POLITECNICO MILANO 1863

# Artificial Neural Networks and Deep Learning

## - Machine Learning vs Deep Learning-

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

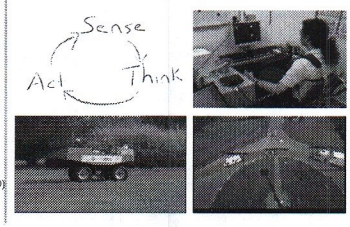
AIRLAB

## «Me, Myself, and I»

Matteo Matteucci, PhD
Associate Professor
Dept. of Electronics, Information & Bioengineering
Politecnico di Milano
**matteo.matteucci@polimi.it**

My research interests
- Robotics & Autonomous Systems
- Machine Learning
- Pattern Recognition
- Computer Vision & Perception

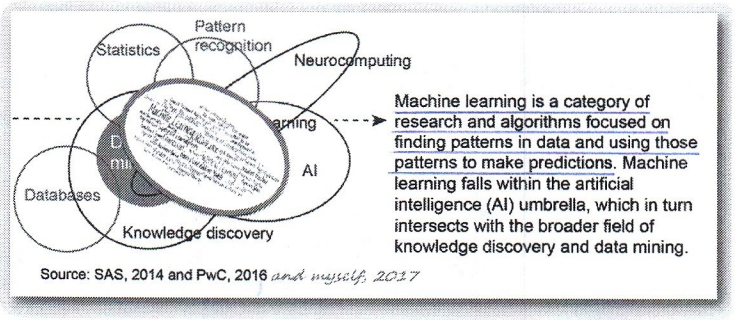
Courses I teach
- Robotics (BS+MS)
- Machine Learning (MS)
- Deep Learning (MS+PhD)
- Cognitive Robotics (MS)




***Enable physical and software autonomous systems to perceive, plan, and act without human intervention in the real world***



## Machine Learning

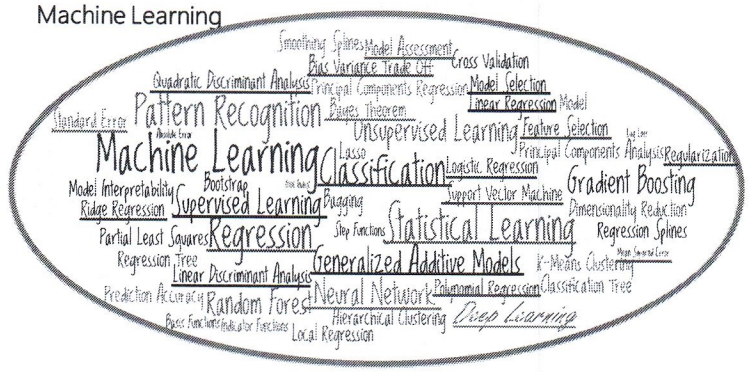



Machine learning is a category of research and algorithms focused on finding patterns in data and using those patterns to make predictions. Machine learning falls within the artificial intelligence (AI) umbrella, which in turn intersects with the broader field of knowledge discovery and data mining.

Source: SAS, 2014 and PwC, 2016 *and myself, 2017*



## Machine Learning



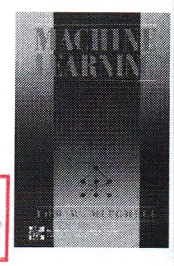


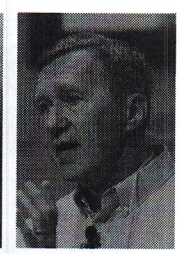

## Machine Learning (Tom Mitchell – 1997)

**Notice**: knowing the task is fundamental since the tools we can use depend on the task!

*T = Regression/Classification/...*
*E = Data*
*P = Errors/Loss*

*"A computer program is said to learn from experience E with respect to some class of task T and a performance measure P, if its performance at tasks in T, as measured by P, improves because of experience E."*



## Machine Learning Paradigms

Big classes of problems that we face with ML :

Immagine you have a certain experience E, i.e., data, and let's name it

$$D = x_1, x_2, x_3, \dots, x_N$$

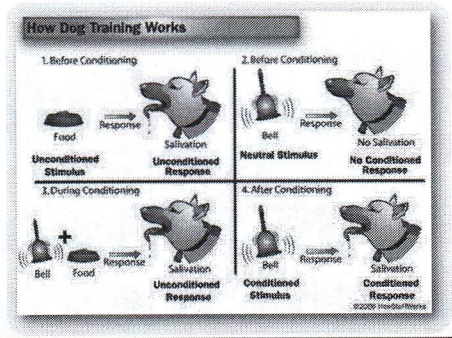
- *Supervised learning*: given the desired outputs $t_1, t_2, t_3, \dots, t_N$ learn to produce the correct output given a new set of input
- *Unsupervised learning*: exploit regularities in $D$ to build a representation to be used for reasoning or prediction
- *Reinforcement learning*: producing actions $a_1, a_2, a_3, \dots, a_N$ which affect the environment, and receiving rewards $r_1, r_2, r_3, \dots, r_N$ learn to act in order to maximize rewards in the long term

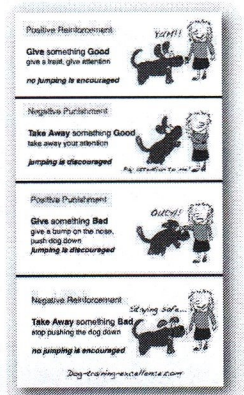
we identify clusters among data looking at similarities among data

the output in this case is a model that can be used to predict the output given a new set of inputs. (! the output is a model, not an algorithm)



## Reinforcement Learning is Wellknown



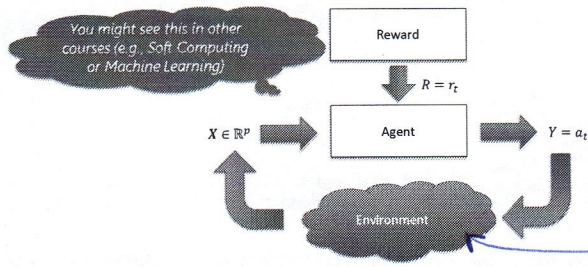






Suppose we take a puppy home and we want him to learn that he has to poop outside. This is not supervised learning (he won't learn from examples) nor unsupervised learning (we can't let him poop everywhere and hope eventually he'll choose the outside). The right way is reinforcement learning: the dog is free to choose among some actions and he receives from us (the environment) some awards or punishments and based on this he'll learn.

## Reinforcement Learning

Let's our machine be an agent interacting with an unknown environment




one supervisor gives a weak feedback to the system



## Machine Learning Paradigms

Immagine you have a certain experience E, i.e., data, and let's name it

$$D = x_1, x_2, x_3, \dots, x_N$$

there are mostly 2 groups:
- CLASSIFICATION
- REGRESSION

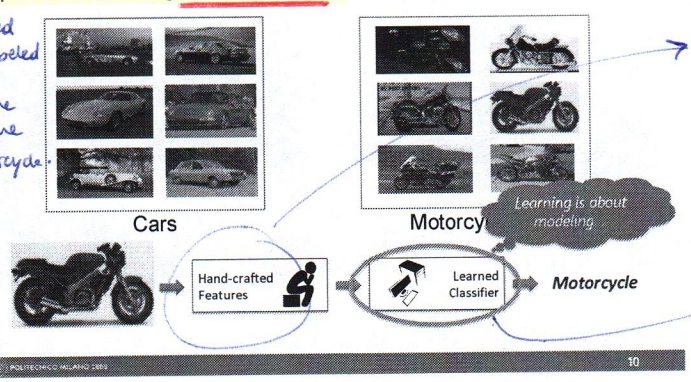
- *Supervised learning*: given the desired outputs $t_1, t_2, t_3, \dots, t_N$ learn to produce the correct output given a new set of input
- *Unsupervised learning*: exploit regularities in $D$ to build a representation to be used for reasoning or prediction
- *Reinforcement learning*: producing actions $a_1, a_2, a_3, \dots, a_N$ which affect the environment, and receiving rewards $r_1, r_2, r_3, \dots, r_N$ learn to act in order to maximize rewards in the long term



## Supervised learning: Classification

We have some images labeled as cars and some images labeled as motorcycles. We want to learn how to tell whether the image that is proposed to the model is a car or a motorcycle.

The output are LABELS, not numbers.



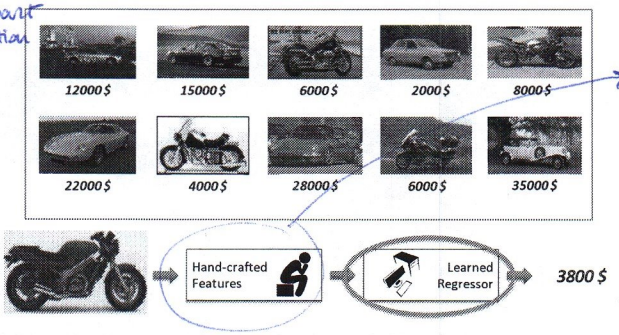

We extract some features: the shape, the contour,.. and we put them in a vector $\underline{x}$ (it is important that we're deciding what to extract)

model which is able to predict "car"/"motorcycle"

## Supervised learning: Regression

This is the continuous counterpart of classification: if classification developes a model which tells us which label is the right one among $\{1,..,n\}$, regression provides us a number.

Given a set of images of vehicles + prices we want a model able to predict prices




again from each image we extract features to put in $\underline{X}$



## Machine Learning Paradigms

Immagine you have a certain experience E, i.e., data, and let's name it

$$D = x_1, x_2, x_3, \dots, x_N$$

- *Supervised learning*: given the desired outputs $t_1, t_2, t_3, \dots, t_N$ learn to produce the correct output given a new set of input
- *Unsupervised learning*: exploit regularities in $D$ to build a representation to be used for reasoning or prediction
- *Reinforcement learning*: producing actions $a_1, a_2, a_3, \dots, a_N$ which affect the environment, and receiving rewards $r_1, r_2, r_3, \dots, r_N$ learn to act in order to maximize rewards in the long term



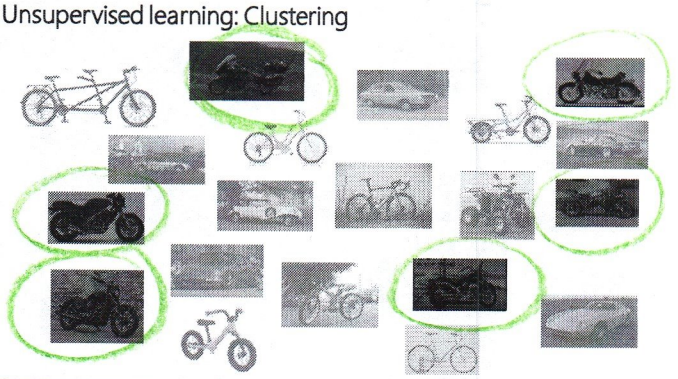
## Unsupervised learning: Clustering

We suppose to have images from both cars and motorcycles (without knowing which one is wich). We want to learn some clusters: even if there are no labbles we want to be able to say that a group of images is more similar among each other than with other images.

Based on similarities between items, we're classifing items into groups.

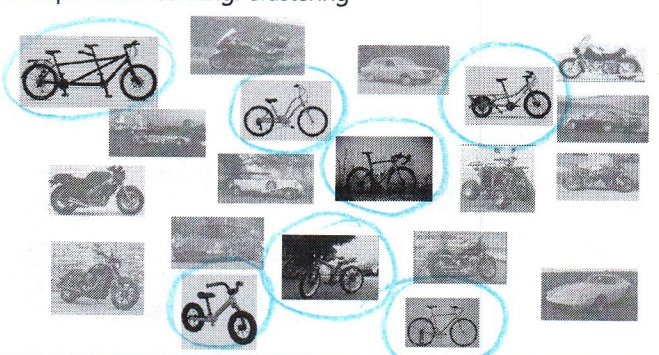


## Unsupervised learning: Clustering

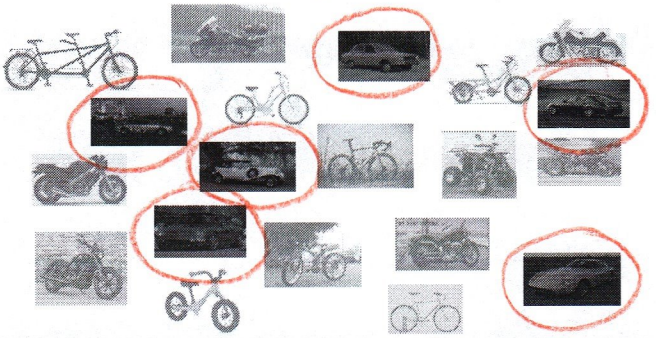




## Unsupervised learning: Clustering



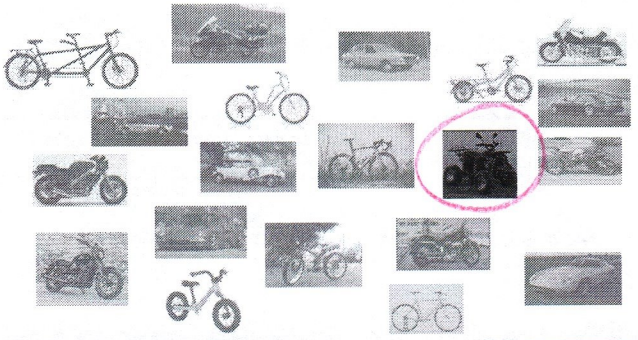
## Unsupervised learning: Clustering





We can even find an item that does not really fit the previous categories.

