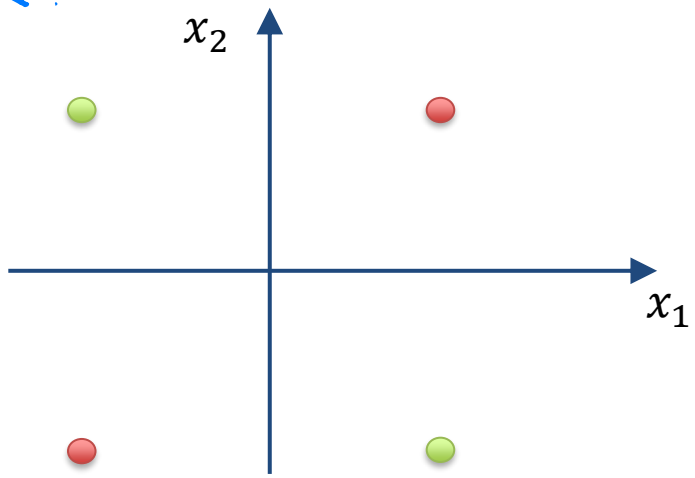


What can't you do with it?

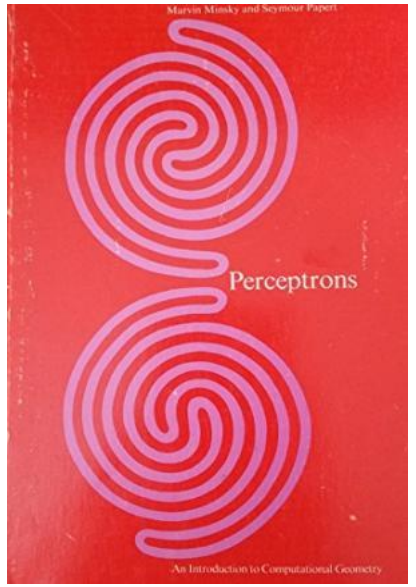
What if the dataset we want to learn does not have a linear separation boundary?

↳ Like the XOR.

x_0	x_1	x_2	XOR
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	-1



Marvin Minsky, Seymour Papert
"*Perceptrons: an introduction to computational geometry*" 1969.



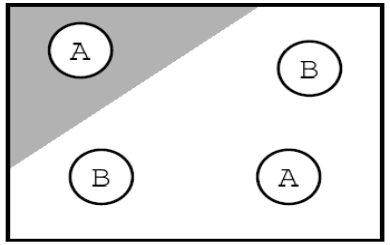
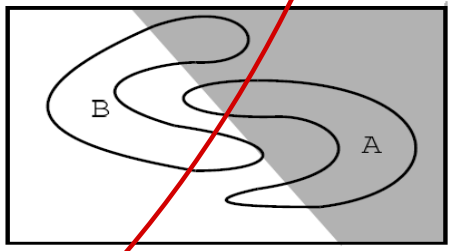
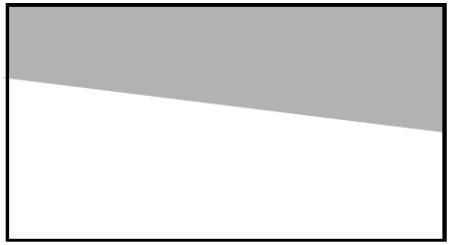
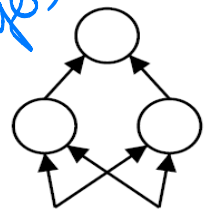
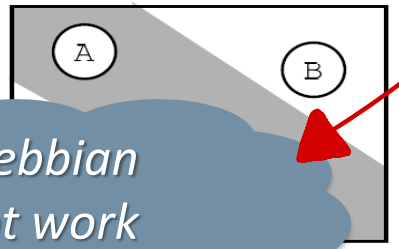
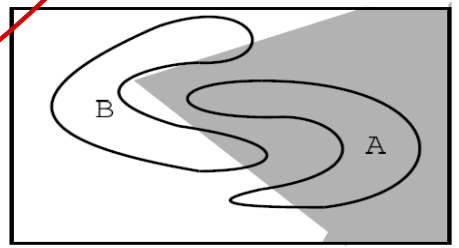
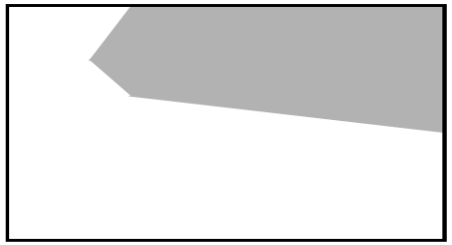
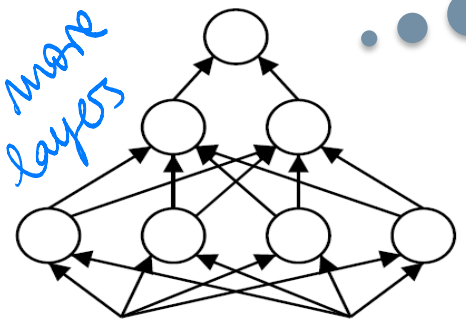
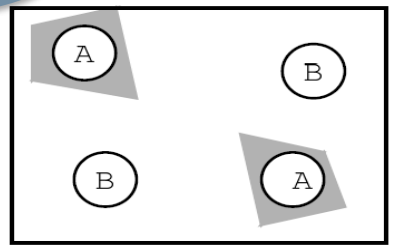
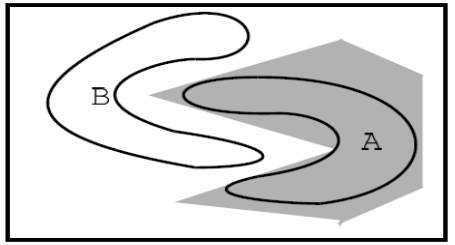
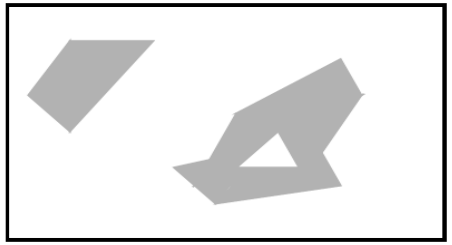
The Perceptron does not work any more and we need alternative solutions

- Non linear boundary
- Alternative input representations

The idea behind Multi Layer Perceptrons

What can't you do with it?

Difficulty : TRAINING

Topology	Type of Decision Region	XOR Problem	Classes with Meshed Regions	Most General Region Shapes
	Half bounded by hyperplanes			
2 layers 	Convex Open or			
more layers 	Arbitrary Regions (Complexity limited by the number of nodes)			

Unfortunately Hebbian learning does not work any more …

Layer Perceptrons

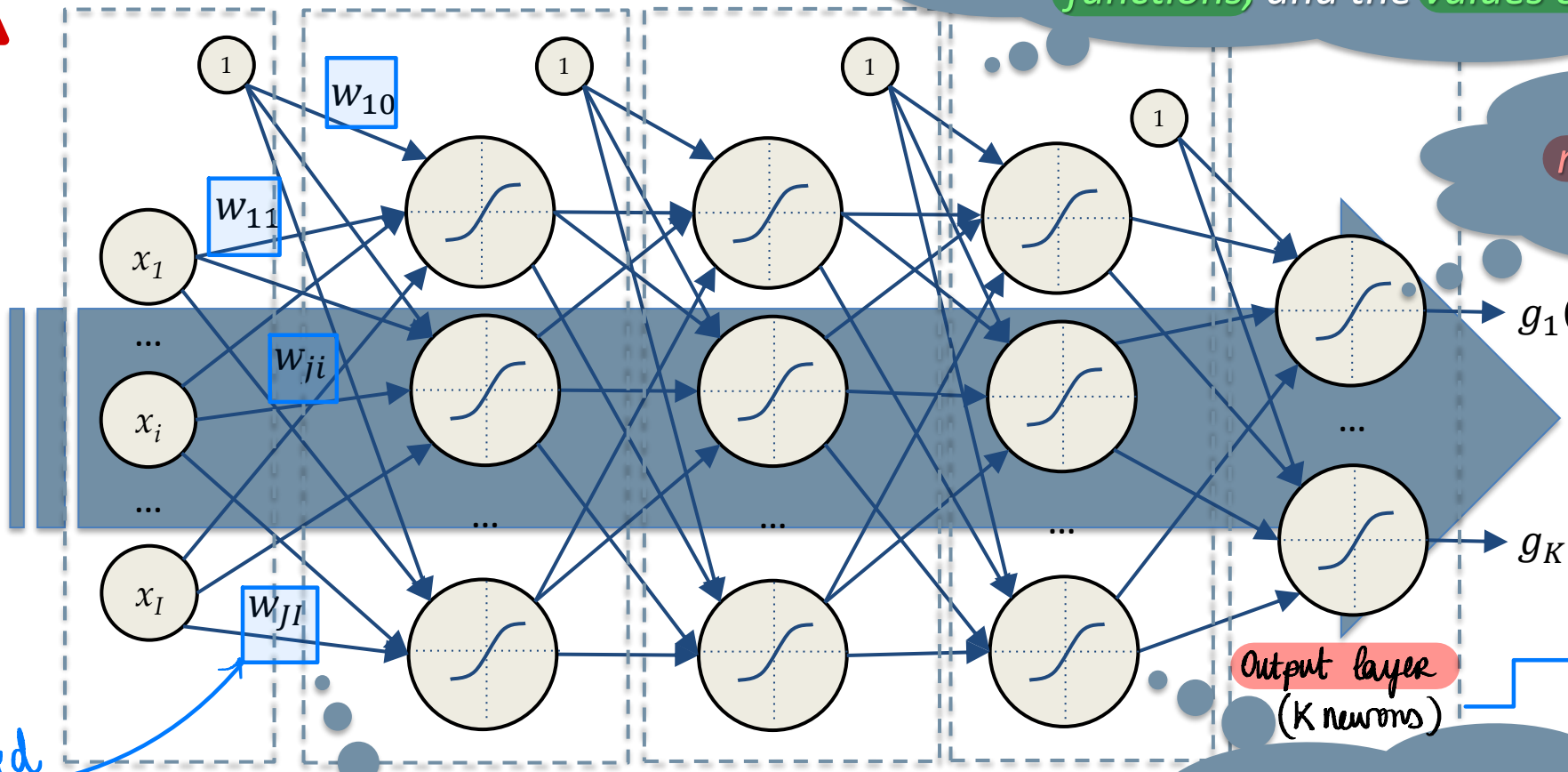
END OF LECTURE 2

Feed Forward Neural Networks

Non-linear model characterized by the number of neurons, activation functions, and the values of weights.

output vector is generated via a highly non-linear fct which depends on these.

Activation functions must be differentiable to train it ...



Layers are connected through weights $W^{(l)} = \{w_{ji}^{(l)}\}$

hidden neuron j. input i.

Deciding how many neurons in each hidden layers = Hyper-parameters tuning.

The output of a neuron depends only on the previous layers

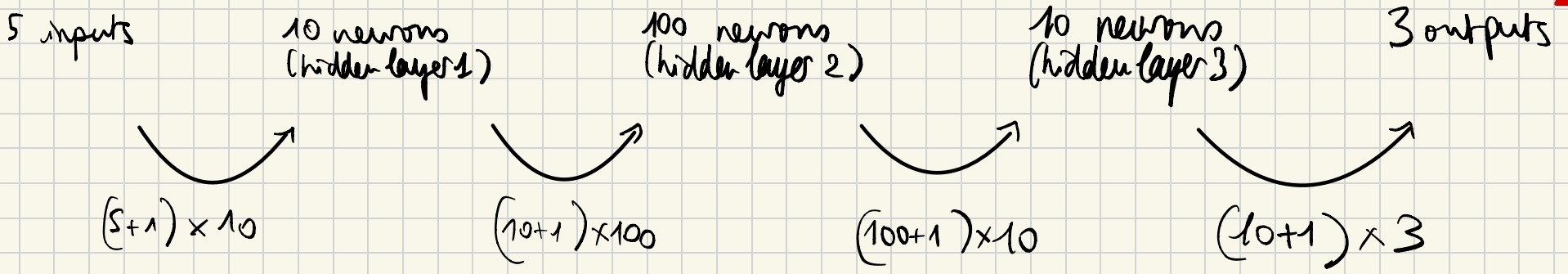
$$h^{(l)} = \{h_j^{(l)}(h^{(l-1)}, W^{(l)})\}$$

How many weights in my network?