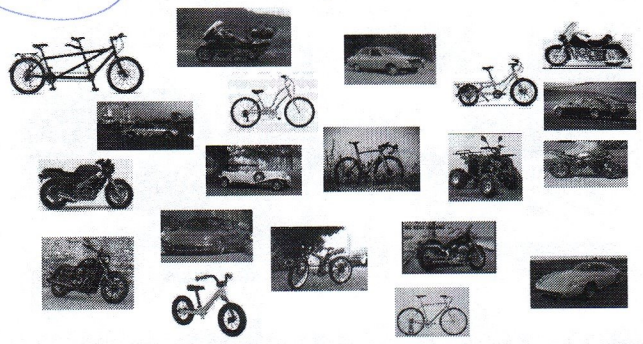
## Unsupervised learning: Clustering





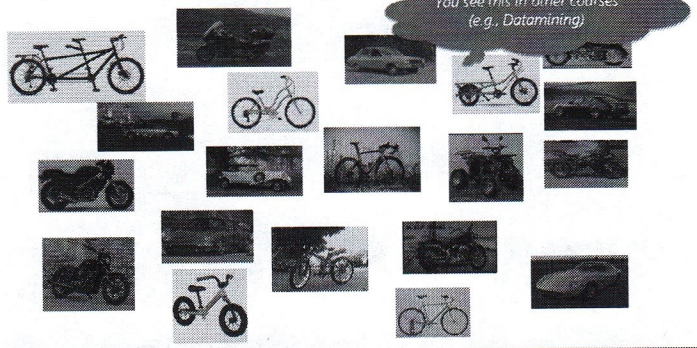
because we don't know the labels

## Unsupervised learning: Clustering





## Unsupervised learning: Clustering






## Machine Learning Paradigms

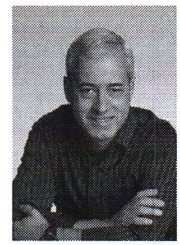
Immagine you have a certain experience E, i.e., data, and let's name it

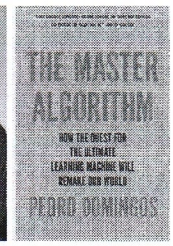
$$D = x_1, x_2, x_3, \dots, x_N$$

- *Supervised learning*: given the desired outputs $t_1, t_2, t_3, \dots, t_N$ learn to produce the correct output given a new set of input
- *Unsupervised learning*: exploit regularities in $D$ to build a representation to be used for reasoning or prediction
- *Reinforcement learning*: producing actions $a_1, a_2, a_3, \dots, a_N$ which affect the environment, and receiving rewards [illegible] to maximize rewards in the long term

This course focuses most on Supervised Learning (with some unsupervised spots)

## The Master Algorithm (Pedro Domingos, 2015)






*"The master algorithm is the ultimate learning algorithm. It's an algorithm that can learn anything from data and it's the holy grail of machine learning ..."*

But we know that such algorithm cannot exist (thm. "No free lunch")
The best algorithm depends on the data.



## The Master Algorithm (Pedro Domingos, 2015)

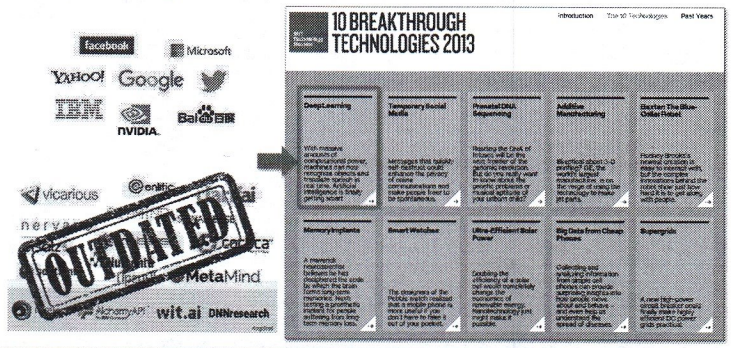
Pedros tried to classify ML tasks in some categories:
for every category he pointed out the best algorithms

| Symbolists | Bayesians | Connectionists | Evolutionaries | Analogizers |
|---|---|---|---|---|
| Animals / Mammals Birds | Likelihood \| Prior / Posterior \| Margin | Cell body / Synapse | | |
| Use symbols, rules, and logic to represent knowledge and draw logical inference | Assess the likelihood of occurrence for probabilistic inference | Recognize and generalize patterns dynamically with matrices of probabilistic, weighted neurons | Generate variations and then assess the fitness of each for a given purpose | Optimize a function in light of constraints ("going as high as you can while staying on the road") |
| **Favored algorithm** Rules and decision trees | **Favored algorithm** Naive Bayes or Markov | **Favored algorithm** Neural networks | **Favored algorithm** Genetic programs | **Favored algorithm** Support vectors |

Source: Pedro Domingos, *The Master Algorithm*, 2015



## Deep Learning: The Master Algorithm?




So concretely what is Deep Learning? Something related to all of this but in a way completely different.
How can we implement it?
Neural Networks.



## What is Deep Learning after all?

... let's say it with flowers!

*Iris Setosa*

*Iris Virginica*

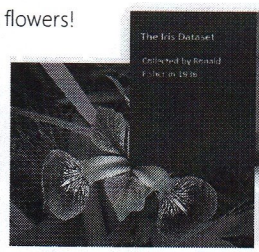
*Iris Versicolor*



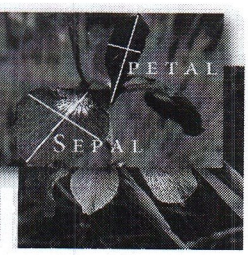
## What is Deep Learning after all?

... let's say it with flowers!

*Iris Setosa*




*Iris Virginica*




*Iris Versicolor*

This dataset collects some features of flowers (the features are related to the length and dept of sepal and petal). The goal is understand from which family a flower comes from only knowing these features. Moreover, which are the relevant characteristics to make the decision?

## What is Deep Learning after all?

(Machine Learning approach)

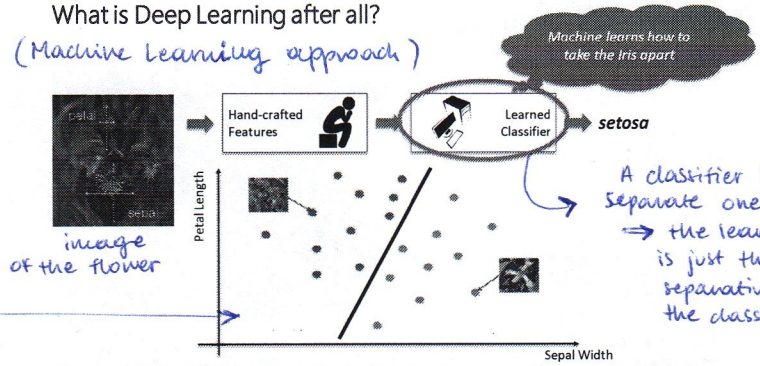


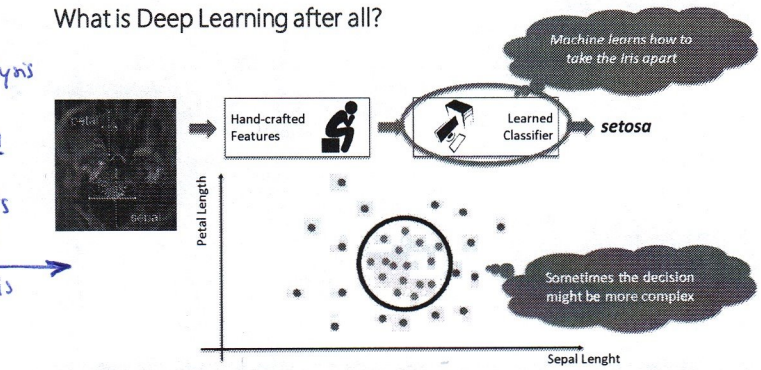

image of the flower

A classifier is just a criterion to separate one class from the other
⇒ the learning of a classifier is just the learning of a separating boundary between the classes

If we plot the data and we're lucky enough one family will be completely separated from the other one
⇒ it's enough to draw a line or a HYPERPLANE and we're done
(we can obtain it via linear discriminant analysis or logistic regression)



## What is Deep Learning after all?

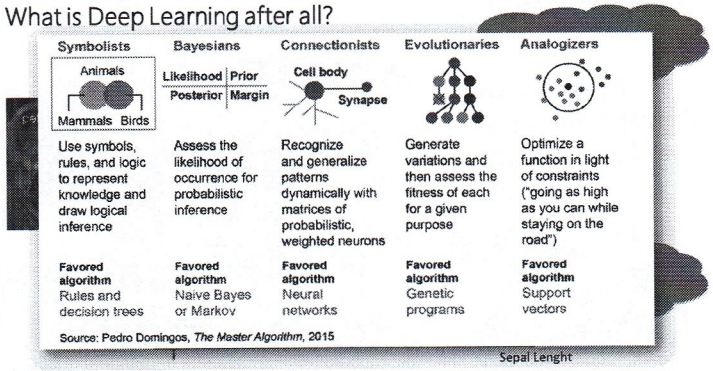



Sometimes it's not easy! We may have that the separating hyperplane is an non-linear function!

(but anyway the ML task is to separate the data)



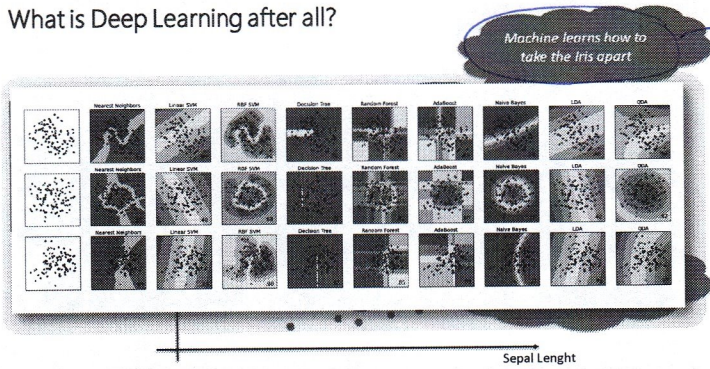
## What is Deep Learning after all?






## What is Deep Learning after all?

Some examples of which boundaries every algorithm can create:



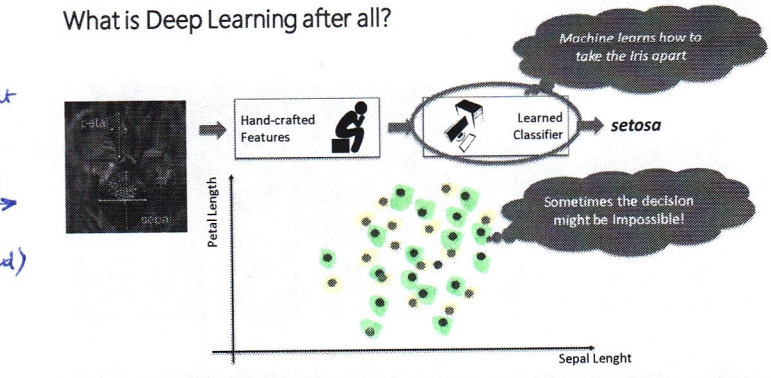

the work of ML is to find out an algorithm that can be able to draw the separating line every time that this line exists.

Problem: features engeneers must be good enough to select features in such way that this line exists. Selecting features we select the space where we'll have to find the separating line.