Between Input and H.L.1 :

$J \times (I+1)$

BIAS !



As you can notice (above) the nb of weights is quadratic:
if you have HL i (100 neurons) ---- HL i+1 (100 neurons) } you get $(100+1) \times 100 \cong 100^2$ weights.

So the nb of parameters can be huge in those models.

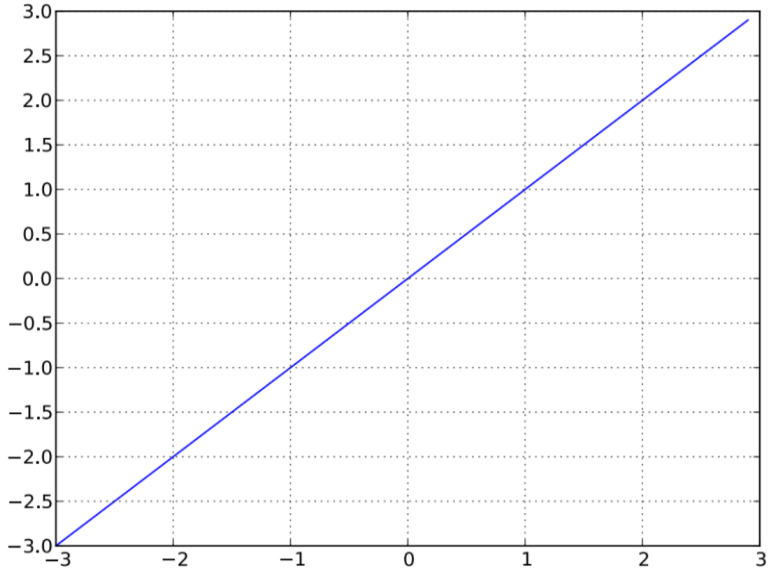
STRUCTURE: EACH LAYER IS FULLY CONNECTED W/ THE PREVIOUS ONE. (OR NEXT)

Here we work with **FEED FORWARD NEURAL NETWORK.**

Why do we like this? Because we have BACKPROPAGATION

Which Activation Function?

We will introduce other activation functions...



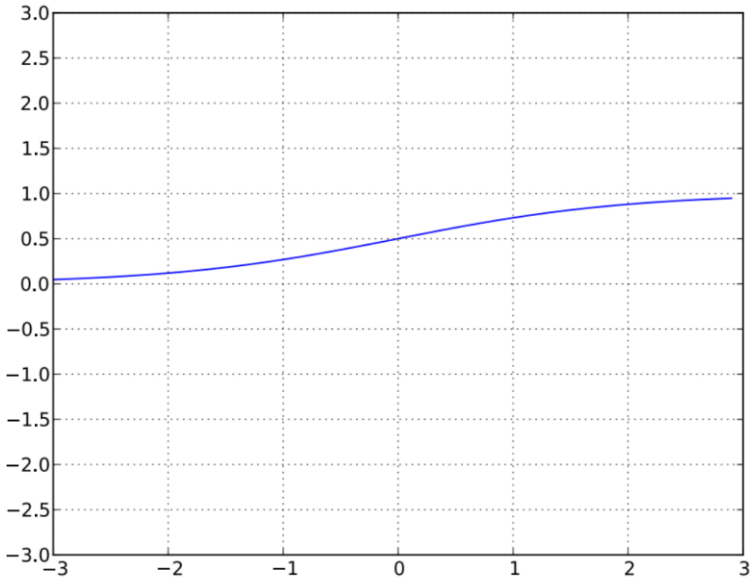
Linear activation function

Don't use only linear act. fct!

$$g(a) = a$$

$$g'(a) = 1$$

Problem: the final output will be a linear combination. That's why we need non-linear act. fct.

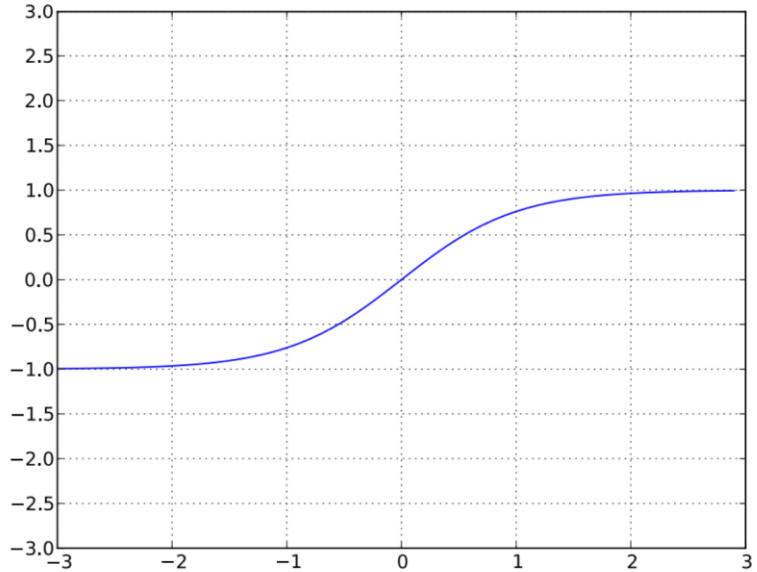


Sigmoid activation function

$$g(a) = \frac{1}{1 + \exp(-a)}$$

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

One of the most used. Easy to compute



Tanh activation function

$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

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

Output Layer in Regression and Classification

Choice of activation fct for output layer in certain cases:

For all hidden neurons use sigmoid or tanh (see later)

(1) In Regression the output spans the whole $\mathfrak{R}$ domain:

- Use a Linear activation function for the output neuron

↳ tanh or sigmoid won't work for $\mathbb{R}$-output.

(2) In Classification with two classes, chose according to their coding:

- Two classes $\{\Omega_0 = -1, \Omega_1 = +1\}$ then use Tanh output activation
- Two classes $\{\Omega_0 = 0, \Omega_1 = 1\}$ then use Sigmoid output activation (it can be interpreted as class posterior probability)

«One hot» encoding

(3) When dealing with multiple classes (K) use as many neuron as classes

- Classes are coded as $\{\Omega_0 = [0\ 0\ 1], \Omega_1 = [0\ 1\ 0], \Omega_2 = [1\ 0\ 0]\}$
- Output neurons use a softmax unit $y_k = \frac{\exp(z_k)}{\sum_k \exp(z_k)} = \frac{\exp\left(\sum_j w_{kj} h_j\left(\sum_i^I w_{ji} \cdot x_i\right)\right)}{\sum_{k=1}^K \exp\left(\sum_j w_{kj} h_j\left(\sum_i^I w_{ji} \cdot x_i\right)\right)}$

→ amplify the differences.

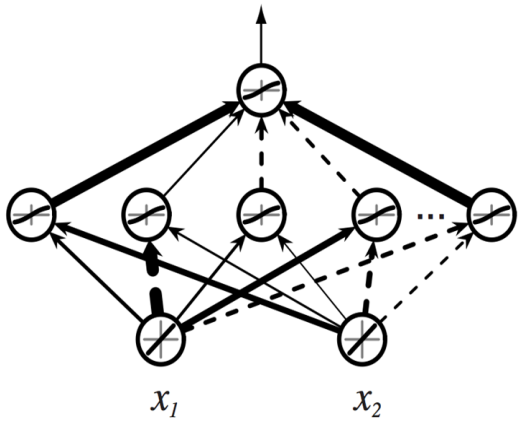
→ normalize.

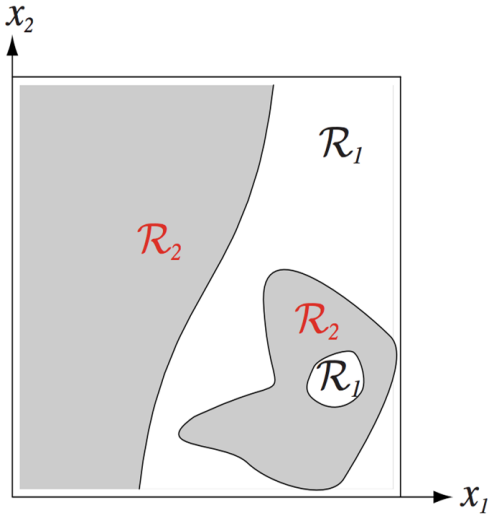
(1): Regression ($\mathbb{R}$); (2) Classification with 2 classes; (3) Classification with K classes.

Neural Networks are Universal Approximators

"A single hidden layer feedforward neural network with S shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set"

Universal approximation theorem
(Kurt Hornik, 1991)





Images from Hugo Larochelle's DL Summer School Tutorial

& it can be hard...

Regardless the function we are learning, a single layer can represent it:

i.e. we need to train it!

- Doesn't mean a learning algorithm can find the necessary weights!
- In the worse case, an exponential number of hidden units may be required ...
- The layer may have to be unfeasibly large and may fail to learn and generalize

Classification requires just one extra layer ... : so 2 hidden layers. ↔ USUAL/BASIC ARCHITECTURE.

So, firstly, we would like to learn how to find the weights. Then, we'll try to do it without overfitting!

Optimization and Learning (Supervised learning)

But @ the beginning they didn't realize ...

We will train using GRADIENT DESCENT (which is equivalent to backpropagation).

Recall learning a *parametric model* $y(x_n|\theta)$ in regression/classification

x_n: our input. θ: some set of parameters.

- Given a training set

$$D =< x_1, t_1 > \cdots < x_N, t_N >$$

- We want to find model parameters such that for new data

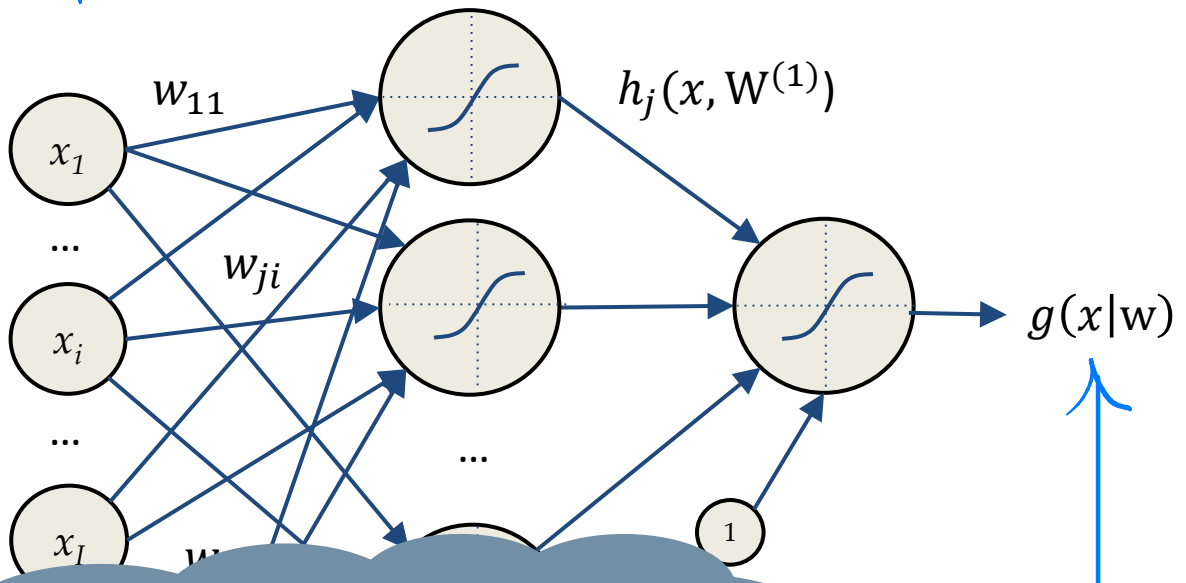
$$y(x_n|\theta) \sim t_n$$

(t_n: target)