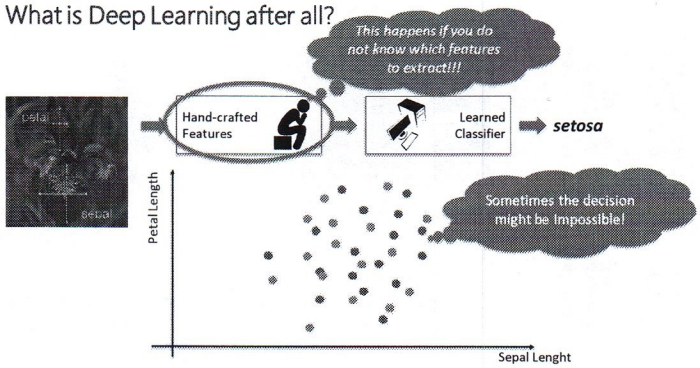


## What is Deep Learning after all?

If we select features that are totally uncorrelated with the class we may end up with something like this
(where the separating line is impossible to find)



## What is Deep Learning after all?



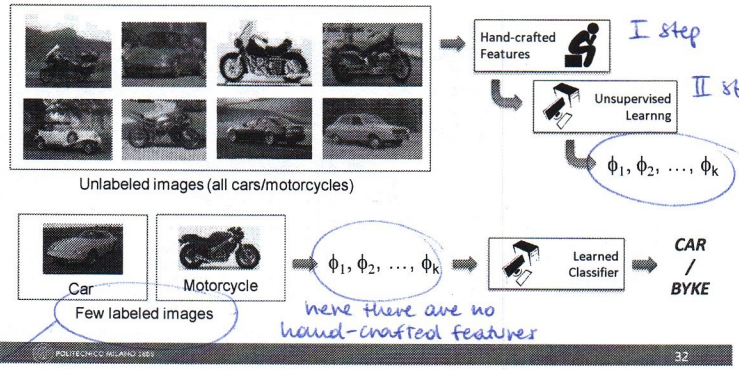



What if we want to improve on the dependence of the features selection? Then we'll have to work also on the feature collection. (We can even only work on the feature collection since if the date well behave the separation line is easy to find using ML). This idea was tried with the approach: "semi-supervised" learning

## Semi-supervised learning

Given a set of unlabeled data we try to extract hand-crafted features as generic as possible (so that we can adapt to any task) and we try to perform unsupervised learning, thanks to which we can extract the relevant features and those become the official features

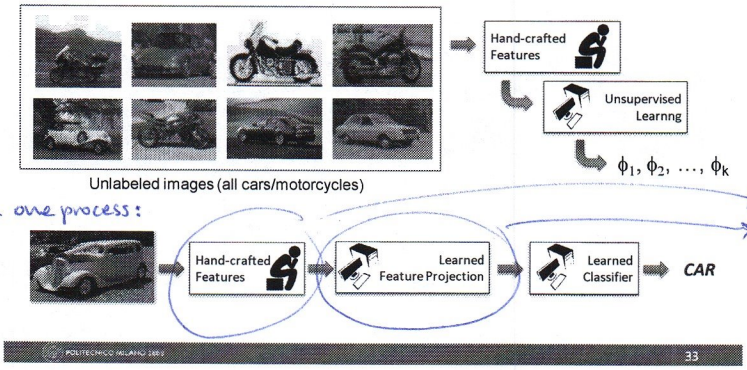
⟹ FEATURES EXTRACTION in two step process




here we need few labeled features (it's a good thing)



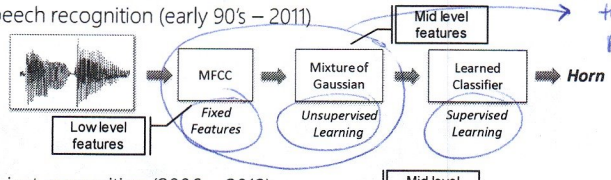
## Semi-supervised learning



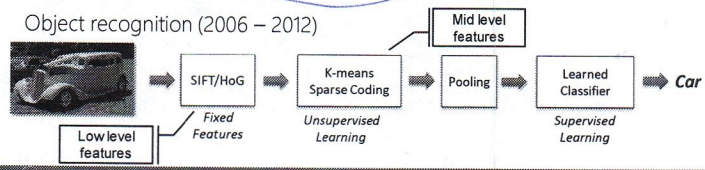



## Modern Pattern Recogniton (that works this way)

- Speech recognition (early 90's – 2011)

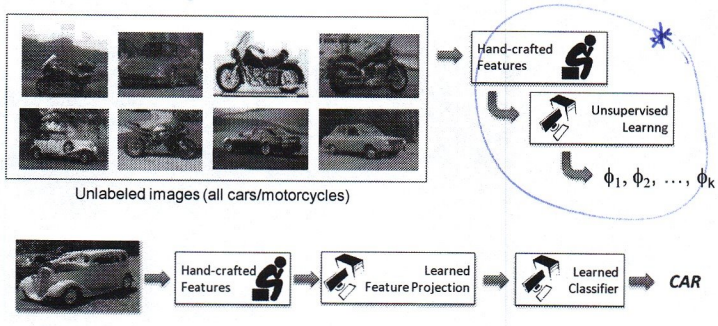



- Object recognition (2006 – 2012)






## Transfer Learning






In this way we can re-use the knowledge: we store knowledge while solving the problem and then we can use the stored knowledge with some other problems. (e.g. knowledge of recognizing cars can be applied to the recognization of trucks)

Notice that with an unsupervised task-oriented learning we're loosing something. Given the car/motorcycle data the step * leads to select features that talk about cars vs. motorcycle. What if we want to know which vehicle is pointing left/right? We would have to do the * again.
⟹ we procede with a general *, not task-oriented: TRANSFER LEARNING

Now suppose that, instead of using task-related features we use generic features, so features from other domains (for instance random images on the web) → This can lead to an improvement of the task.

(For example; if we want to do a text classification we may use words that belongs to non-related datasets)

## Transfer Learning = usage of the knowledge of data uncorrelated to the task for the improving

e.g. for the recognition of cars we can use the knowledge gained from the recognition of trucks

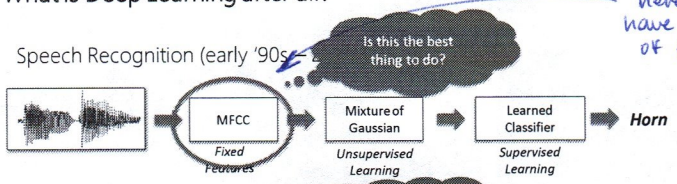


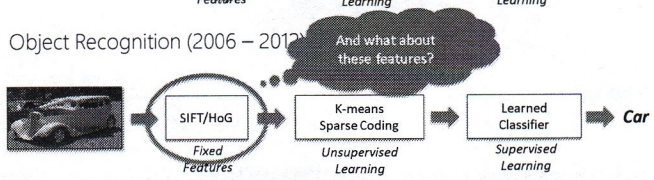





## What is Deep Learning after all?

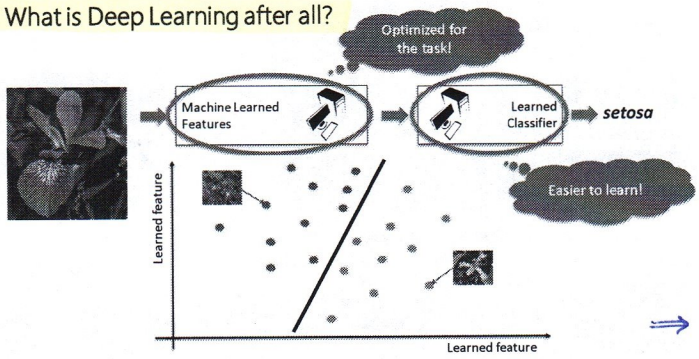
here we still have the hand-extraction of the features






## What is Deep Learning after all?



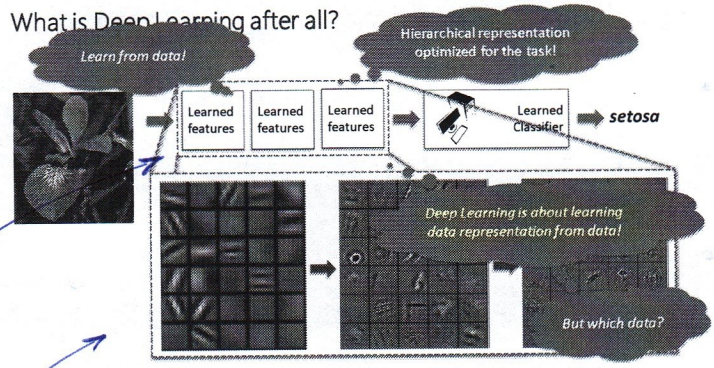

Deep learning takes care of the WHOLE features extraction and selection process (there is no "hand-extr.") There is NO FEATURES ENGENEERING.

⟹ DL asks the machine to learn the best representation of the data (and then we'll only need any good ML classifier) which means find the best space to put the points so that they can be easily separated.

How? We use a hierarchical representation; ←



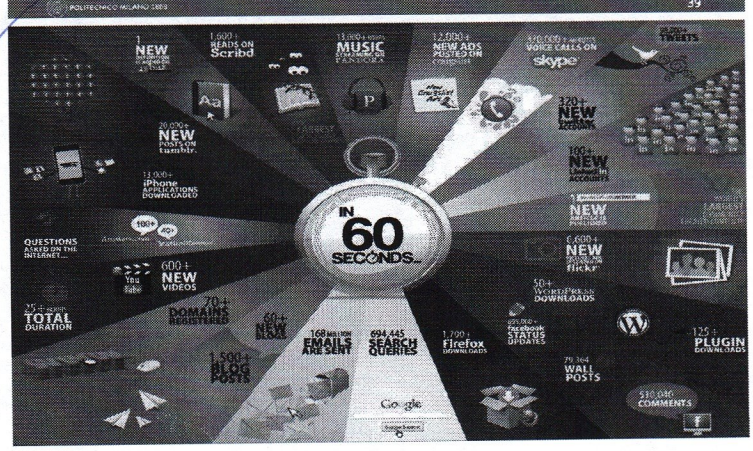
## What is Deep Learning after all?




because of such dependence of the data deep learning needs a lot! of data (a lot of LABELED data)

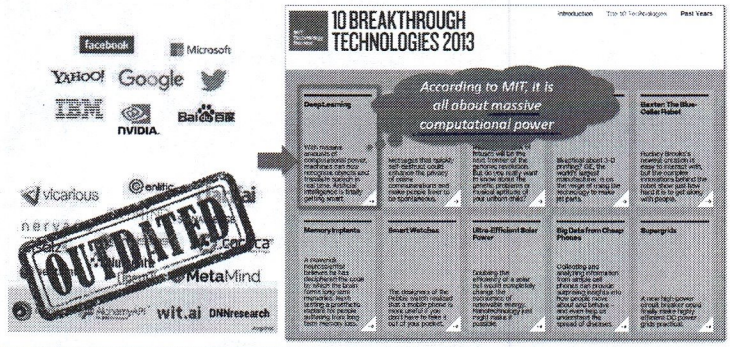
! another thing is: since we don't know which are the good features, DL finds them itself even with some combinations. This means that the ultimate features can be completely UN-INTERPRETABLE (the model is BLACK BOX)






" where these data comes from? everywhere "

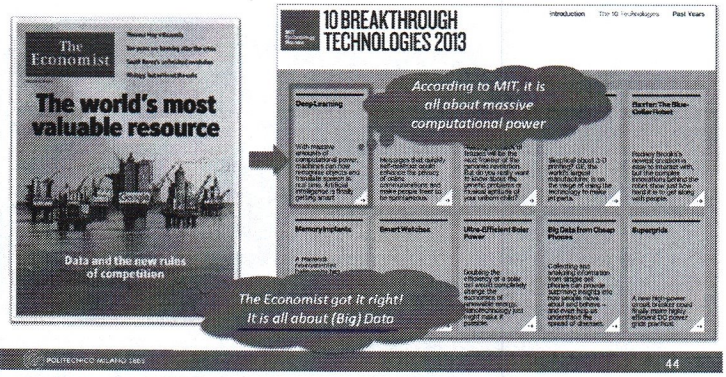
## What's behind Deep Learning?





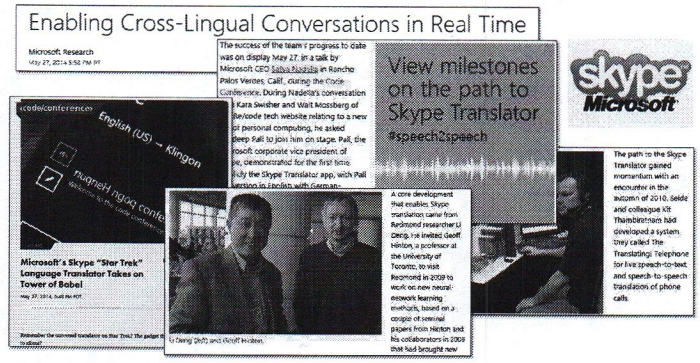

the problem of DL is not really the computational cost/power; is the huge amount of data needed !

## What's behind Deep Learning?






impressive results:






Note on DL: if we already know the feature space (optimal) is pointless to use DL! DL is very good in case of unstructured data (speach, text, images,..)

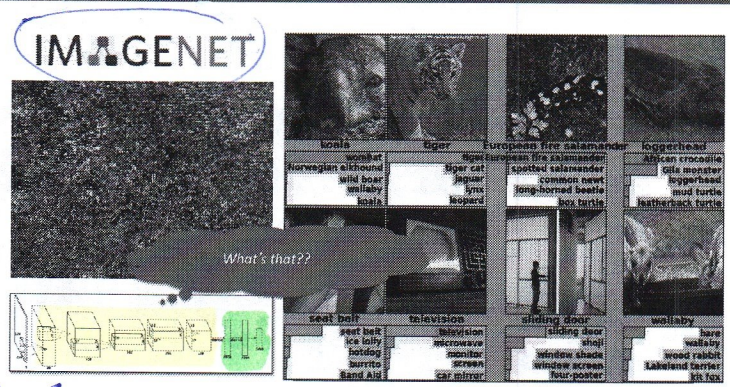
classification of images:
having an image the competition was about predict/classify the top 5 concepts

people tried with all the possible ML till came DL: it was a lot better than any ML attempt.

→ (neural network actually)

: feature extractor

: classifier






SEMANTIC SEGMENTATION
classification of pixels according to their class (road, car, human,..)

INSTANT SEGMENTATION :
not only we want to see that this is a person, but also that are 2 different persons (we want not only the generic class)

(+ SKELETON EXTRACTION)



STYLE TRANSFER :
match style and contents

https://github.com/luanfujun/deep-photo-styletransfer

https://github.com/jcjohnson/neural-style
https://github.com/jcjohnson/fast-neural-style
https://ml4a.github.io/ml4a/style_transfer/



INCREASING RESOLUTION :

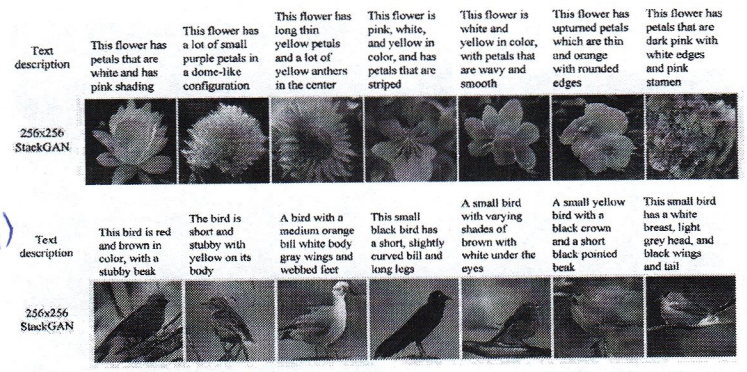
https://github.com/alexjc/neural-enhance



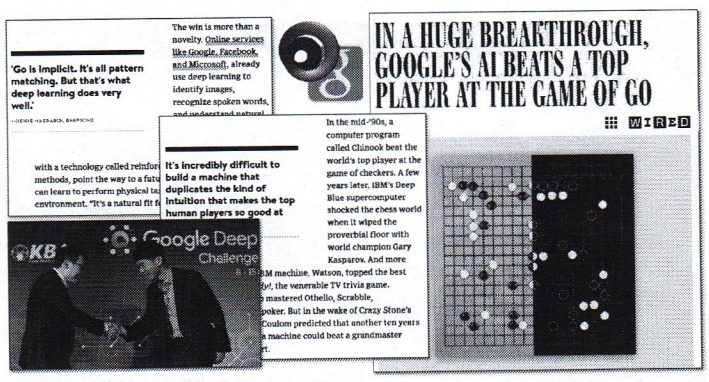

IMAGE GENERATION
images are generated based on the text

input →
output →

(similar to transfer learning)

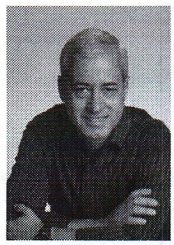


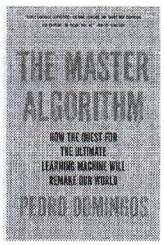


The Master Algorithm (Pedro Domingos, 2015)

"The master algorithm is the ultimate learning algorithm. It's an algorithm that can learn anything from data and it's the holy grail of machine learning ..."

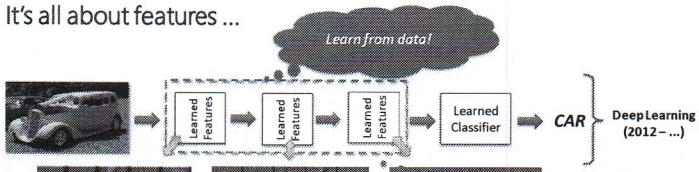


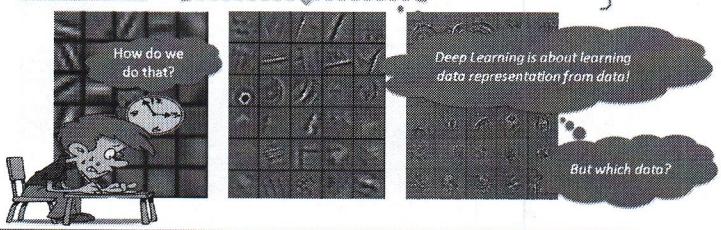



↳ But Deeplearning is not! We still have to face DL limits.



It's all about features ...



POLITECNICO MILANO 1863

# Artificial Neural Networks and Deep Learning

## - From Perceptrons to Feed Forward Neural Networks -

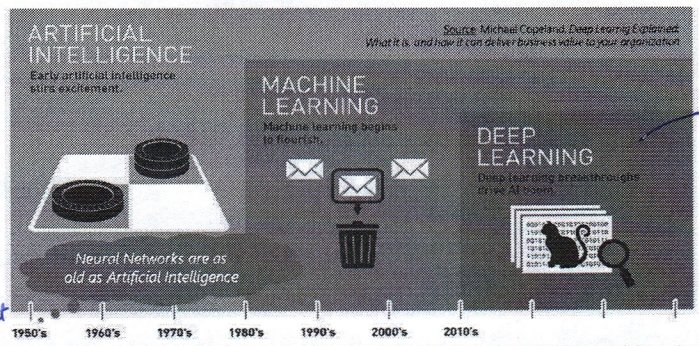
Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB

About names:
(Deep learning and Neural Nets)

"Deep" means that we pushed learning down in the hierarchy between raw data and classification through all the layers of abstraction (which are implemented by features extractor).
Then by chance we implement this with neural networks.
"By chance" because neural networks are much older than Deep Learning.

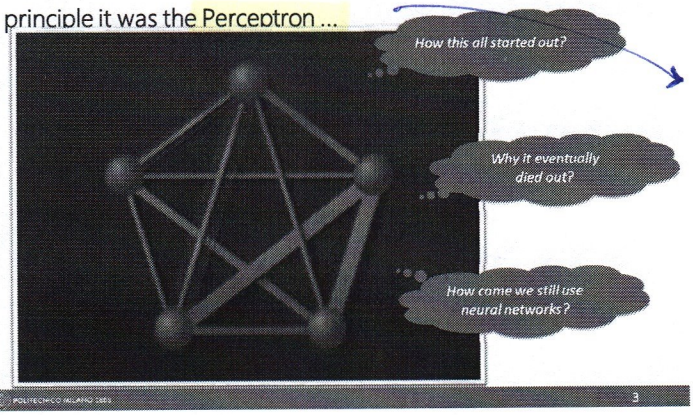
## «Deep Learning is not AI, nor Machine Learning»




models to learn data representation. They're more effective where we don't have an effective data representation, that is unstructured data (that's why DL is particularly effective with images, audio, speech, ..)



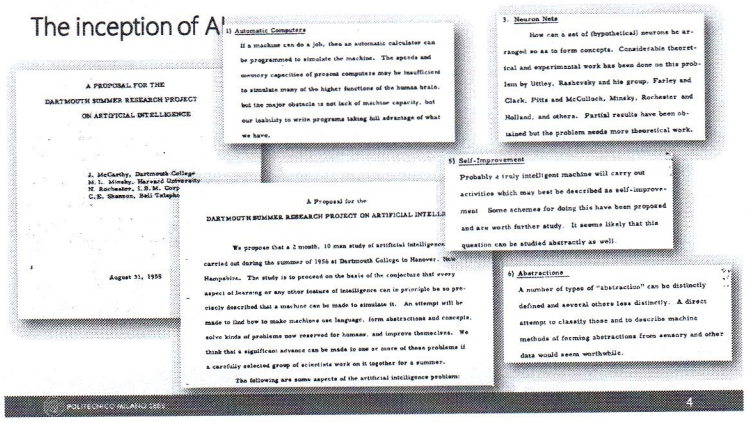
## In principle it was the Perceptron ...




First artificial neural network: they showed it a lot of examples and they told if it was wrong/correct as a classificator. In the end it learned.



## The inception of AI





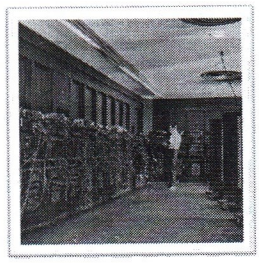

## Let's go back to 1940s ...

Computers were already good at

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

However we 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!

it was a model of a neuron. The idea was to put together many of these neurons and create something similar to the human brain.

How do the neuron works? The human neuron has a body and is connected to other neurons. How are the informations sent? How do the neurons communicate? The informations travel as charges: a charge moves from one neuron to another and it can increase or decrease the charge in the receiving neuron.

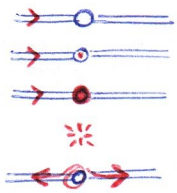
⇒ a human neuron is something that receive, send and accumulate charge

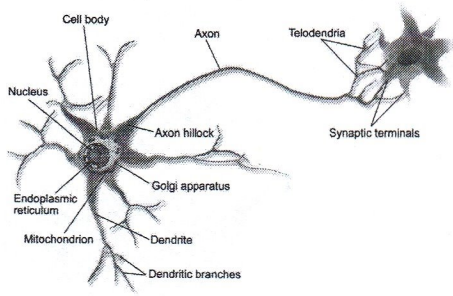


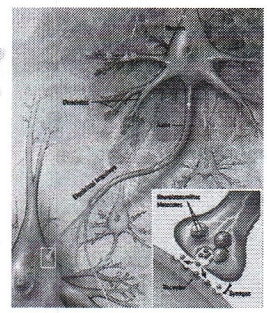
## Computation in Biological Neurons

* once the charge in a neuron exceed a certain treshold, the entire charge of the neuron discharge and reach other neurons

(charging → slow process
discharging → fast process)





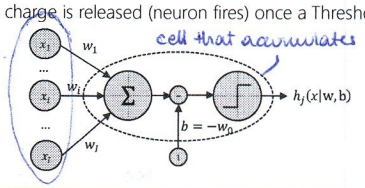


## Computation in Artificial Neurons

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

inputs signals (neurons connected to a single cell that give (weighted) signals to the cell. The signals are summed, that's why "$\Sigma$")




= information is exchanged in chemical charge. This charge is collected by a neuron, weighted by the synaptic weights. Once this accumulated charge exceed a treshold is realised in a single shot.

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

output of the neuron

Once inside the cell, the signal (charge) is compared to a BIAS (which is the treshold):
if charge > BIAS then we got a positive output (otherwise a negative one)
⇒ output ∈ {-1/+1}
(or even output ∈ {0,1})

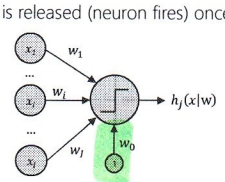
we sum the signals incoming ($x_i$) weighted ($w_i$) and we compare it with the bias ($b$). Then we apply the non-linear function $h(\cdot)$.



## Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

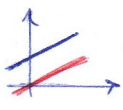
- Dendrites collect charges from synapses, both Inhibitory and Excitatory
- Cumulates charge is released (neuron fires) once a Threshold is passed




we can see the bias as a weight which weights an input which is always 1 ⇒ $x_0 = 1$, $w_0 = -b$

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

WHY BIAS?
It's like a linear regression without the constant term!



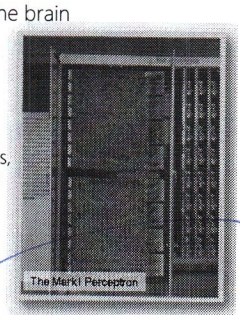
we'll never be able to learn this (—) line without the bias, the best probably will be —



## 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)

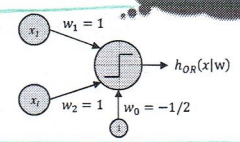



He introduced the bias as any other weight, so something that has to be trained.

## What can you do with it?

Truth tables:

| $x_0$ | $x_1$ | $x_2$ | OR |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

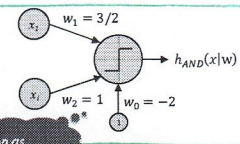



$$h_{OR}(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2) = h_{OR}\left(-\frac{1}{2} + x_1 + x_2\right) = \begin{cases} 1, & if\ \left(-\frac{1}{2} + x_1 + x_2\right) > 0 \\ 0, & otherwise \end{cases}$$

Perceptron as Logical OR

$x_0 = 1$ always

| $x_0$ | $x_1$ | $x_2$ | AND |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 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

This means: connecting 3 neurons that feed inputs $x_0 = 1$, $x_1 = 0$, $x_2 = 0$ with those weights $(-1/2, 1, 1)$ we get as an output a 0.

with the right weights we can obtain logical operators: OR, AND. Why are we interested? Because by putting enough neurons we can get a LOGIC NETWORK.

POLITECNICO MILANO 1863 11

## Hebbian Learning

= learning method that specifies how much the connection between to units (in our case this is $w_i$) should be increased/decreased in proportion to the product of their activation

*"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)

we have a synapsis and whenever it makes a mistake we correct the synaptic weight by modifying it a little bit. How do we modify it?

the power of the perceptron is in the fact that we learn the weights, not set them

Hebbian learning can be summarized by the following:

Start from a random initialization

$$w_i^{k+1} = w_i^k + \Delta w_i^k$$
$$\Delta w_i^k = \eta \cdot x_i^k \cdot t^k$$

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

Where we have:

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

A machine set some weights, try on labeled data if they work: if yes ok, if not the machine changes the weights and restart. If this procedure converges then we'll have a classifier.

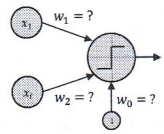
POLITECNICO MILANO 1863 12

## Perceptron Example (on white papers)

Life is easier with Sign(.)

Learn the weights to implement the OR operator

- Start from random weights, e.g., $w = [1\ 1\ 1]$
- 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




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

| $x_0$ | $x_1$ | $x_2$ | OR |
|---|---|---|---|
| 1 | 1 | -1 | +1 |
| 1 | 1 | 1 | +1 |
| 1 | -1 | -1 | -1 |
| 1 | -1 | 1 | +1 |

Does the procedure always converge? **No**
Does it always converge to the same sets of weights? **No**

POLITECNICO MILANO 1863 13

## Perceptron Math

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

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

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

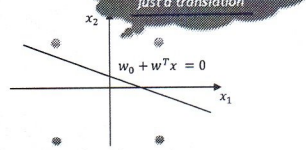
$$w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I = 0$$

that's why the set of points for which the perceptron is in doubt is the set of points for which this sum is 0.