- In case of a Neural Network this can be rewritten as

$$g(x_n|w) \sim t_n$$

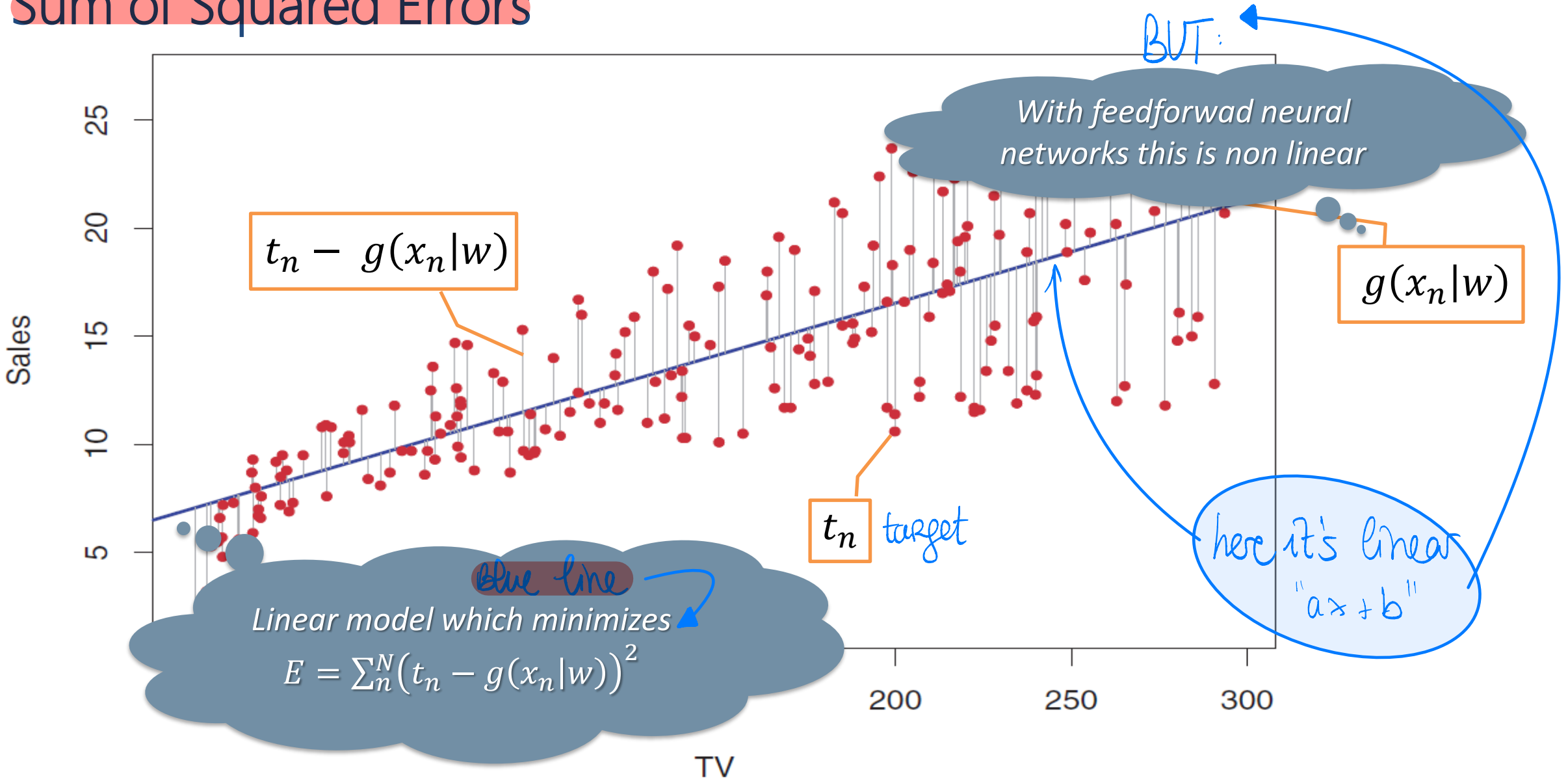
($g(x_n|w)$: output of our NN)



For this you can minimize $E = \sum_n^N (t_n - g(x_n|w))^2$ (ERROR / LOSS)

That's why we need differentiable functions.

Sum of Squared Errors



Non Linear Optimization 101

To find the minimum of a generic function, we compute the partial derivatives of the function and set them to zero

$$\frac{\partial J(w)}{\partial w} = 0$$

with highly non-linear fcts: That's why we use grad. descent.

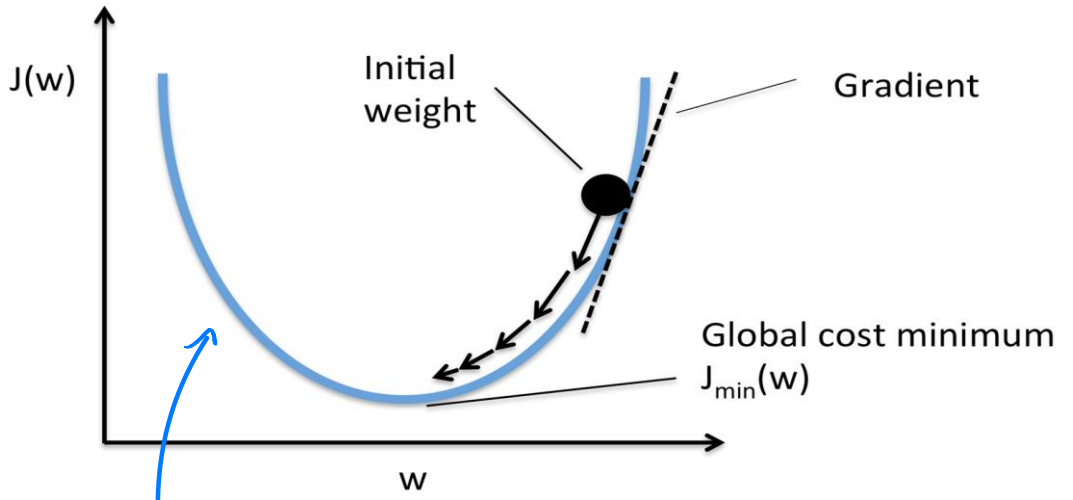
Closed-form solutions are practically never available so we can use iterative solutions (gradient descent):

GD

1) Initialize the weights to a random value
2) Iterate until convergence

$$w^{k+1} = w^k - \eta \left.\frac{\partial J(w)}{\partial w}\right|_{w^k}$$

Learning Rate.



that's a very "nice" shape: IRL, it's harder (cf next slide)

Gradient descent - Backpropagation

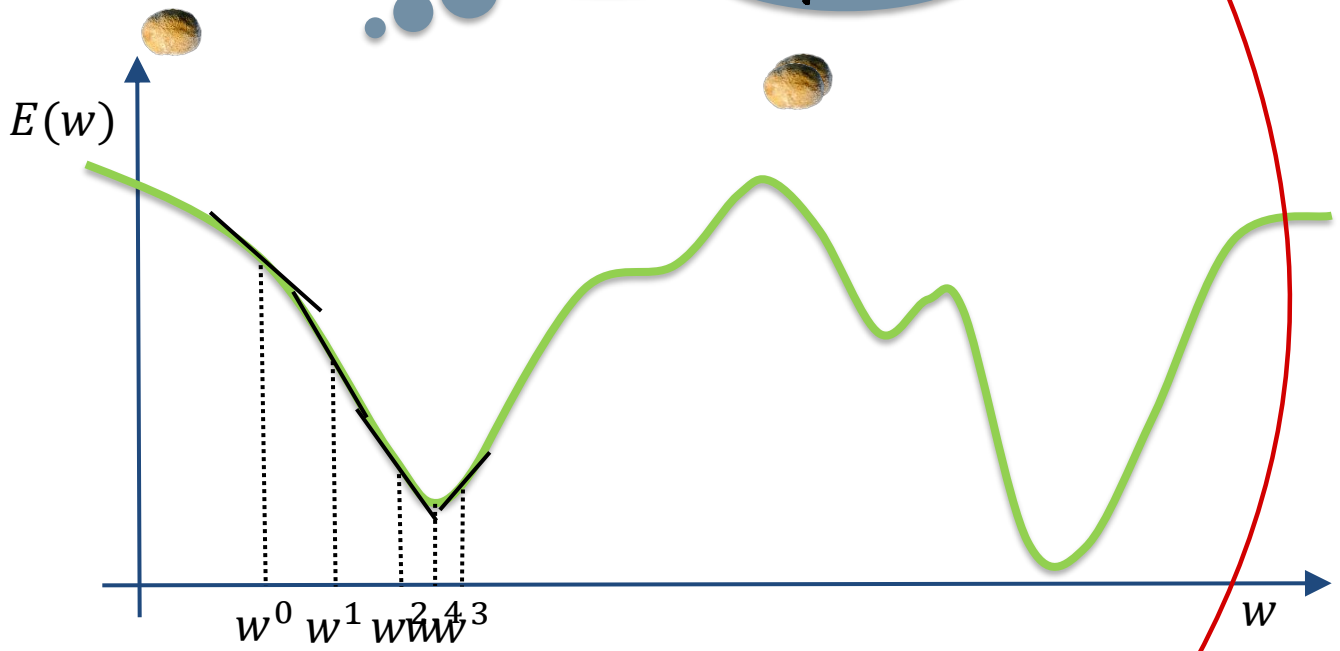
Finding the weighs of a Neural Network is a non-linear optimization:

$$argmin_w \; E(w) = \sum_{n=1}^{N} (t_n - g(x_n, w))^2$$

We iterate starting from an initial random configuration:

Iteration:

$$w^{k+1} = w^k - \eta \left. \frac{\partial E(w)}{\partial w} \right|_{w^k}$$



Use multiple restarts to seek for a proper global minimum.

It depends on where we start from

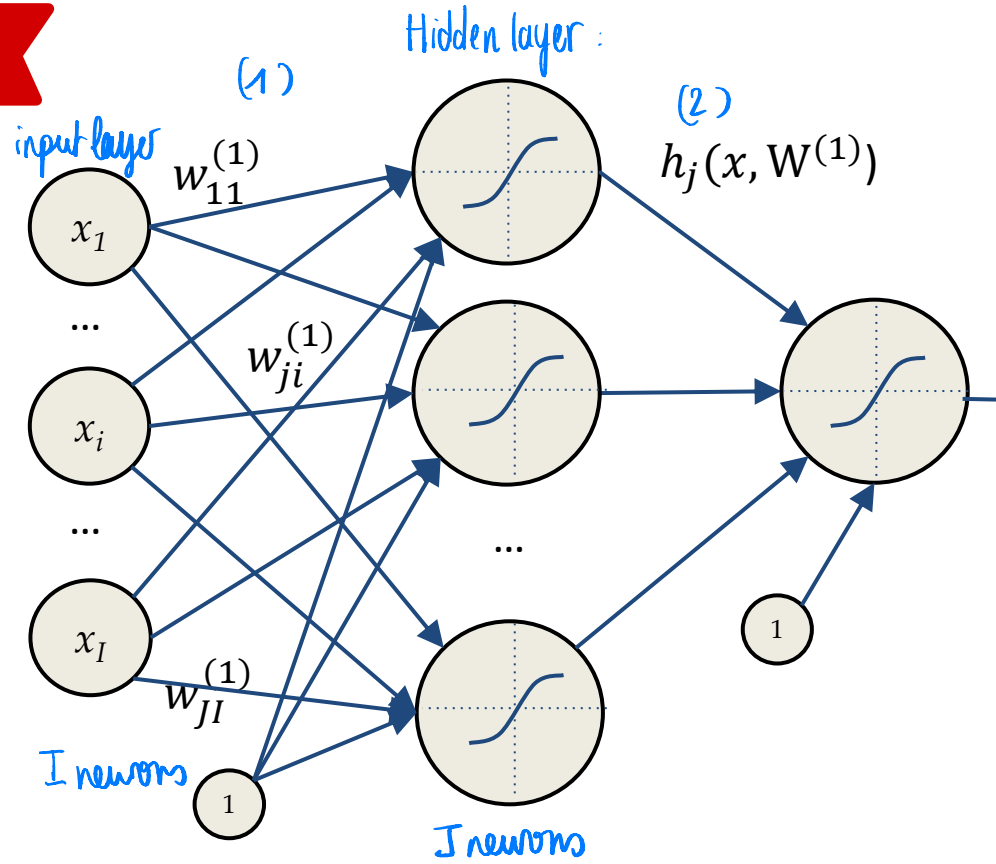
To avoid local minima can use momentum:

Like "ADAM".

$$w^{k+1} = w^k - \eta \left. \frac{\partial E(w)}{\partial w} \right|_{w^k} - \alpha \left. \frac{\partial E(w)}{\partial w} \right|_{w^{k-1}}$$

You can use past gradient value(s)...

Gradient Descent Example



This is the fct computed by the NN: "output function".

$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

out · hidden layer · input layer · inputs

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Use *j=3* and *i=5*

W_{35} : $i=5 \rightarrow j=3$

Compute the $w_{ji}^{(1)}$ weight update formula by gradient descent

Exercise. cf next slide.

$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

our starting points

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

derivation formula: $(u^2)' = 2u'u$.

$$\frac{\partial E(w)}{\partial w_{3,5}^{(1)}} = \frac{\partial \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2}{\partial w_{3,5}^{(1)}} = \sum_{n=1}^{N} \frac{\partial (t_n - g_1(x_n, w))^2}{\partial w_{3,5}^{(1)}} = -2 \sum_{n} (t_n - g_1(x_n, w)) \frac{\partial g_1(x_n, w)}{\partial w_{3,5}^{(1)}}$$

$(g \circ f)'(x) = g'(f(x)) \times f'(x)$.

$$\frac{\partial g_1(x_n, w)}{\partial w_{3,5}^{(1)}} = \frac{\partial g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j(.)\right)}{\partial w_{3,5}^{(1)}} = \underbrace{g_1'(x_n, w)}_{\text{"}g'(f(x))\text{"}} \cdot \underbrace{\frac{\partial \sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j(.)}{\partial w_{3,5}^{(1)}}}_{\text{"}f'(x)\text{"}} = g_1'(x_n, w) \cdot w_{1,3}^{(2)} \cdot \frac{\partial h_3\left(\sum_{i=0}^{I} w_{3i}^{(1)} \cdot x_{i,n}\right)}{\partial w_{3,5}^{(1)}}$$

see previous page.

$$\frac{\partial h_3\left(\sum_{i=0}^{I} w_{3i}^{(1)} \cdot x_{i,n}\right)}{\partial w_{3,5}^{(1)}} = h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) \frac{\partial \sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}}{\partial w_{3,5}^{(1)}} = h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) x_{5,n}$$

only 1 term depends on $w_{3,5}^{(1)}$.

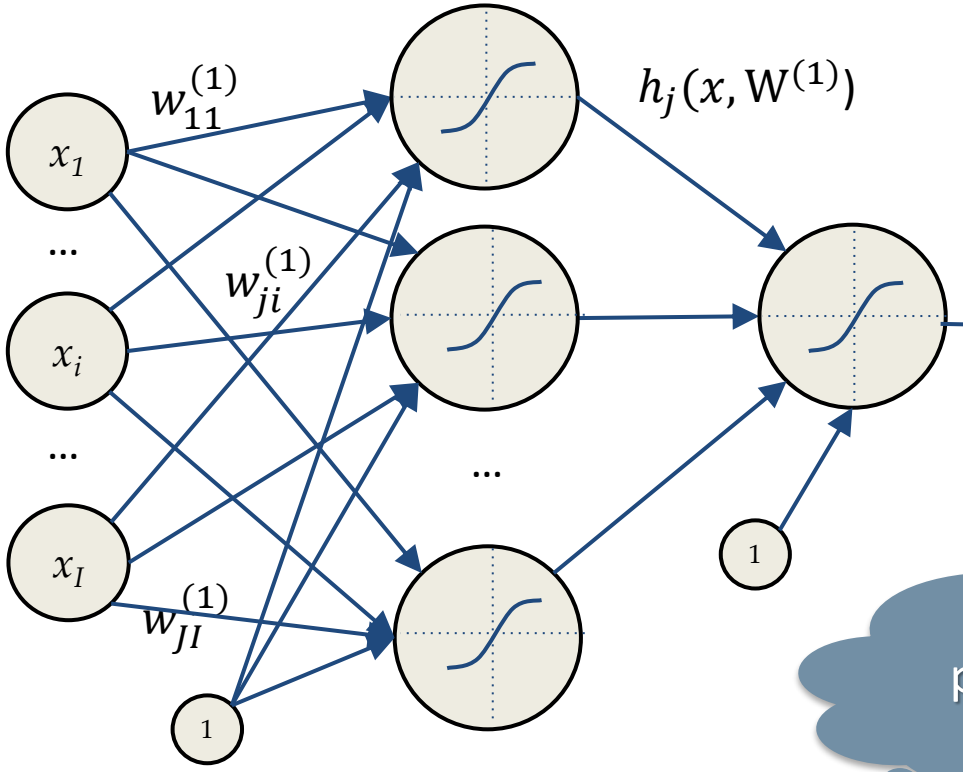
Gradient part of the backpropagation formula for $w_{3,5}^{(1)}$.

$$\frac{\partial E(w)}{\partial w_{3,5}^{(1)}} = -2 \sum_{n}^{N} (t_n - g_1(x_n, w)) g_1'(x_n, w) w_{1,3}^{(2)} h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) x_{5,n}$$

For all weights of (1), you get the same; you just have to change indexes (like $w_{1,3}^{(2)}$ eg)

⚠ But you have to recompute for (2)!

Gradient Descent Example



$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

$$E(w) = \sum^{N} \left(t_n - g_1(x_n, w)\right)^2$$

Using all the data points (**batch**) might be unpractical → sometimes all the data CANNOT FIT in the RAM of the GPU.

$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = -2 \sum_{n}^{N} \left(t_n - g_1(x_n, w)\right) g_1'(x_n, w) w_{1j}^{(2)} h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right) x_i$$

↓ of next slide.

Gradient Descent Variations

Batch gradient descent : to solve the problem (cf previous slide), we can do 1 sample @ a time.

$$\frac{\partial E(w)}{\partial w} = \frac{1}{N}\sum_{n}^{N}\frac{\partial E(x_n, w)}{\partial w}$$

Use a single sample, unbiased, but with high variance

Stochastic gradient descent (SGD)

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{SGD}(w)}{\partial w} = \frac{\partial E(x_n, w)}{\partial w}$$

Use a subset of samples, good trade off variance-computation

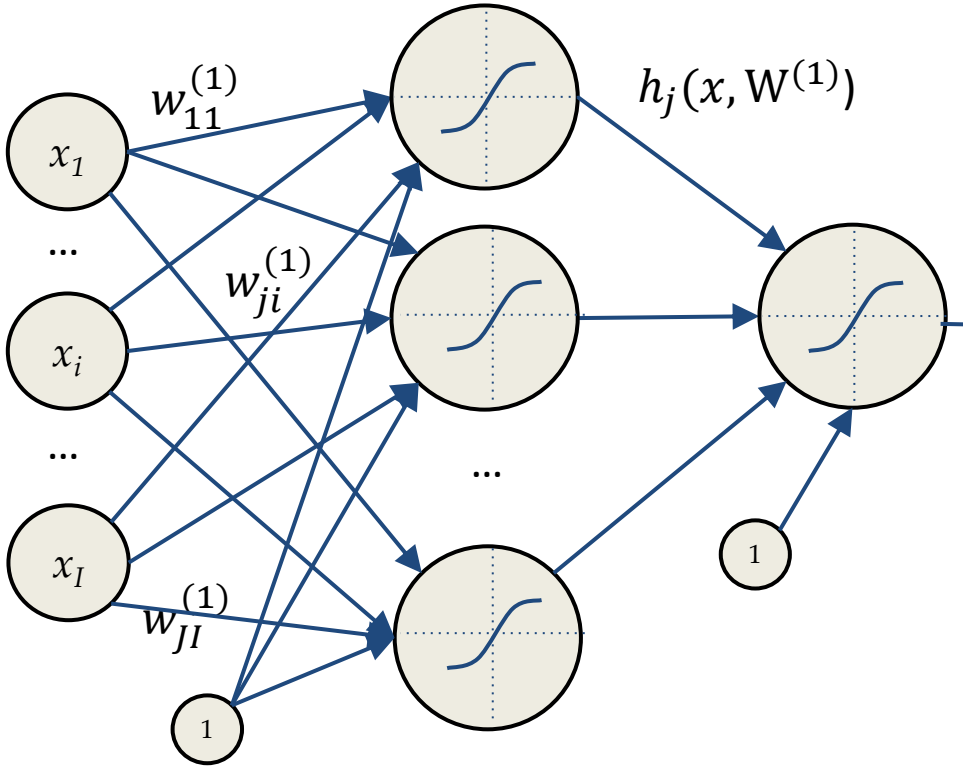
Mini-batch gradient descent

↑ Not the full dataset: a subset.

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{MB}(w)}{\partial w} = \frac{1}{M}\sum_{n\in Minibatch}^{M<N}\frac{\partial E(x_n, w)}{\partial w}$$

Gradient Descent Example

Do I have to compute the gradient myself?



$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Can I make it automatic?

$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = -2 \sum_{n}^{N} \left(t_n - g_1(x_n, w)\right) g_1'(x_n, w) w_{1j}^{(2)} h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right) x_i$$

Backpropagation and Chain Rule (1)

We have "weighted sum of non-linear functions of weighted sum..." : NESTED Structure.