With 0/1 input it is just a translation

In 2D, this turns into

$$w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$
$$w_2 \cdot x_2 = -w_0 - w_1 \cdot x_1$$
$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} \cdot x_1$$




The perceptron is looking for the separating hyperplane that puts all the points classified as "+1" in one region and all the "-1" in the other.

That's why, if we get $[w_0, w_1, .., w_I]$ that characterizes the hyperplane, also $c \cdot [w_0, w_1, .., w_I]$ characterizes the same hyperplane! (That's why we got $[1,1,1]$ and $[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ as optimal weights)

⇓

Does the Hebbian learning converge? Or it will cycle an infinite amount of time?
It depends on if it $\exists$ a separating hyperplane.

POLITECNICO MILANO 1863 14

## Boolean Operators are Linear Boundaries

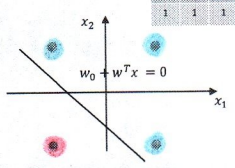
What's about it? We had already Boolean operators

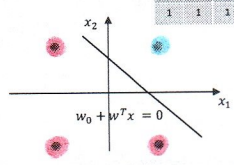
Linear boundary explains how Perceptron implements Boolean operators

| $x_0$ | $x_1$ | $x_2$ | OR |
|---|---|---|---|
| 1 | -1 | -1 | -1 |
| 1 | -1 | 1 | 1 |
| 1 | 1 | -1 | 1 |
| 1 | 1 | 1 | 1 |

Yes! But this is 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 |



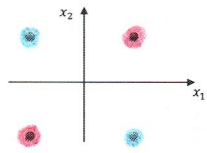




POLITECNICO MILANO 1863 15

## What can't you do with it?

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

| $x_0$ | $x_1$ | $x_2$ | 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.

Instead of changing the boundaries from linear to non linear we can change the representation of the inputs and make the inputs linearly separable.

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?

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

*Unfortunately Hebbian learning does not work any more ...*

*Multi Layer Perceptrons*

More layers we have and more complex boundaries we can make

The issue of this is that Hebbian learning is no more working! Why? Before we had: $x \xrightarrow{w} \bigcirc \rightarrow t$ and we changed the weight as: $w = w + \eta x t$ (we know all the elements). Now it is:

$x \xrightarrow{w_1} \bigcirc \xrightarrow{w_2} \bigcirc \rightarrow t$

$w_1' = w_1 + \eta x (\cdot)$ we don't have the desired target to update $w_1$

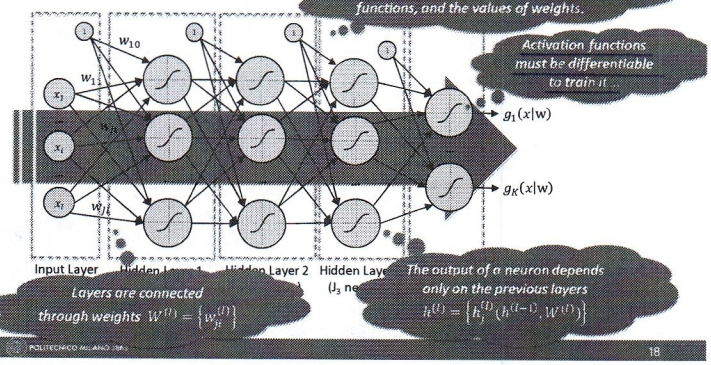
$w_2' = w_2 + \eta (\cdot) t$ we don't know the input for which we see $t$ (we only know the input for $w_1$)



(+)

## Feed Forward Neural Networks

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




*Activation functions must be differentiable to train it ...*

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

*The output of a neuron depends only on the previous layers* $h_i^{(l)} = \{h_i^{(l)}(h^{(l-1)}, W^{(l)})\}$

The way to have backpropagation working is by changing the structure of the neural network: we don't talk about multi layer perceptrons (since we don't want the step function to be the activation function). We change the activation function to a function more smooth. We now talk about ARTIFICIAL NEURONS (≠ perceptrons). and FEED FORWARD NEURAL NET. (≠ multi layer perceptrons).

The way (method) to solve this problem is **BACKPROPAGATION** (= being able to backpropagate the error so that to reach each of these weights). To do this we have to change something: we need differentiable activation functions ( ~~⟋~~ )

→ the biggest problem with the perceptron is the step function as activation function!

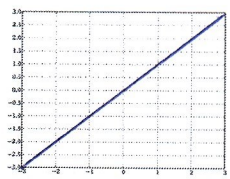


## Which Activation Function?

this function is the smooth version of the step function (continuous version)

this is the smooth (continuous) function of the sign function

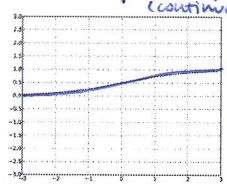
**Thm.** Why Sigmoid / Tanh? With them we can approximate any non-linear function.



Linear activation function

$g(a) = a$

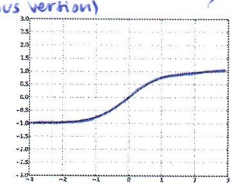
$g'(a) = 1$



Sigmoid activation function

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

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



Tanh activation function

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

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

we don't want to use only linear functions because we would have a linear classifier

To understand which one we should choose we have to start from the output: (the choice depends on the task)



## Output Layer in Regression and Classification

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

In Regression the output spans the whole $\Re$ domain:

- Use a Linear activation function for the output neuron

that's why it's preferred as the act. funct.s of the output

easy to interpret it as LOGISTIC REGRESSION

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)

among the classes only one is the true one and we "make it not" by marking it with the "1"

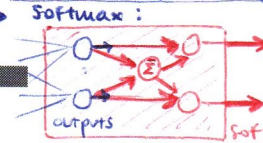
*«One hot» encoding*

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: $y_k = \frac{\exp(z_k)}{\sum_k \exp(z_k)} = \frac{\exp\left(\sum_j w_{kj} h_j \left(\sum_i^I w_{ji} \cdot x_i\right)\right)}{\sum_{k=1}^K \exp\left(\sum_j w_{kj} h_j \left(\sum_i^I w_{ji} \cdot x_i\right)\right)}$

we learn a model with 3 outputs, each one representing a class

Softmax :



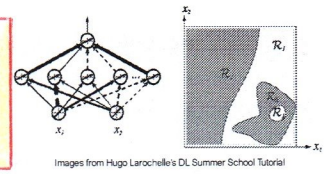

! If we do this way there is nothing to impose that Σ outputs = 1. So we normalize the output to have a probability interpretation and we can get the maximum of these. What usually is done is to take the maximum. Actually instead of the maximum it's used a special kind of normalization

$z_k$ = weighted sum

## Neural Networks are Universal Approximators

sigmoid or tanh

> *"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

any nonlinear function can be approximated with any arbitrary accuracy provided that we put enough neurons in the hidden layer.

Regardless the function we are learning, a single layer can represent 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 = it might overfit

Classification requires just one extra layer ... *

* This theorem talks about continuous function, i.e. for regression it's enough one layer, for classification are enough 2.
In practice instead, we see that having more layers (with less neurons) is more effective.

Why? We have to transform simple inputs in new space. If the transformation is very complex we prefer to learn it as a sequence of simple transformations.



(We assume to work with regression, but all of this is equally valid for classification)

## Optimization and Learning

described as a function of a set of parameters

Recall about learning a *parametric model* in regression and classification

- Given a training set (x input, t output)

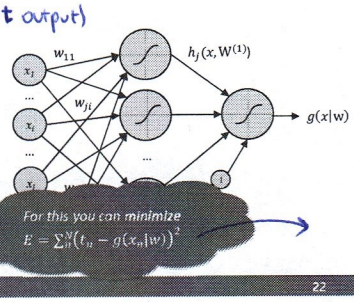
$$D = < x_1, t_1 > \cdots < x_N, t_N >$$

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

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

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

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



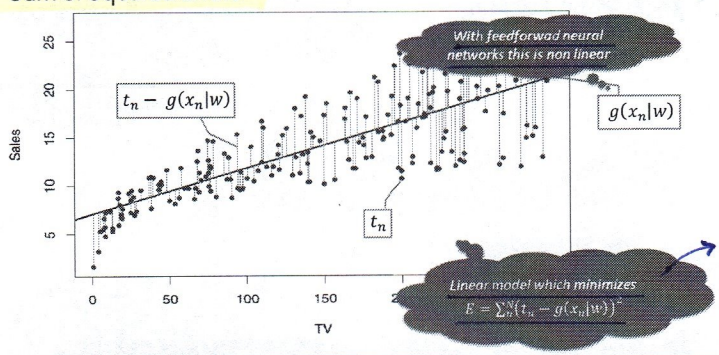

g is the nonlinear function implemented by the neural net.

We minimize:
$E = \sum_n^N (t_n - g(x_n|w))^2$

the problem is: find the right weights w such that $g(x_n|w)$ is "similar" to the output $t_n$.



## Sum of Squared Errors




Learning means that we have to find the minimum of the error function.

We're looking for the least squares fitting of the model.
(Here the model is a linear model, in general it's not)



## 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

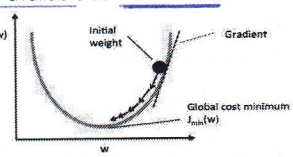
$$\frac{\partial J(w)}{\partial w} = 0$$

Closed-form solutions are practically never available so we can use iterative solutions:

- Initialize the weights to a random value
- Iterate until convergence

Gradient descent:

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




There are more precise methods but is it really worth it to use computationally heavy methods to find, probably, an overfitting global minimum?
Gradient descent is the best method for now.

The problem is that with neural networks the function to minimize is not that simple (convex, ...)
The error function is a highly non-linear function of the parameters.



## Gradient descent - Backpropagation

(Backpropagation is gradient descent)

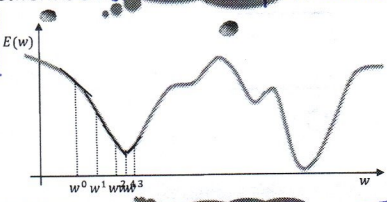
Finding the weighs of a Neural Network is a non linear optimization.

Use multiple reastarts to seek for a proper global minimum.

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

We iterate starting from an initial random configuration

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




To avoid local minima can use momentum

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

It depends on where we start from

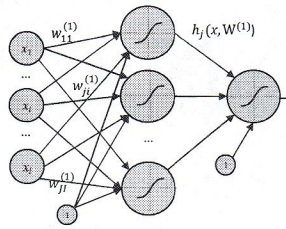
big problem

what are the problems of backpropagation? The same of gradient descent! Slow convergence, conv. to local minimum, different starting points, the gradient should be defined

that's why we need differentiable activation functions

the gradient $\left(\frac{\partial E}{\partial w}\right)$ describes the direction, $\eta$ describes the step (the length of the step)

## Gradient Descent Example



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

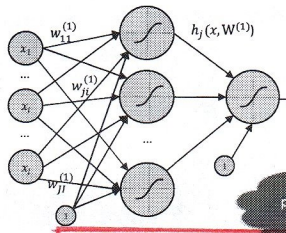
$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Use $j$=3 and $i$=5

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



## Gradient Descent Example



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

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

Using all the data points (batch) might be unpractical

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

it makes sense because if one data point pulls in one direction and an other data point pulls in the opposite if we operate one datum at the time it will probably oscillates. By doing a **BATCH UPDATE** we obtain a smoother optimization. However this might be unpractical.



## Gradient Descent Variations

= update the weights by considering all the derivatives w.r.t. all the data.

Can we do it? Yes, and it's called Stochastic gradient descent.

Problem: with SGD we get a lot of noise.

Optimal solution →

in batch GD we have $1/N$ because it takes all the data, SGD consider only 1 datum at the time

Batch gradient descent

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

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

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

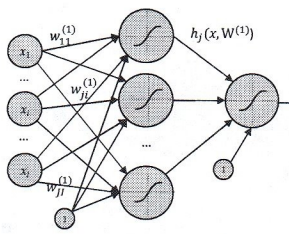
the problem is that this provide one datum at the time: ⇒ (noisy)

(at the end we obtain a solution equivalent to the batch grad. desc.)

bigger the batch, smoother the convergence.



## Gradient Descent Example



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

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

Can I make it automatic? → Yes.

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

Do we really have to compute this "by hand" every time?

No.



## Backpropagation and Chain Rule (1)

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 deivative of $f$ w.r.t. $x$ can be computed applying the chain rule

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

The same holds for backpropagation

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

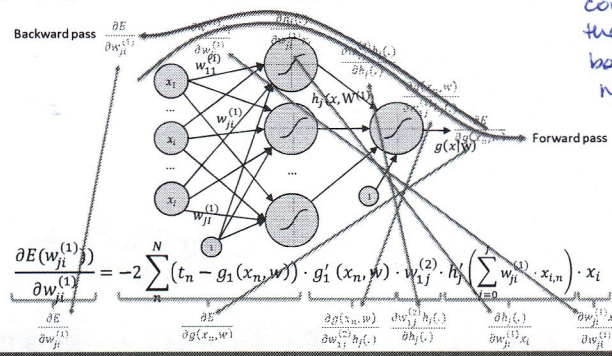
We now see that it can be splitted in multiple pieces, this however does not solve the problem of calculating $f$ = how do we extract the pieces).