⚠ Weights update can be done in parallel, locally, and requires just 2 passes

- Let x be a real number and two functions $f: \Re \to \Re$ and $g: \Re \to \Re$
- Consider the composed function $z = f(g(x)) = f(y)$ where $y = g(x)$
- The derivative of z w.r.t. x can be computed applying the chain rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

⚠ PROOF: $\frac{d f(g(x))}{dx} = g'(x) \times f'(g(x)) = \frac{dg(x)}{dx} \times \frac{df(g(x))}{dg(x)} = \frac{dy}{dx} \times \frac{dz}{dy}$

→ The same holds for backpropagation

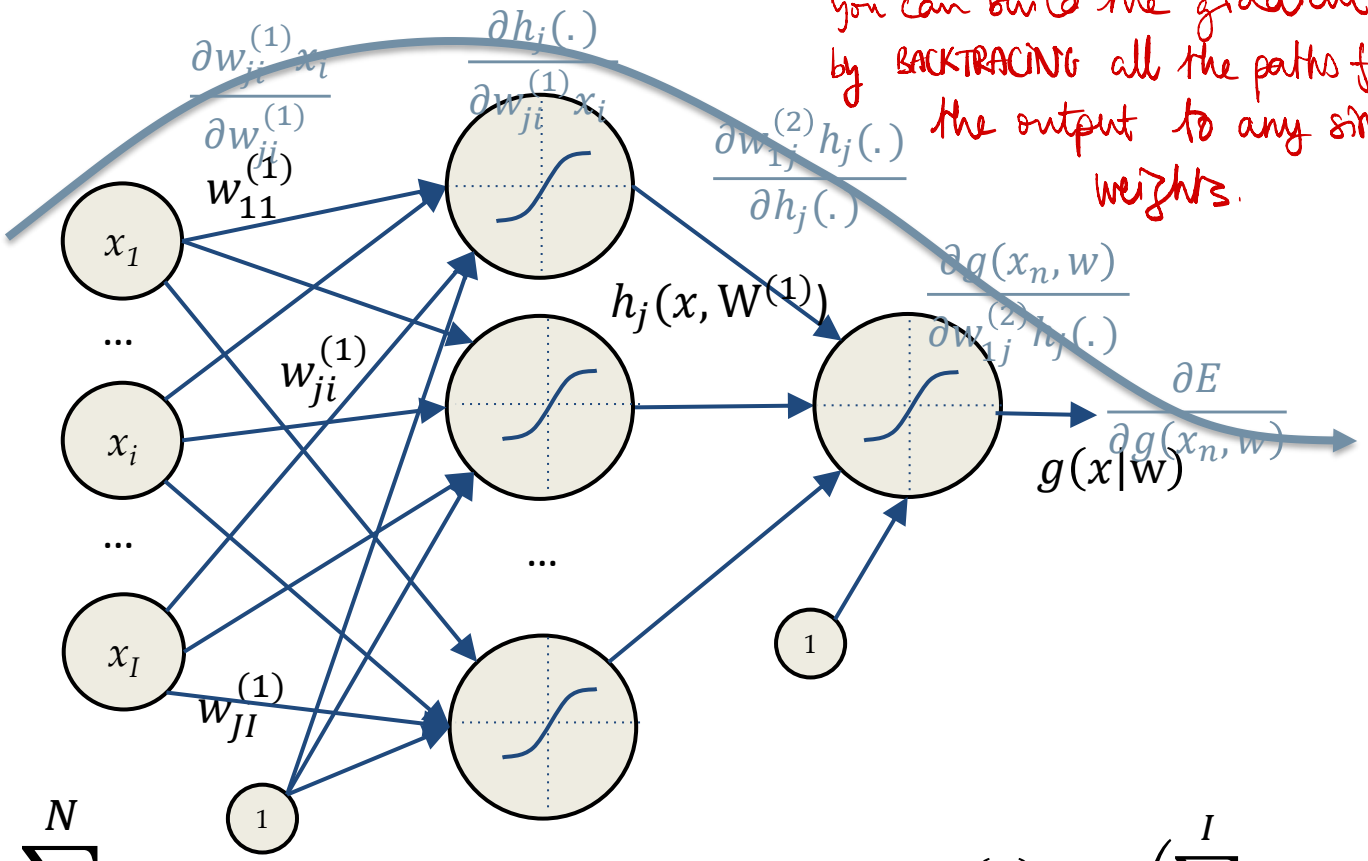
$$\underbrace{\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}}}_{\frac{\partial E}{\partial w_{ji}^{(1)}}} = \underbrace{-2 \sum_{n}^{N} \left(t_n - g_1(x_n, w)\right)}_{\frac{\partial E}{\partial g(x_n, w)}} \cdot \underbrace{g_1'(x_n, w)}_{\frac{\partial g(x_n, w)}{\partial w_{1j}^{(2)} h_j(.)}} \cdot \underbrace{w_{1j}^{(2)}}_{\frac{\partial w_{1j}^{(2)} h_j(.)}{\partial h_j(.)}} \cdot \underbrace{h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)}_{\frac{\partial h_j(.)}{\partial w_{ji}^{(1)} x_i}} \cdot \underbrace{x_i}_{\frac{\partial w_{ji}^{(1)} x_i}{\partial w_{ji}^{(1)}}}$$

If we want to change one activation fct: we don't need to recompute everything.

Backpropagation and Chain Rule (2)

"forward-backward algorithm"

you can build the gradient by BACKTRACING all the paths from the output to any single weights.



Forward pass: you compute all the weighted sums ⊕ all the blocks: they are LOCAL, they don't depend on other neurons.

$$\underbrace{\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}}}_{\frac{\partial E}{\partial w_{ji}^{(1)}}} = \underbrace{-2 \sum_{n}^{N} \left(t_n - g_1(x_n, w)\right)}_{\frac{\partial E}{\partial g(x_n, w)}} \cdot \underbrace{g_1'(x_n, w)}_{\frac{\partial g(x_n, w)}{\partial w_{1j}^{(2)} h_j(.)}} \cdot \underbrace{w_{1j}^{(2)}}_{\frac{\partial w_{1j}^{(2)} h_j(.)}{\partial h_j(.)}} \cdot \underbrace{h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)}_{\frac{\partial h_j(.)}{\partial w_{ji}^{(1)} x_i}} \cdot \underbrace{x_i}_{\frac{\partial w_{ji}^{(1)} x_i}{\partial w_{ji}^{(1)}}}$$

Backpropagation and Chain Rule (2)

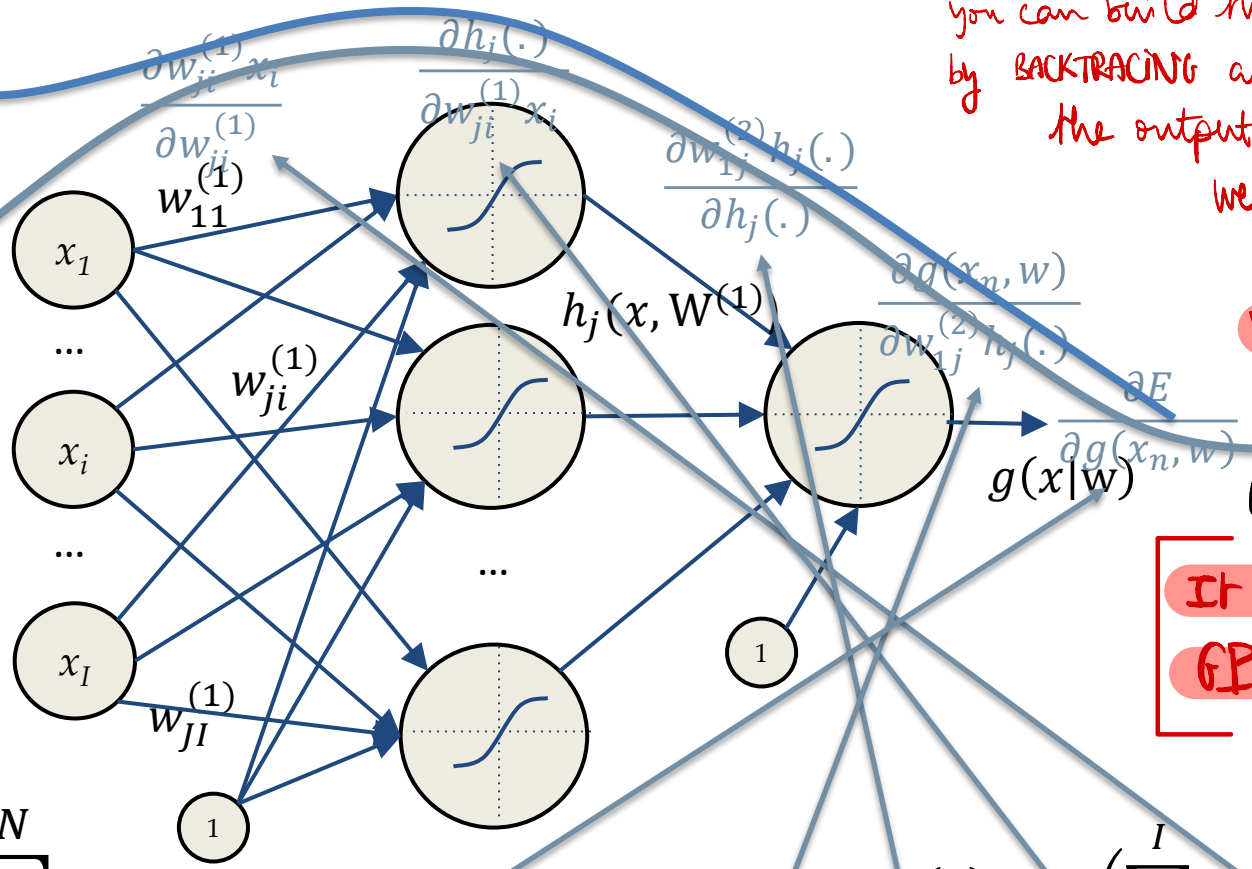
"forward-backward algorithm"

you can build the gradient by BACKTRACING all the paths from the output to any single weights.

you can evaluate every weights

(you multiply everything)
Backward pass

you multiply everything together to get $\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}}$



Forward pass
(you compute every values)

IT IS VERY efficient on GPUs (perfect for // computing)

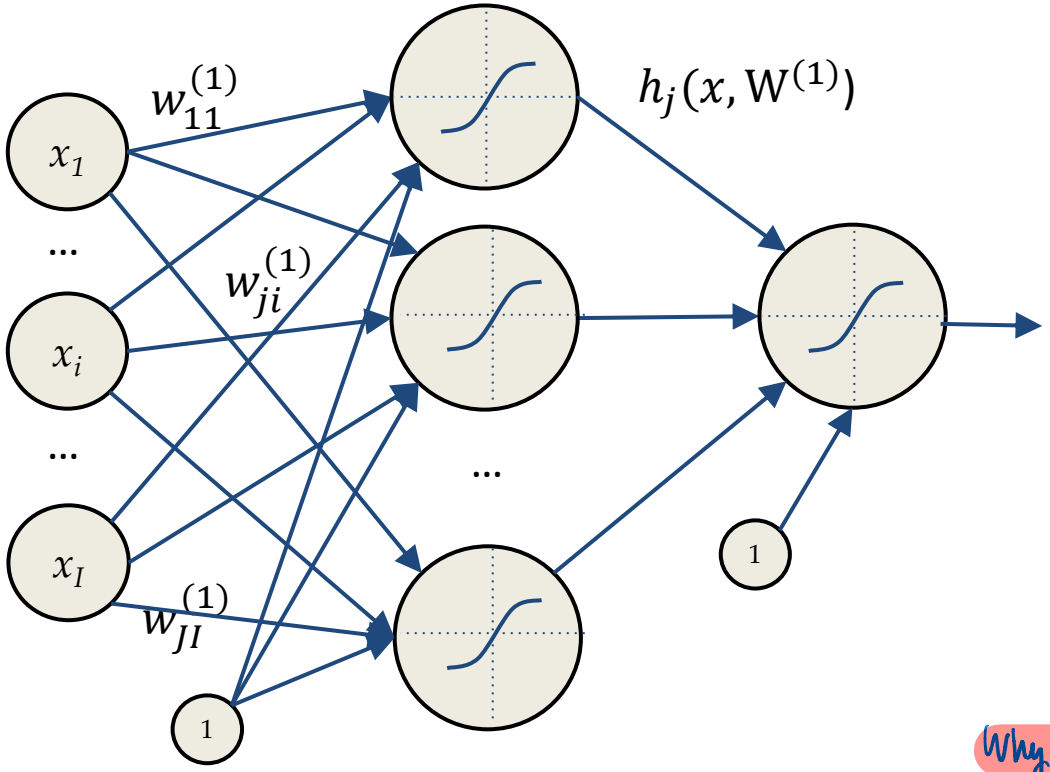
$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = -2 \sum_{n}^{N} \left(t_n - g_1(x_n, w)\right) \cdot g_1'(x_n, w) \cdot w_{1j}^{(2)} \cdot h_j' \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right) \cdot x_i$$

$\frac{\partial E}{\partial w_{ji}^{(1)}}$ $\frac{\partial E}{\partial g(x_n, w)}$ $\frac{\partial g(x_n, w)}{\partial w_{1j}^{(2)} h_j(.)}$ $\frac{\partial w_{1j}^{(2)} h_j(.)}{\partial h_j(.)}$ $\frac{\partial h_j(.)}{\partial w_{ji}^{(1)} x_i}$ $\frac{\partial w_{ji}^{(1)} x_i}{\partial w_{ji}^{(1)}}$

END OF LECTURE 3

Python's library can do this very easily!

Gradient Descent Example



$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Why is this a good "Loss" (or "ERROR") function?

Why should I use this?

$$\frac{\partial E(w_{ji}^{(1)}))}{\partial w_{ji}^{(1)}} = -2 \sum_{n}^{N} (t_n - g_1(x_n, w)) g_1'(x_n, w) w_{1j}^{(2)} h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right) x_{i,n}$$

Let's go back to the basics...

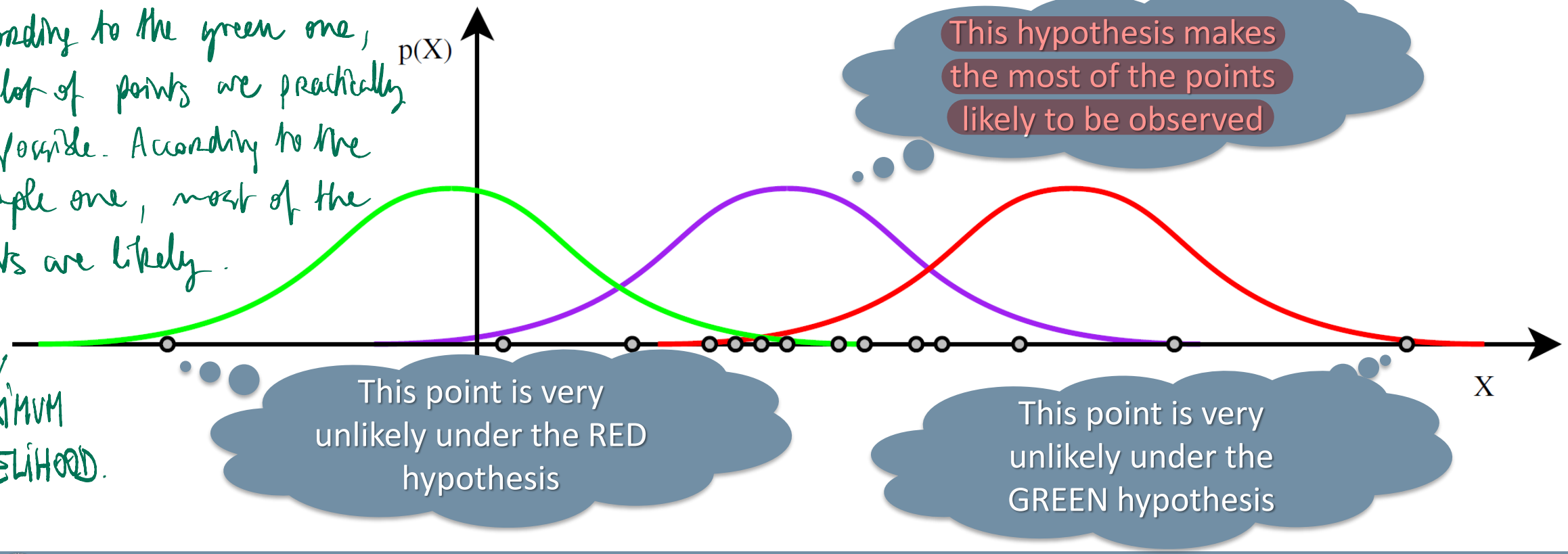
A Note on Maximum Likelihood Estimation

Let's observe i.i.d. samples from a Gaussian distribution with known σ^2

$$x_1, x_2, \ldots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

According to the green one, a lot of points are practically impossible. According to the purple one, most of the points are likely.

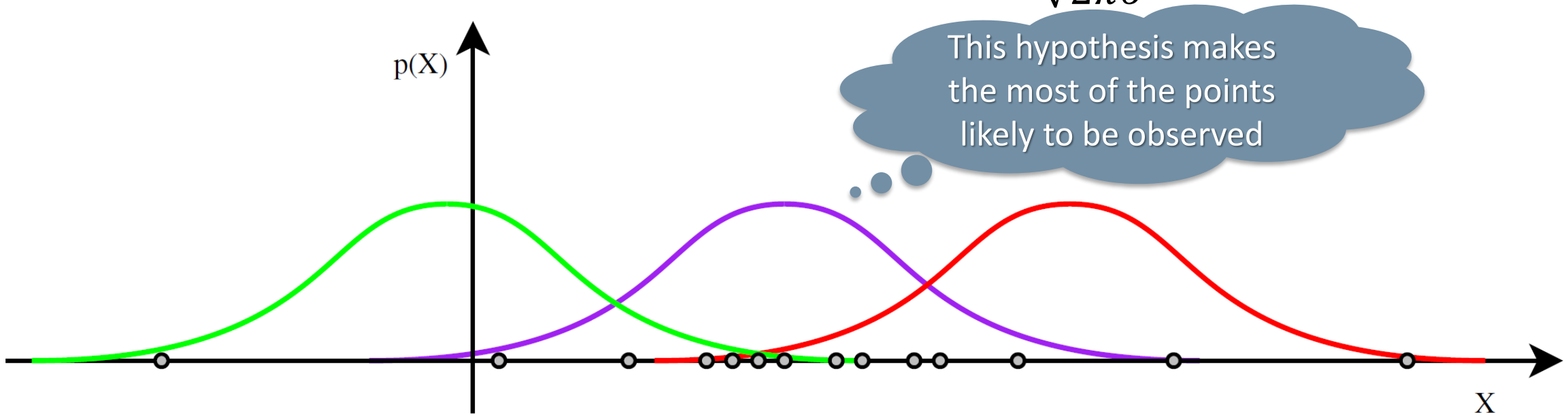
MAXIMUM LIKELIHOOD.



A Note on Maximum Likelihood Estimation

Let's observe i.i.d. samples from a Gaussian distribution with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Maximum Likelihood: Chose parameters which maximize data probability

Maximum Likelihood Estimation: The Recipe

Let $\theta = (\theta_1, \theta_2, \ldots, \theta_p)^T$ a vector of parameters, find the MLE for θ:

- Write the likelihood $L = P(Data|\theta)$ for the data
- [Take the logarithm of likelihood $\mathrm{l} = \log P(Data|\theta)$]
- Work out $\frac{\partial L}{\partial \theta}$ or $\frac{\partial l}{\partial \theta}$ using high-school calculus
- Solve the set of simultaneous equations $\frac{\partial L}{\partial \theta_i} = 0$ or $\frac{\partial l}{\partial \theta_i} = 0$
- Check that θ^{MLE} is a maximum

Optional

We know already about gradient descent, let's try with some analitical stuff ...

To maximize/minimize the (log)likelihood you can:

- Analytical Techniques (i.e., solve the equations)
- Optimization Techniques (e.g., Lagrange multipliers)
- Numerical Techniques (e.g., gradient descend)

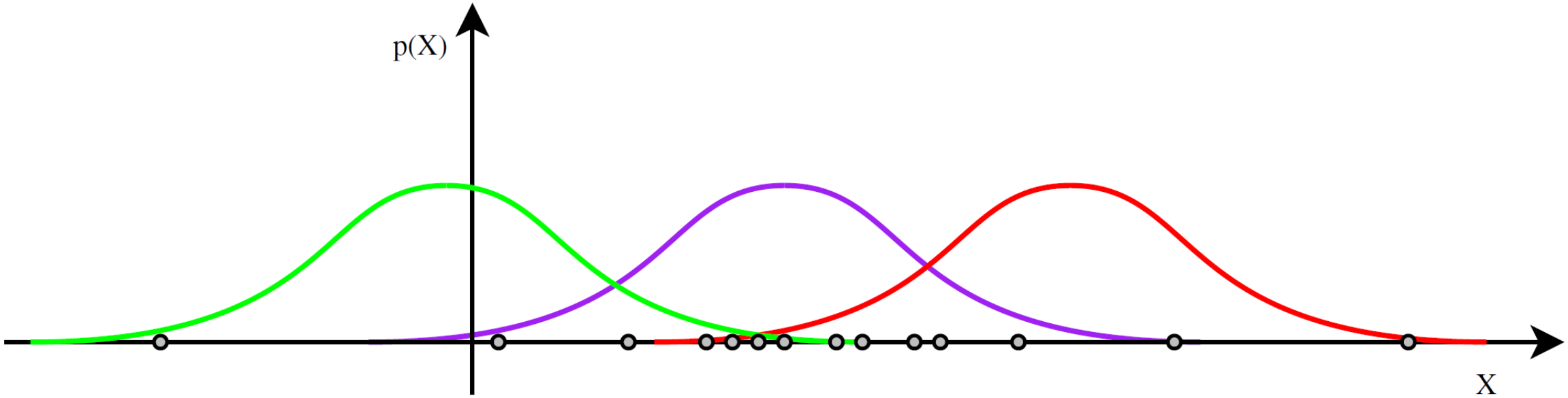
← the 3 main techniques!

Maximum Likelihood Estimation Example

Let's observe i.i.d. samples coming from a Gaussian with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2)$$

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Find the Maximum Likelihood Estimator for μ

Maximum Likelihood Estimation Example

Let's observe i.i.d. samples coming from a Gaussian with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

1. Write the likelihood $L = P(Data|\theta)$ for the data

INDEPENDENT RV

$$L(\mu) = p(x_1, x_2, \dots, x_N|\mu, \sigma^2) = \prod_{n=1}^{N} p(x_n|\mu, \sigma^2) =$$

$$= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}$$

Maximum Likelihood Estimation Example

Let's observe i.i.d. samples coming from a Gaussian with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

2. Take the logarithm $l = \log P(Data|\theta)$ of the likelihood

$$l(\mu) = \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2 \cdot \pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\cdot\sigma^2}}\right) = \sum_{n=1}^{N} \log \frac{1}{\sqrt{2 \cdot \pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\cdot\sigma^2}} =$$

$$= N \cdot \log \frac{1}{\sqrt{2 \cdot \pi}\sigma} - \frac{1}{2 \cdot \sigma^2} \sum_{n}^{N} (x_n - \mu)^2$$

Maximum Likelihood Estimation Example

Let's observe i.i.d. samples coming from a Gaussian with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

3. Work out $\partial l / \partial \theta$ using high-school calculus

$$\frac{\partial l(\mu)}{\partial \mu} = \frac{\partial}{\partial \mu}\left(N \cdot \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_n^N (x_n - \mu)^2 \right) =$$

$$= -\frac{1}{2\sigma^2} \frac{\partial}{\partial \mu} \sum_n^N (x_n - \mu)^2 = \frac{1}{2\sigma^2} \sum_n^N 2(x_n - \mu)$$