How do we proceede?

We want to take advantage of the structure of the Feedforward Neural Network.

(+)

## Backpropagation and Chain Rule (2)



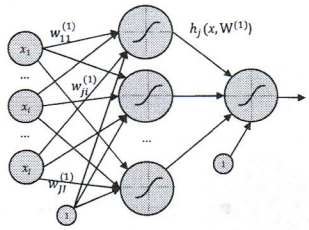

$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = -2\sum_{n}^{N}(t_n - g_1(x_n, w)) \cdot g_1'(x_n, w) \cdot w_{1j}^{(2)} \cdot h_j'\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right) \cdot x_i$$



Resume

We perform a forward-pass which computes temporary values and stores them. These temporary values are basically the outputs of the hidden neurons and the derivatives of the hidden neurons in the points. Then, with backward-pass we can easily implement the backpropagation, which updates the weights.

## Gradient Descent Example




because we have only 1 output

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

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

Why should I use this?

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

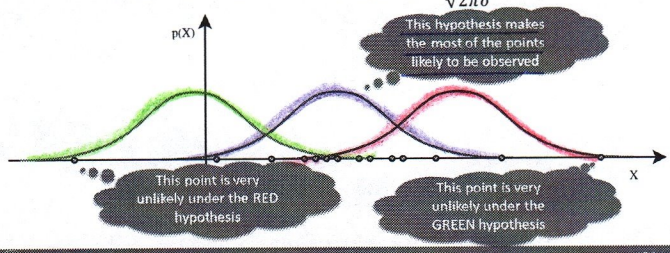


is this the right error function to use? Or we should try something different?
To answer this: ↓

## A Note on Maximum Likelihood Estimation

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

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



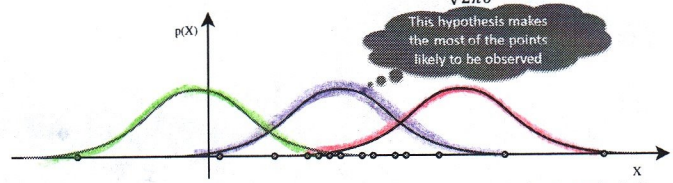



Maximum likelihood: estimation of some unknown parameters of a distribution based on the observation of some data

## A Note on Maximum Likelihood Estimation

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

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




Maximum Likelihood: Chose parameters which maximize data probability



which is the probability that those data come from a distr. with $\mu = \bar{\mu}$?
The $\bar{\mu}$ with the highest prob. is the $\mu_{MLE}$.

## Maximum Likelihood Estimation: The Recipe

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

- Write the likelihood $L = P(Data|\theta)$ for the data
- [Take the logarithm of likelihood $l = \log P(Data|\theta)$]
- 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 $\theta^{MLE}$ is a maximum (and not min)

Optional

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

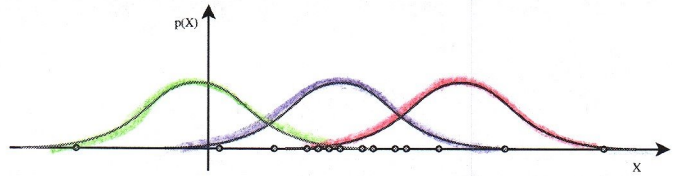
To maximize/minimize the (log)likelihood you can:

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

## Maximum Likelihood Estimation Example

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

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




Find the Maximum Likelihood Estimator for $\mu$



## Maximum Likelihood Estimation Example

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

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

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

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

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



## Maximum Likelihood Estimation Example

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

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

- 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 $\sigma^2$

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

- 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 $\sigma^2$

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

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

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

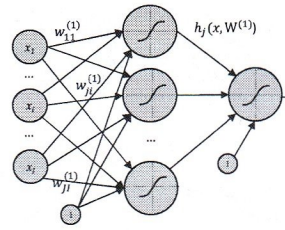
$$\sum_{n}^{N} (x_n - \mu) = 0$$

$$\sum_{n}^{N} x_n = \sum_{n}^{N} \mu \Rightarrow \boxed{\mu^{MLE} = \frac{1}{N} \sum_{n}^{N} x_n}$$

Let's apply this all to Neural Networks!

parameter that maximizes the probability of the data

## Neural Networks for Regression




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

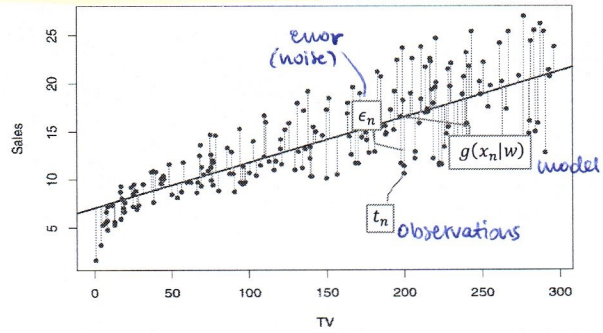
we assume that our data come from our network ($g(x_n|w)$) plus some noise. we assume the noise to be gaussian

Goal: approximate a target function $t$ having N observations

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



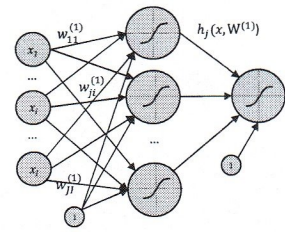
## Statistical Learnig Framework




Notice: later (slide 96) it'll come out that $\sigma^2$ should be at least constant. However, here it seems that if the values are low the variance is low, if the values are high, so the variance is. To (try to) avoid it we can, for instance, transform the data: we can learn to predict $\log(t)$ instead of $t$ (or other transformations).



## Neural Networks for Regression




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

Goal: approximate a target function $t$ having N observations

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

our data come from a deterministic function plus some noise. The deterministic function is the neural network. Note: the only param is $w$, the hyperparams ($g$) are already fixed.



## Maximum Likelihood Estimation for Regression

We have i.i.d. samples coming from a Gaussian with known $\sigma^2$

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

We want to learn the weights in such way that the probability of data is maximized.

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

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

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



## Maximum Likelihood Estimation for Regression

We have i.i.d. samples coming from a Gaussian with known $\sigma^2$

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

that's why we use: $E = \sum_n (t_n - g_1(x_n w))^2$

Look for the weights which maximixe the likelihood

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

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

$$= argmin_w \sum_{n}^{N} (t_n - g(x_n|w))^2$$

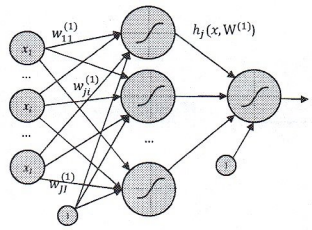
we pass to the likelihood

Maximum likelihood estimation of the network weights (under the previous assumptions)

(!) $\sigma^2$ is a constant, so it's not needed to be known, we only need it to be constant

Here we have to change the settings: we cannot think about classification as a real term output + some gaussian error.

## Neural Networks for Classification




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

We think the output of the network as random variable with 2 possible outcomes $\Rightarrow$ Bernoulli($\theta$) with $\theta$ that says how likely it is to be 0 and 1

We can represent the output of the network as the probability of the object being equal to 1 given the input $x_n$. We can interpret the output of the network in terms of probability.

Goal: approximate a posterior probability $t$ having N observations

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



## Maximum Likelihood Estimation for ~~Regression~~ Classification

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

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

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

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

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



## Maximum Likelihood Estimation for ~~Regression~~ Classification

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

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

a posterior probability of the class given the input (we're learning the Bayes classifier, not the naive one)

Look for the weights which maximize the likelihood

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

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

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

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

it is the sum over all the records of the target value times the logarithm of the output of the record.

What about perceptron

notice that $g(x_n|w)$ is never exactly 0, and so we don't have $\log(0)$

BINARY

MULTIPLE

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

this is a $t_n^T$ because here it's considered as a vector (because if we consider for instance 3 labels we need to use a vector for the output: [1,0,0], [0,1,0], [0,0,1] (each for a label)) The one in yellow is for [0,1], [1,0] (special case).

CASE OF 3 LABELS:

$g = [g_1, g_2, g_3]^T$

$t \in \{[1,0,0]^T, [0,1,0]^T, [0,0,1]^T\}$

crossentropy: $= -\sum_n t_n^T \log(g)$

## How to Chose the Error Function?

We have observed different error functions so far

Sum of Squared Errors

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

Binary Crossentropy

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

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
- Use your creativity!

This requires lots of trial and errors ...

As for the Perceptron ...

like we did: we designed an error function by making some assumptions and by modeling the error (the noise) through MLE



Under some assumptions the heddian learning (perceptron) can be interpreted in terms of gradient descent and maximum likelihood.

We recall that: the perceptron is basically dividing data in two parts $\Rightarrow$ the perceptron is a separating hyperplane, and so is characterized by $w$:

$w_0 + w^T x = 0$

## Hyperplanes Linear Algebra

Let consider the hyperplane (affine set) $L \in \Re^2$

$L: w_0 + w^T x = 0$

Any two points $x_1$ and $x_2$ on $L \in \Re^2$ have

$w^T(x_1 - x_2) = 0$

The versor normal to $L \in \Re^2$ is then

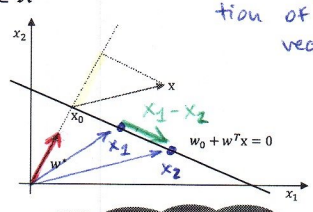
$w^* = w/\|w\|$

For any point $x_0$ in $L \in \Re^2$ we have

$w^T x_0 = -w_0$

The signed distance of any point $x$ in $\Re^2$ from $L$ is:

$$w^{*T}(x - x_0) = \frac{1}{\|w\|}(w^T x + w_0)$$




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

considering $x_1, x_2$: $w_0 + w^T x_i = 0 \quad i = 1,2$ and so: $w^T(x_1 - x_2) = 0$, which means that $(x_1 - x_2)$ is $\perp$ to $w$. An interpretation of the parameters (weights) is a vector that is $\perp$ to the hyperplane.

$w^{*T}(x - x_0)$ is representing the distance from the hyperplane of any point $x$ in the space. This is a signed distance: points above the line have a "+", points below have a "−".

## 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 $\mathbf{w}^T\mathbf{x} + w_0 < 0$
- For an output with -1 we have the opposite

We have an error when the point is on the wrong side of the line. How do we know?

The goal becomes minimizing

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

Set of points misclassified

This is non negative and proportional to the distance of the misclassified points from $\mathbf{w}^T\mathbf{x} + w_0 = 0$

This cost function (error function) is different from the previous ones. It is not based on assumptions of the probability distribution of the data. It's build on geometric reasoning.

We minimize this error function through SGD

minimize the errors means to minimize this cost function. this cost function is the total distance of the errors.

$t_i(\mathbf{w}^T\mathbf{x}_i + w_0) < 0$ for all the misclassified $x_i$, and so the $\Sigma$ is $< 0$, and so we put a "$-$".



## Perceptron Learning Algorithm (2/2)

Let's minimize by stochastic gradient descend the error function

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

The gradients with respect to the model parameters are

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

Stochastic gradient descent applies for each misclassified point

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

Hebbian learning implements Stocastic Gradient Descent



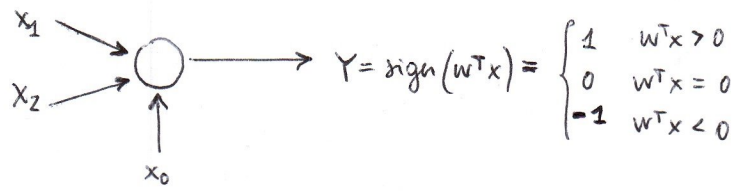
equivalent to:

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

How does it work (in practice) supervised learning?
(We see **Hebbian learning** and **perceptrons**, however these are <u>not</u> used these days, what it's used instead is BACKPROPAGATION)

We want to learn the "OR":

| | $x_0$ | $x_1$ | $x_2$ | $t_{OR}$ |
|---|---|---|---|---|
| (1st) | 1 | -1 | -1 | -1 |
| (2nd) | 1 | -1 | 1 | 1 |
| (3rd) | 1 | 1 | -1 | 1 |
| (4th) | 1 | 1 | 1 | 1 |




We start with $w^0 = [w_0, w_1, w_2]^0 = [1, 1, 1]$

1st: $Y = sign(w^T x) = sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = sign(1 - 1 - 1) = -1$ ✓

2nd: $Y = sign(w^T x) = sign(1 - 1 + 1) = 1$ ✓

3rd: $Y = sign(w^T x) = sign(1 + 1 - 1) = 1$ ✓

4th: $Y = sign(w^T x) = sign(1 + 1 + 1) = 1$ ✓

Lucky initialization!
What if we start from: $w^0 = [w_0, w_1, w_2]^0 = [-1, -1, -1]$

1st: $Y = sign(w^T x) = sign((-1)(1) + (-1)(-1) + (-1)(-1)) = sign(-1 + 1 + 1) = 1$ ✗

How can we fix it? Hebbian learning!