Maximum Likelihood Estimation Example

Let's observe i.i.d. samples coming from a Gaussian with known σ^2

$$x_1, x_2, \dots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

4 • Solve the set of simultaneous equations $\frac{\partial l}{\partial \theta_i} = 0$

$$\frac{1}{2\sigma^2} \sum_n^N 2(x_n - \mu) = 0$$

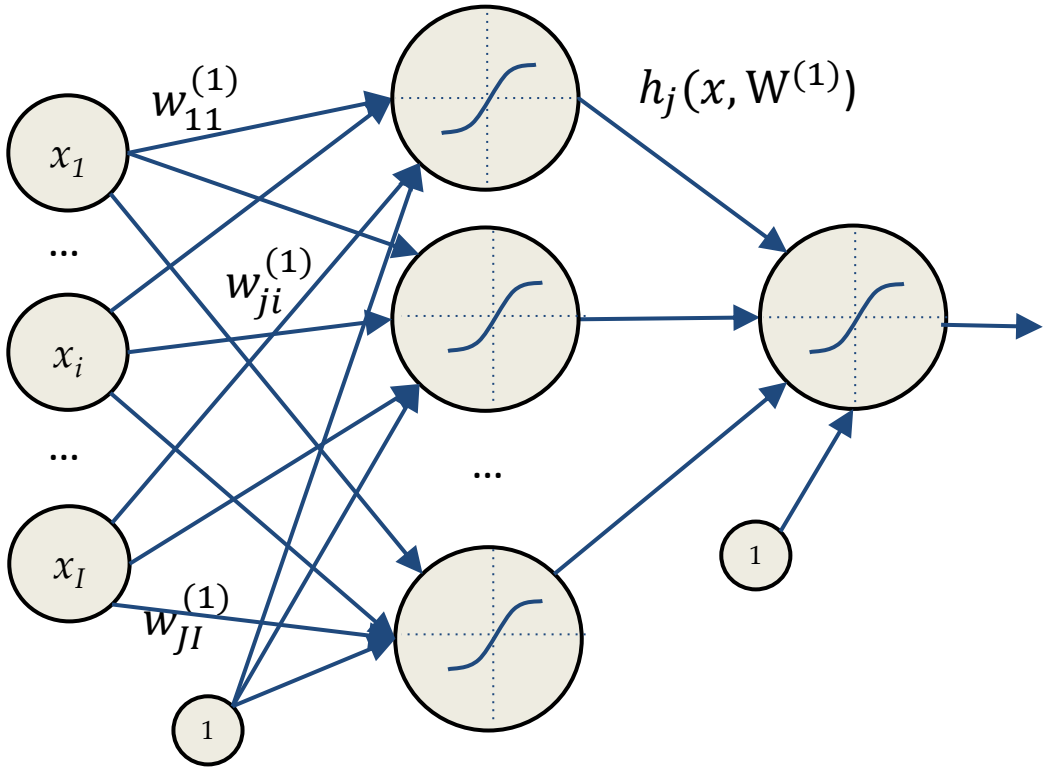
$$\sum_n^N (x_n - \mu) = 0$$

$$\sum_n^N x_n = \sum_n^N \mu \quad \Rightarrow \quad \boxed{\mu^{MLE} = \frac{1}{N} \sum_n^N x_n}$$

Let's apply this all to Neural Networks!

VERY NATURAL !

Neural Networks for Regression



$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

NN is a universal approximator so let's assume that $\exists$ a set of weights for which we can say that the target comes from the NN.

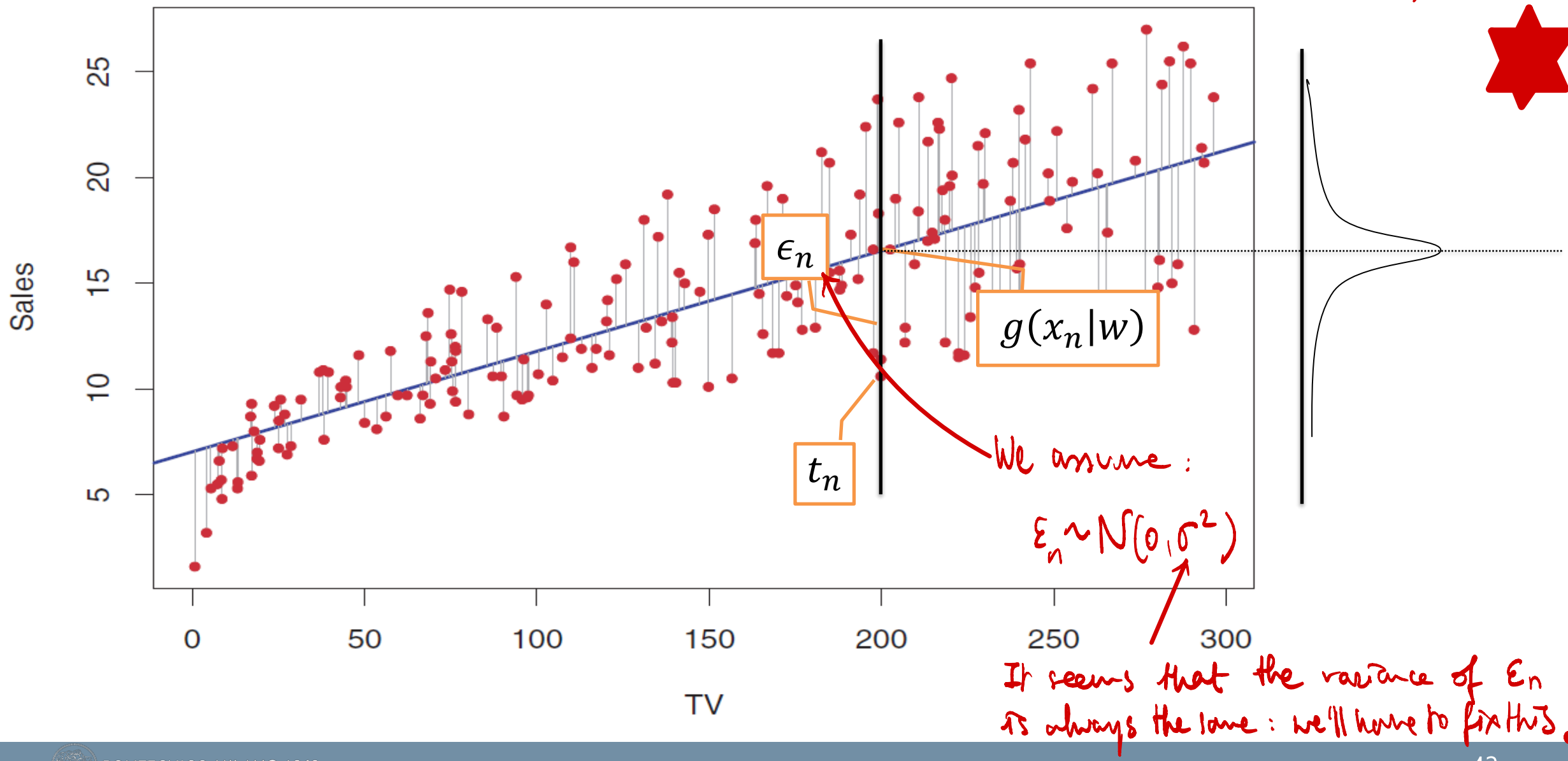
Goal: approximate a target function t having N observations

$$t_n = g(x_n|w) + \epsilon_n, \qquad \epsilon_n \sim N(0, \sigma^2)$$

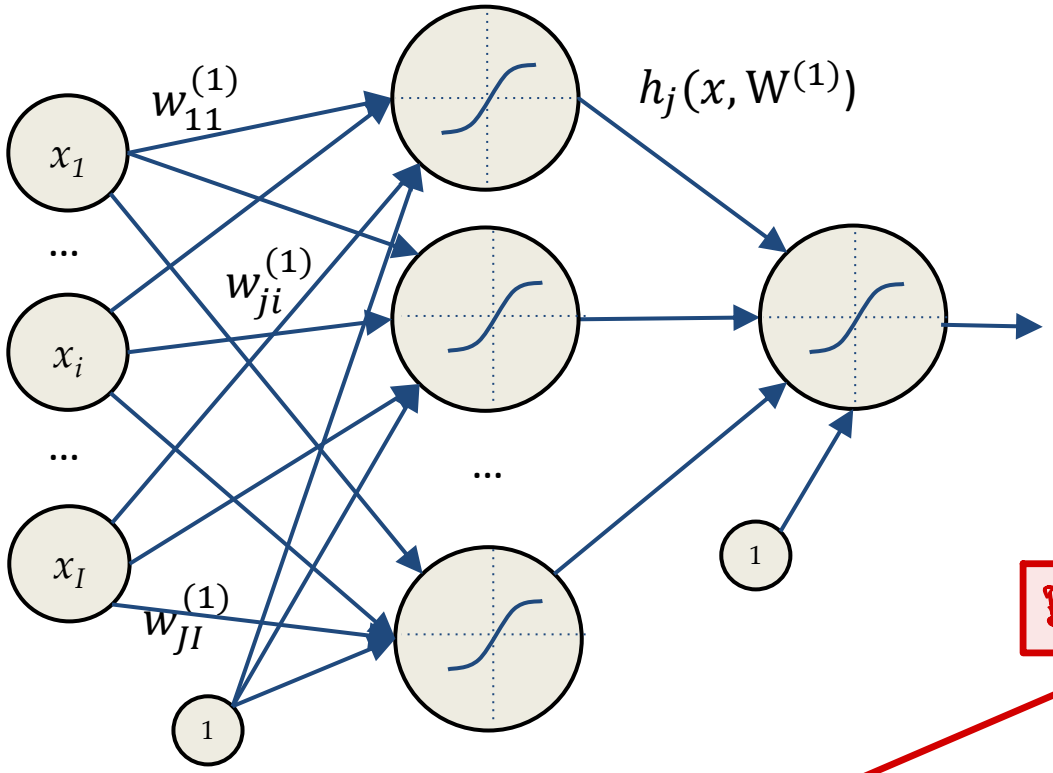
↓ However, we assume that these observations come from the NN but have been corrupted by some noise

Statistical Learnig Framework

Here: observations = • : we have noise!
target = ——— : f(x) I want to approximate.



Neural Networks for Regression

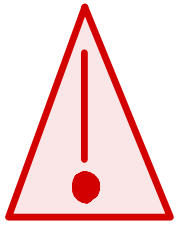


$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

DETERMINISTIC → So:

Goal: approximate a target function t having N observations

$$t_n = g(x_n|w) + \epsilon_n, \qquad \epsilon_n \sim N(0, \sigma^2) \quad \Rightarrow \quad t_n \sim N(g(x_n|w), \sigma^2)$$



so let's find the weights of my NN such that the likelihood of my data is MAXIMIZED!

Maximum Likelihood Estimation for Regression

AS WE DID IN A PREVIOUS EXAMPLE (cf above):

We have i.i.d. samples coming from a Gaussian with known σ^2

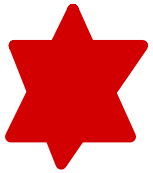
$$t_n \sim N(g(x_n|w), \sigma^2) \qquad p(t|g(x|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-g(x|w))^2}{2\sigma^2}}$$

Write the likelihood $L = P(Data|\theta)$ for the data

$$L(w) = p(t_1, t_2, \dots, t_N|g(x|w), \sigma^2) = \prod_{n=1}^{N} p(t_n|g(x_n|w), \sigma^2) =$$

$$= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}}$$

Maximum Likelihood Estimation for Regression



We have i.i.d. samples coming from a Gaussian with known σ^2

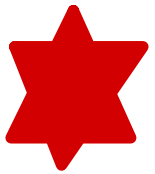
$$t_n \sim N(g(x_n|w), \sigma^2) \qquad p(t|g(x|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-g(x|w))^2}{2\sigma^2}}$$

Write the loglikelihood $\mathbf{l = \log P(Data|\theta)}$ for the data

$$l(w) = log \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}} = \sum_{n}^{N} \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}}\right)$$

$$= \sum_{n}^{N} \log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}(t_n - g(x_n|w))^2$$

Maximum Likelihood Estimation for Regression



We have i.i.d. samples coming from a Gaussian with known σ^2

$$t_n \sim N(g(x_n|w), \sigma^2) \qquad p(t|g(x|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-g(x|w))^2}{2\sigma^2}}$$

Look for the weights which maximixe the loglikelihood

doesn't depend on the weights.