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^1 = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^0 + \eta \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} t = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} (-1) = \begin{bmatrix} -3/2 \\ -1/2 \\ -1/2 \end{bmatrix}$$

$\Rightarrow$ new set of weight: $[w_0, w_1, w_2]^1 = [-\frac{3}{2}, -\frac{1}{2}, -\frac{1}{2}]$

2nd: $Y = sign(w^T x) = sign(-\frac{3}{2} \cdot 1 - \frac{1}{2} \cdot (-1) - \frac{1}{2}(-1)) = sign(-\frac{3}{2}) = -1$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^2 = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^1 + \eta \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} t = \begin{bmatrix} -3/2 \\ -1/2 \\ -1/2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} (1) = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix}$$

$\Rightarrow [w_0, w_1, w_2]^2 = [-1, -1, 0]$

3rd: $Y = sign(w^T x) = sign((-1)(1) + (-1)(1) + 0(-1)) = sign(-1 - 1) = -1$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^3 = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} 1 = \begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \end{bmatrix}$$

$\Rightarrow [w_0, w_1, w_2]^3 = [-\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}]$

4th: $Y = sign(w^T x) = sign(-\frac{1}{2}(1) - \frac{1}{2}(1) - \frac{1}{2}(1)) = sign(-3/2) = -1$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^4 = \begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} 1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$\Rightarrow [w_0, w_1, w_2]^4 = [0, 0, 0]$

<u>notice</u>: when the output is zero we're probably wrong

Batch procedure ! is not possible with Hebbian learning ←

Notice: we use the "online" procedure, we update the weights and we use them for the <u>next</u> record (there are some alternatives: if we go through all the data and we update all together the weights at the end we're using BATCH procedure)

We have been through all the data. A pass is called **EPOCH.**
Every time we update the weights we do one "iteration".
Here we did one epoch and four iterations.
We're not done yet, we should have a fully correct EPOCH.
Theoretically, if it'll converge, at every epoch we have less mistakes.

EPOCH 2:

1st: $Y = \text{sign}(w^T x) = \text{sign}(0+0+0) = 0$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^5 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}(-1) = \begin{bmatrix} -1/2 \\ 1/2 \\ 1/2 \end{bmatrix}$$

$$\longrightarrow [w_0\ w_1\ w_2]^5 = [-1/2,\ 1/2,\ 1/2]$$

2nd: $Y = \text{sign}(w^T x) = \text{sign}(-1/2 - 1/2 + 1/2) = -1$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^6 = \begin{bmatrix} -1/2 \\ 1/2 \\ 1/2 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}(1) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\longrightarrow [w_0, w_1, w_2]^6 = [0, 0, 1]$$

3rd: $Y = \text{sign}(w^T x) = \text{sign}(-1) = -1$ ✗

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}^7 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}(1) = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}$$

$$\longrightarrow [w_0, w_1, w_2]^7 = [1/2,\ 1/2,\ 1/2]$$

4th: $Y = \text{sign}\left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) = \text{sign}\left(\frac{3}{2}\right) = 1$ ✓

Are we done? No, even if the last result is correct we have to obtain an entire epoch which is correct!

EPOCH 3:

1st: $Y = \text{sign}(w^T x) = \text{sign}\left(\frac{1}{2} - \frac{1}{2} - \frac{1}{2}\right) = \text{sign}\left(-\frac{1}{2}\right) = -1$ ✓

2nd: $Y = \text{sign}(w^T x) = \text{sign}\left(\frac{1}{2} - \frac{1}{2} + \frac{1}{2}\right) = \text{sign}\left(\frac{1}{2}\right) = 1$ ✓

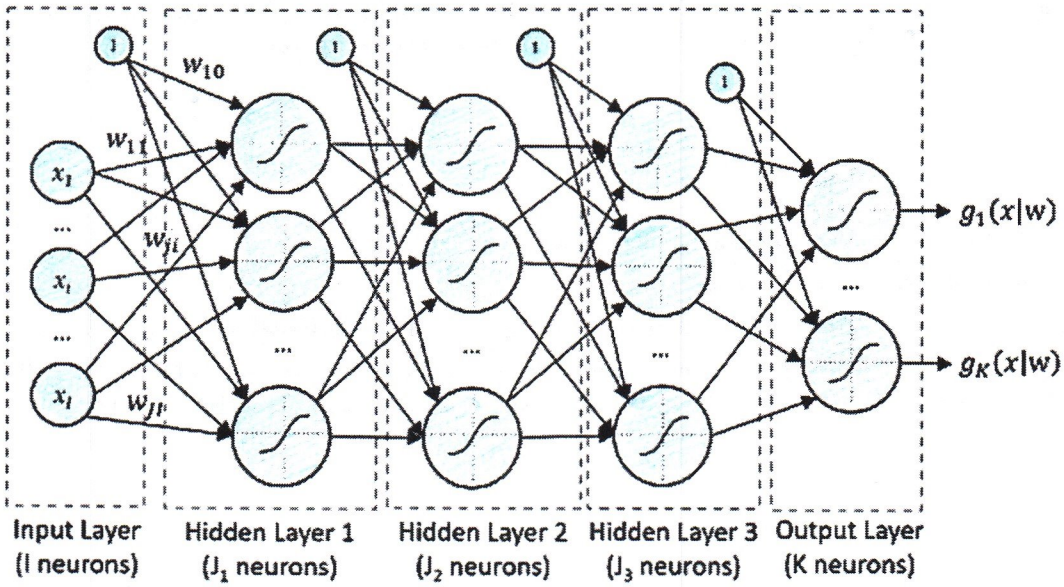
3rd: $Y = \text{sign}(w^T x) = \text{sign}\left(\frac{1}{2} + \frac{1}{2} - \frac{1}{2}\right) = \text{sign}\left(\frac{1}{2}\right) = 1$ ✓

4th: $Y = \text{sign}(w^T x) = \text{sign}\left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) = \text{sign}\left(\frac{3}{2}\right) = 1$ ✓

$\Longrightarrow$ we converged with $[w_0, w_1, w_2] = \left[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right]$

notice that this is ≠ from [1,1,1] (which was the first lucky case!)

# FEED FORWARD NEURAL NETWORKS




We saw that to solve the problem of $\nexists$ linear boundary (separating hyperplane) we can either try to find a nonlinear boundary or try to change the form of the input such that the transformed inputs are linearly separable. This structure is doing both. Every layer transform the inputs into new representation and at the end this new representation is processed. At the end the boundary performed at the output is highly non-linear w.r.t. the inputs (the original inputs).

The true parameters of this model are the weights, while how many layers, how many neurons and how many activation functions are sometimes called "hyperparameters" (because once we decide them, then we know how many parameters has the model). = all the things that we have to decide to build the model

All the layers are connected through weights.
We collect all the weights in a matrix:

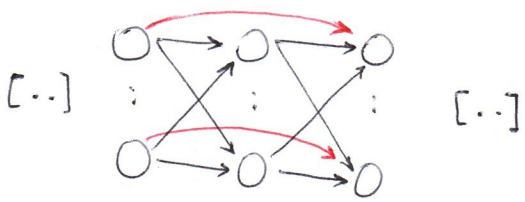
$$W^{(l)} = \{ w_{ji}^{(l)} \} := \text{"dense matrix"}$$

We make the assumption that $W^{(l)} = \{ w_{ji}^{(l)} \}$ is full. (so every hidden layer is connected).

Notice that: the output of a neuron ($\forall$neuron, $\forall$layer) depends only on the previous layers:

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

There are some exceptions. It may be something like:

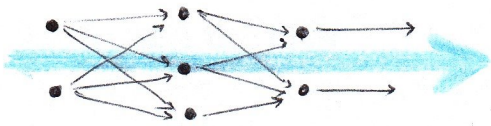



these connections are called "shortcuts". The model we're going to discuss works also with them (since we can imagine some "fictitious" neurons which copy the inputs:


)

In principle we can even connect the inputs with the outputs through these shortcuts.

Anyway, the important thing is that the informations must alway flow forward:



the signal is going through the network once and it does <u>not</u> create cycles. Moreover we don't have connections between neurons in the same layer.

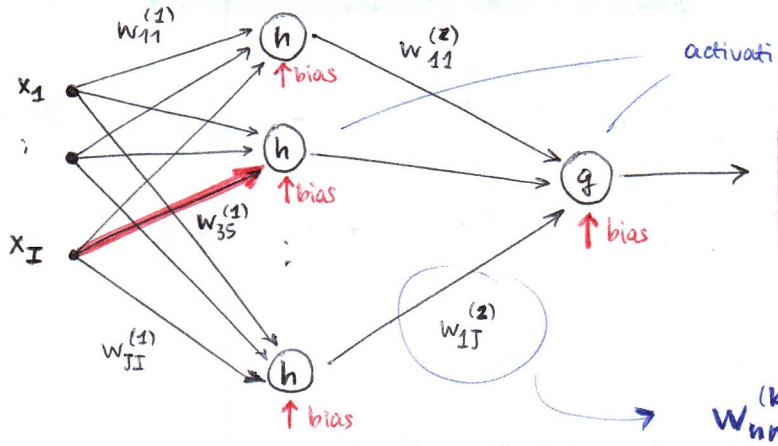
The requirements for the method are two:

- the signal goes only forward
- the activation functions are differentiable ($\exists$ a derivative and it's continuous)
  (if they're all differentiable then the whole model (which is the combination of differentiable functions) is differentiable and then can be derived).

Possible activation functions:

- Linear
- Sigmoid
- Tanh

# Gradient Descent Example




I = # input neurons (actually I + 1 because of the bias)
J = # neurons (hidden)
1 hidden layer

We want to update the weights, more precisely we focus on $w_{35}^{(1)}$:

$$w_{35}^{k+1} = w_{35}^{k} - \eta \left.\frac{\partial E}{\partial w_{35}}\right|_{w^k}$$

error function

How many weights to be updated?

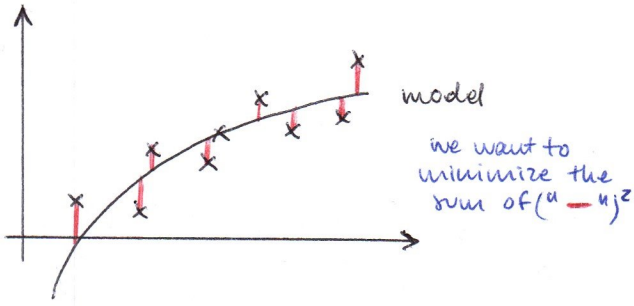
(I+1)·J = # updates from I to J
(J+1)·1 = # updates from J to 1

$$\Rightarrow \quad I \xrightarrow{(I+1)J} J \xrightarrow{(J+1)\cdot 1} 1$$

What is the error function?

$$E = \sum_n (t_n - y_n)^2$$

sum of squared differences of the target and what we have as output




Concretely:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = \frac{\partial}{\partial w_{35}^{(1)}} \sum_n (t_n - y_n)^2$$

$$= \sum_n \frac{\partial}{\partial w_{35}^{(1)}} (t_n - y_n)^2$$

$$= \sum_n 2(t_n - y_n) \frac{\partial (t_n - y_n)}{\partial w_{35}^{(1)}}$$

$$= -2 \sum_n (t_n - y_n) \frac{\partial y_n}{\partial w_{35}^{(1)}}$$

$$\frac{\partial t_n}{\partial w_{35}^{(1)}} = 0 \quad , \quad \frac{\partial (-y_n)}{\partial w_{35}^{(1)}} = -\frac{\partial y_n}{\partial w_{35}^{(1)}}$$

$$\frac{\partial y_n}{\partial w_{35}^{(1)}} = \frac{\partial}{\partial w_{35}^{(1)}} g\left(\sum_{j=0}^{J} w_{1j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} x_{i,n}\right)\right)$$

$$\frac{\partial y_n}{\partial w_{35}^{(1)}} = g'\left(\sum_{j=0}^{J} w_{1j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} x_{i,n}\right)\right) \frac{\partial}{\partial w_{35}^{(1)}} \sum_{j=0}^{J} w_{1j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} x_{i,n}\right)$$

$$\frac{\partial}{\partial w_{35}^{(1)}} \sum_{j=0}^{J} w_{1j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} x_{i,n}\right) = \frac{\partial}{\partial w_{35}^{(1)}}\left[w_{10}^{(2)} + w_{11}^{(2)} h_1\left(\sum_i w_{1i}^{(1)} x_{i,n}\right) + w_{12}^{(2)} h_2\left(\sum_i w_{2i}^{(1)} x_{i,n}\right) + w_{13}^{(2)} h_3\left(\sum_i w_{3i}^{(1)} x_{i,n}\right) + \dots + w_{1J}^{(2)} h_J\left(\sum_i w_{Ji}^{(1)} x_{i,n}\right)\right]$$

$$= \frac{\partial}{\partial w_{35}^{(1)}}\left(w_{13}^{(2)} h_3\left(\sum_i w_{3i}^{(1)} x_{i,n}\right)\right)$$

$$= w_{13}^{(2)} \frac{\partial}{\partial w_{35}^{(1)}} h_3\left(\sum_i w_{3i}^{(1)} x_{i,n}\right)$$

$$\frac{\partial}{\partial w_{35}^{(1)}} h_3\left(\sum_i w_{3i}^{(1)} x_{i,n}\right) = h_3'\left(\sum_i w_{3i}^{(1)} x_{i,n}\right) \cdot \frac{\partial}{\partial w_{35}^{(1)}}\left(\sum_i w_{3i}^{(1)} x_{i,n}\right)$$

$$\frac{\partial}{\partial w_{35}^{(1)}}\left(\sum_i w_{3i}^{(1)} x_{i,n}\right) = \frac{\partial}{\partial w_{35}^{(1)}}\left[w_{30}^{(1)} + w_{31}^{(1)} x_{1,n} + \dots + w_{35}^{(1)} x_{5,n} + \dots + w_{3I}^{(1)} x_{I,n}\right]$$

$$= x_{5,n}$$

$$\Longrightarrow \frac{\partial E}{\partial w_{35}^{(1)}} = \left[-2\sum_n (t_n - y_n)\right] \cdot \left[g'(\bullet)\right] \cdot \left[w_{13}^{(2)}\right] \cdot \left[h_3'(\bullet)\right] \cdot \left[x_{5,n}\right]$$

where $g'(\bullet) = g'\left(\sum_j w_{1j}^{(2)} h_j\left(\sum_i w_{ji}^{(1)} x_{i,n}\right)\right)$ and $h_3'(\bullet) = h_3'\left(\sum_i w_{3i}^{(1)} x_{i,n}\right)$