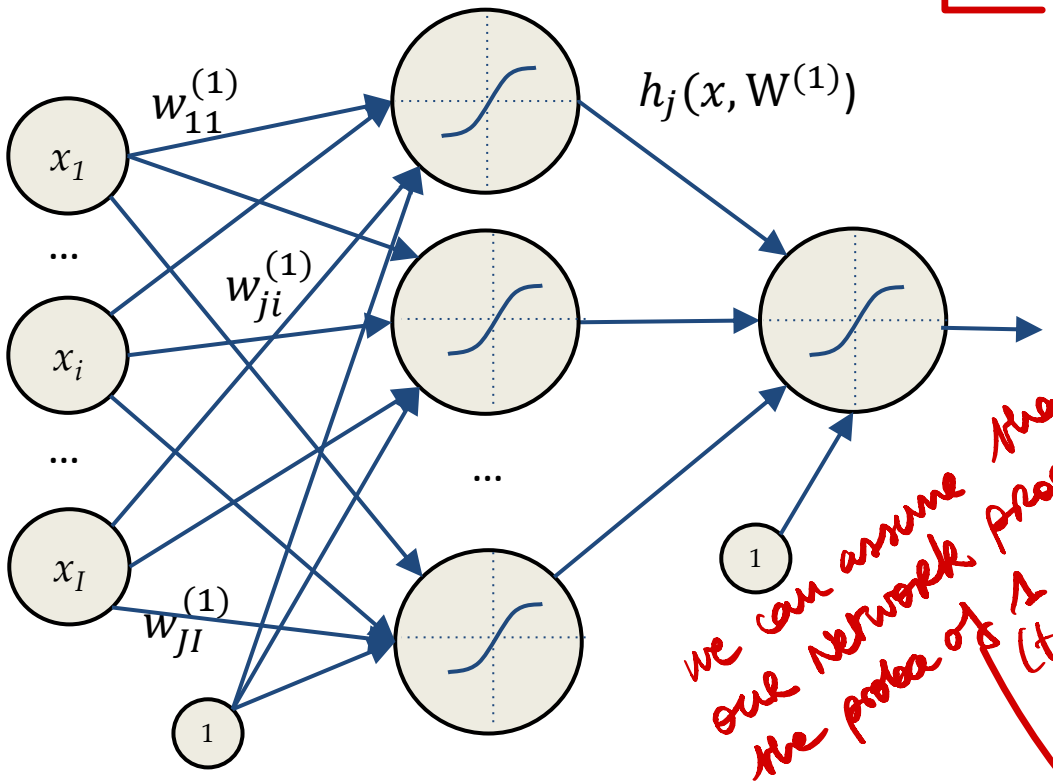
$$argmax_w\, l(w) = argmax_w \sum_n^N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}\left(t_n - g(x_n|w)\right)^2 =$$

$$= argmin_w \sum_n^N \left(t_n - g(x_n|w)\right)^2$$

finding the weights of a NN under assumption of additive Gaussian noise w/ 0 mean and cst variance $\Leftrightarrow$ minimizing the sum of squares.

Neural Networks for Classification

CLASSIFICATION this time!



$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

we can assume that our network produces the probability of 1 : binary data (t_n)

We are trying to learn the "a posteriori probability" over the classes

Goal: approximate a posterior probability t having N observations

$$g(x_n|w) = p(t_n|x_n), \quad t_n \in \{0, 1\}$$

$$t_n \sim Be\big(g(x_n|w)\big)$$

Let's assume that my NN is producing the "a posteriori probability" over the classes

↳ the output = the a posteriori probability.

Maximum Likelihood Estimation for Classification

We estimate the weights once again with MLE

In classification, note = 0 instead of 1 (& vice versa).

We have some i.i.d. samples coming from a Bernulli distribution

$$t_n \sim Be\big(g(x_n|w)\big) \qquad p\big(t|g(x|w)\big) = g(x|w)^t \cdot \big(1 - g(x|w)\big)^{1-t}$$

proba of $t=1$: $g(x|w)$
$t=0$: $1-g(x|w)$

we can write a $B(.)$ like this (try with $t=0, t=1$).

t = output

Write the likelihood $L = P(Data|\theta)$ for the data

$$L(w) = p\big(t_1, t_2, \dots, t_N|g(x|w)\big) = \prod_{n=1}^{N} p\big(t_n|g(x_n|w)\big) =$$

$$= \prod_{n=1}^{N} g(x_n|w)^{t_n} \cdot \big(1 - g(x_n|w)\big)^{1-t_n}$$

Maximum Likelihood Estimation for Classification

We have some i.i.d. samples coming from a Bernulli distribution

$$t_n \sim Be\big(g(x_n|w)\big) \qquad p\big(t|g(x|w)\big) = g(x|w)^t \cdot \big(1 - g(x|w)\big)^{1-t}$$

Compute the log likelihood $l = logP(Data|\theta)$ for the data

$$l(w) = \log \prod_{n=1}^{N} g(x_n|w)^{t_n} \cdot \big(1 - g(x_n|w)\big)^{1-t_n}$$

$$= \sum_{n}^{N} t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

Maximum Likelihood Estimation for Classification

We have some i.i.d. samples coming from a Bernulli distribution

$$t_n \sim Be\big(g(x_n|w)\big) \qquad p\big(t|g(x|w)\big) = g(x|w)^t \cdot \big(1 - g(x|w)\big)^{1-t}$$

This is the MLE for the parameters of the NN when the output can be described as a be distribution with a parameter which is the a-posteriori probability predicted by the network.

Look for the weights which maximize the loglikelihood

$$argmax_w\, l(w) = argmax_w \sum_n^N t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

$$= argmin_w - \sum_n^N t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$$

can be vectors

Crossentropy

$-\sum_n^N t_n^T \log g(x_n|w)$

Loss function that we SHOULD use in classification! ≠ from Σ squares error!

What about perceptron

How to Chose the Error Function?

Sum of Squared Errors

We have observed different error functions so far

Regression: $E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$

Binary Crossentropy

binary classification: $E(w) = -\sum_{n}^{N} t_n \log g(x_n|w) + (1 - t_n) \log(1 - g(x_n|w))$

in our regression example, the data violate our assumption of constant std.

Error functions define the task to be solved, but how to design them?

- Use all your knowledge/assumptions about the data distribution
- Exploit background knowledge on the task and the model → ex: SVM: linearly separate data.
- Use your creativity!

This requires lots of trial and errors ...

As for the Perceptron ...

Hyperplanes Linear Algebra

Let consider the hyperplane (affine set) $L \in \mathfrak{R}^2$

$$L: w_0 + w^T \mathrm{x} = 0$$

Any two points $\mathbf{x_1}$ and $\mathbf{x_2}$ on $L \in \mathfrak{R}^2$ have

$$w^T(\mathrm{x}_1 - \mathrm{x}_2) = 0$$

The versor normal to $L \in \mathfrak{R}^2$ is then

$$w^* = w/\|w\|$$

norm = 1

It gives the direction $\perp L$.

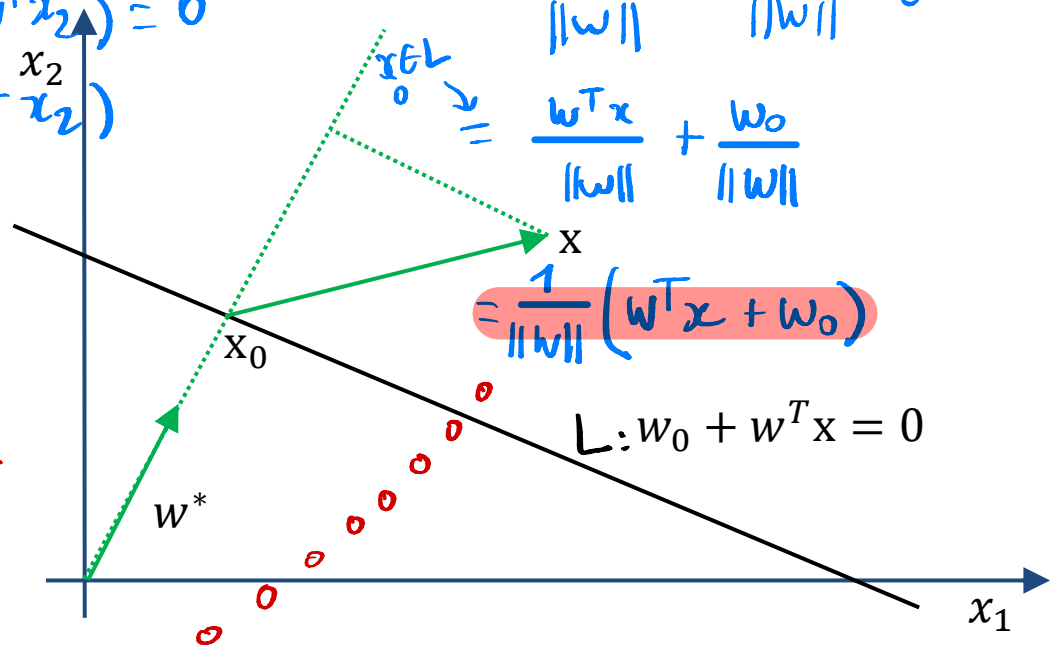
For any point $\mathbf{x_0}$ in $L \in \mathfrak{R}^2$ we have

$$w^T \mathrm{x}_0 = -w_0$$

The signed distance of any point $\mathbf{x}$ from $L \in$

$$w^{*T}(\mathrm{x} - \mathrm{x}_0) = \frac{1}{\|w\|}(w^T \mathrm{x} + w_0)$$

any point on the line.



$(w^T \mathrm{x} + w_0)$ *is proportional to the distance of* x *from the plane defined by* $(w^T \mathrm{x} + w_0) = 0$

if x above L: $w^T x + w_0 > 0$

below L: $w^T x + w < 0$

Perceptron Learning Algorithm (1/2)

It can be shown, the error function the Hebbian rule is minimizing is the distance of misclassified points from the decision boundary.

Let's code the perceptron output as +1/-1

- If an output which should be +1 is misclassified then $\mathbf{w}^{\mathrm{T}}\mathbf{x} + \mathbf{w}_0 < 0$
- For an output with -1 we have the opposite

The goal becomes minimizing

$$D(w, w_0) = -\sum_{i \in \mathrm{M}} t_i(\mathbf{w}^{\mathrm{T}}\mathbf{x}_\mathbf{i} + \mathbf{w}_0)$$

Set of points misclassified

(with $-$: becomes positive)

$< 0 \rightarrow$ error

So by minimizing $D \rightarrow$ we reach the right solution.

This is non negative and proportional to the distance of the misclassified points from $\mathbf{w}^{\mathrm{T}}\mathbf{x} + \mathbf{w}_0 = \mathbf{0}$

Perceptron Learning Algorithm (2/2)

Let's minimize by stochastic gradient descend the error function

$$D(w, w_0) = -\sum_{i \in \mathrm{M}} t_i (\mathrm{w}^{\mathrm{T}} \mathrm{x_i} + \mathrm{w_0})$$

The gradients with respect to the model parameters are

$$\frac{\partial D(w, w_0)}{\partial w} = -\sum_{i \in \mathrm{M}} t_i \cdot \mathrm{x_i} \qquad \frac{\partial D(w, w_0)}{\partial w_0} = -\sum_{i \in \mathrm{M}} t_i$$

Stochastic gradient descent applies for each misclassified point

$$\begin{pmatrix} w^{k+1} \\ w_0^{k+1} \end{pmatrix} = \begin{pmatrix} w^k \\ w_0^k \end{pmatrix} + \eta \begin{pmatrix} t_i \cdot x_i \\ t_i \end{pmatrix} = \begin{pmatrix} w^k \\ w_0^k \end{pmatrix} + \eta \begin{pmatrix} t_i \cdot x_i \\ t_i \cdot x_0 \end{pmatrix}$$

Hebbian learning implements Stocastic Gradient Descent

END OF LECTURE 4