Notice that:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = \underbrace{\left[-2\sum_n (t_n - y_n)\right]}_{\frac{\partial E}{\partial y_n}} \cdot \underbrace{\left[g'(\bullet)\right]}_{\frac{\partial y_n}{\partial \sum_j w_{1j}^{(2)} h_j(\bullet)}} \cdot \underbrace{\left[w_{13}^{(2)}\right]}_{\frac{\partial \sum_j w_{1j}^{(2)} h_j(\bullet)}{\partial h_3(\bullet)}} \cdot \underbrace{\left[h_3'(\bullet)\right]}_{\frac{\partial h_3(\bullet)}{\partial \sum_i w_{3i}^{(1)} x_{i,n}}} \cdot \underbrace{\left[x_{5,n}\right]}_{\frac{\partial \sum_i w_{3i}^{(1)} x_{i,n}}{\partial w_{35}^{(1)}}}$$

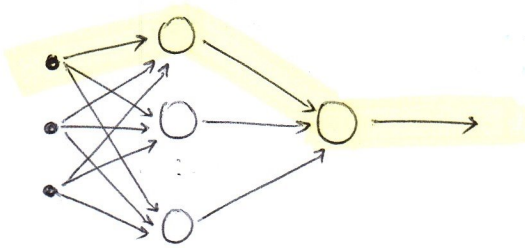
# Backpropagation and chain rule

The "problem" is how to compute:

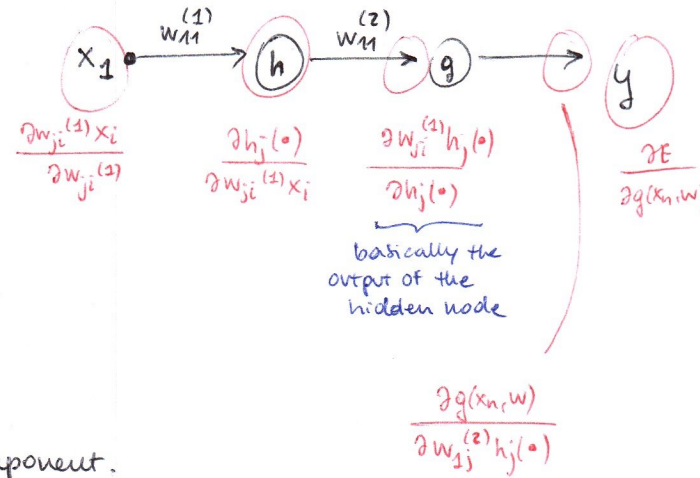
$$\frac{\partial E(w_{ji}^{(1)})}{\partial w_{ji}^{(1)}} = \boxed{-2\sum_n (t_n - g_1(x_n, w))} \boxed{g'(x_n, w)} \boxed{w_{1j}^{(2)}} \boxed{h_j'\left(\sum_{j=0}^{J} w_{ji}^{(1)} x_{i,n}\right)} \boxed{x_i}$$

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

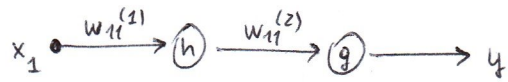
We want to take advantage of the Feedforward Neural Network.
By doing a first pass through the network we can compute all the pieces that we need:



:

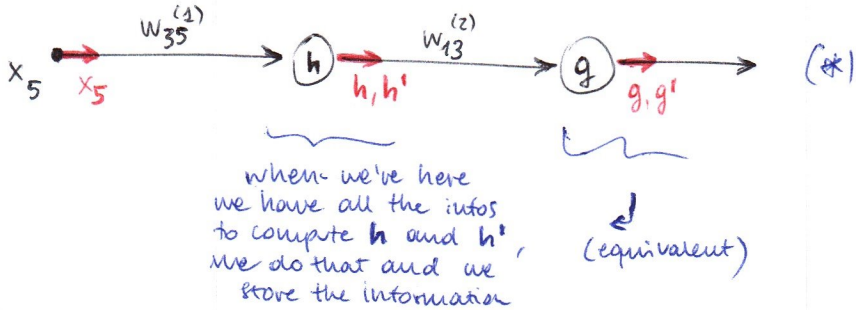



and so, going forward we can store every component.
Once we get to the end we can go back and update the weight.




→ store the information ⟹ $\frac{\partial E}{\partial w_{ji}^{(1)}}$

more precisely: (from the example)

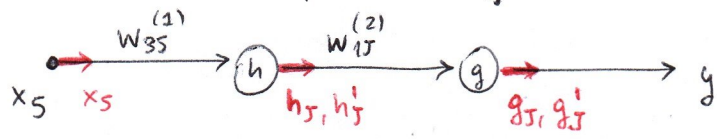



what is the update formula for $w_{35}^{(1)}$?

$$\left(\text{derivative of the error}\right) \cdot \left(g'\right) \cdot \left(h'\right) \cdot \left(x_5\right) \cdot \left(w_{35}^{(1)}\right) = \frac{\partial E}{\partial w_{35}^{(1)}}$$

from the more out (*) to $x_5$: all the derivatives · $x_i$ · $w_{ji}^{(1)}$

What if we want to update $w_{1J}^{(2)}$ ?

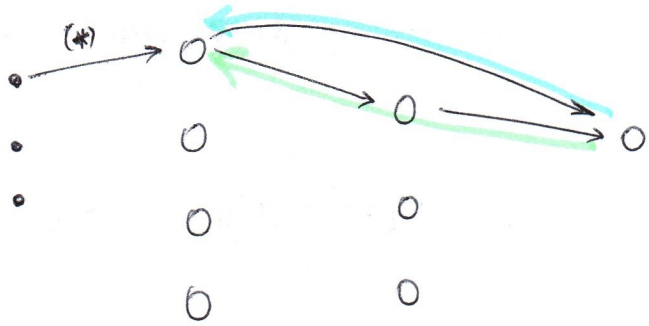



derivative of the error w.r.t. $w_{1J}^{(2)}$ = $\left(-2\sum_n (t_n - y_n)\right) \cdot \left(g'(\cdot)\right) \cdot \left(h_J(\cdot)\right)$

derivative of the error

we stop here because we arrived at the weight that we were interested in

What if we have this neural network: and we want to update (*) ?




⟹ we add the gradients computed through and

POLITECNICO MILANO 1863

# Artificial Neural Networks and Deep Learning

## - Neural Networks Training and Overfitting-

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

AIRLAB

## 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)

Regardless of what function we are learning, a single layer can do it ...

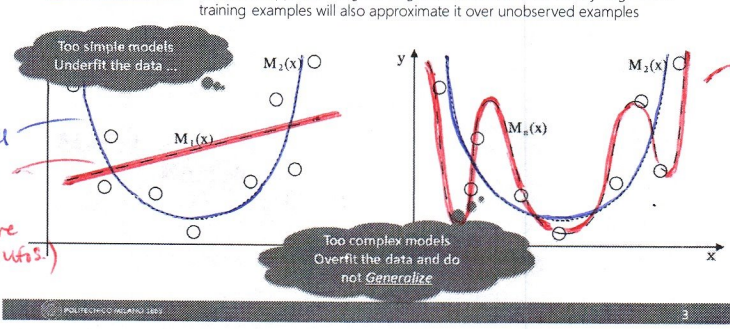
- ... but it doesn't mean we can find the necessary weights!
- ... but an exponential number of hidden units may be required
- ... but it might be useless in practice if it does not generalize! (overfitting)

*"Entia non sunt multiplicanda praeter necessitatem"*
*William of Ockham (c 1285 – 1349)*



## Model Complexity

Inductive Hypothesis: A solution approximating the target function over a sufficiently large set of training examples will also approximate it over unobserved examples




true model

too simple: UNDERFIT (high bias, we're loosing some info)

too adapted: OVERFITTING (high variance, we can't generalize)

Overfitting := training a model so to have a very low error in the training set but which (the model) cannot be generalized on new data (we've overfitting when there is a big difference between training error and test error)



## How to Measure Generalization?

Training error/loss is not a good indicator of performance on future data:

- The classifier has been learned from the very same training data, any estimate based on that data will be optimistic
- New data will probably not be exactly the same as training data
- You can find patterns even in random data

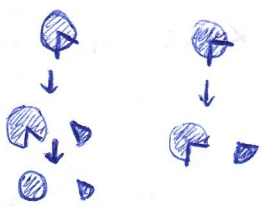
We need to test on an independent new test set

Done for training on small datasets

- Someone provides you a new dataset
- Split the data and hide some of them for later evaluation
- Perform random subsampling (with replacement) of the dataset

this creates a correlation between training data and test data (we end up with a bias estimate of the performance)

In classification preserve class distribution, i.e., stratified sampling!

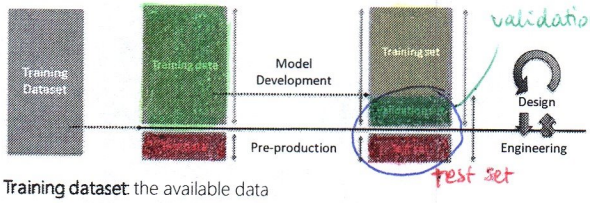


subsampling ≠ splitting:

- if we split a pie, two persons will have half pie
- if a person give to the other a piece of pie and then the person replace it ⇒ one will have a piece and the other will have the entire pie (the key point is "replacement", which creates the problem)



## Clearing the terms ...

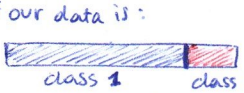



validation set

- **Training dataset**: the available data
- **Training set**: the data used to learn model parameters
- **Test set**: the data used to perform final model assessment
- **Validation set**: the data used to perform model selection
- **Training data**: used to train the model (fitting + selection)
- **Validation data**: used to assess the model quality (selection + assessment)

data we use while training to check if what we're doing is correct

If our data is:




the division (training/test) should be:


(some proportions)

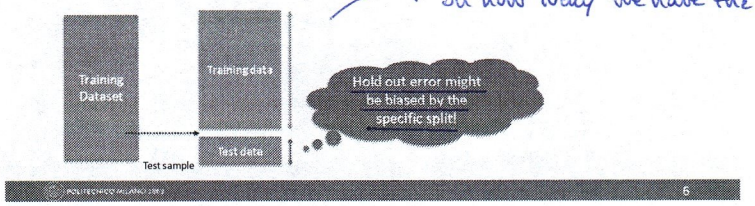
## Cross-Validation

**Validation** := evaluation of the quality of the model.

**How to validate models?**
- ask for new set of data
- Hold-Out
- Leave-one-Out cross-valid.
- K-fold cross-validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data
- When lots of data are available use a Hold Out set and perform validation

problem: the performance depends on how lucky we have the split





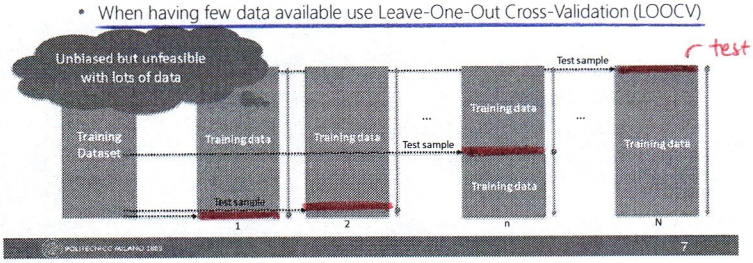

## Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data
- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)

(example) We hide each time just 1 datum, we train on all the others and compute the error on the hidden sample.

This approach is more correct: we use all the data and we test all the data. So, the error should be accurate.

However if we have too many data this procedure is too long (practically impossible)



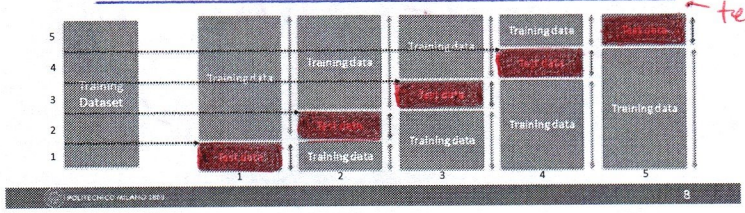



**Notice**: every column (from 1 to N) is trained and tested and so it generates a new model. Here we have **N** models (↓ we'll have **K** models)

## Cross-Validation

Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data
- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)
- K-fold Cross-Validation is a good trade-off (sometime better than LOOCV)

We split the data in **K** independent groups and we repeat the training-test procedure **k** times.






## Cross-Validation

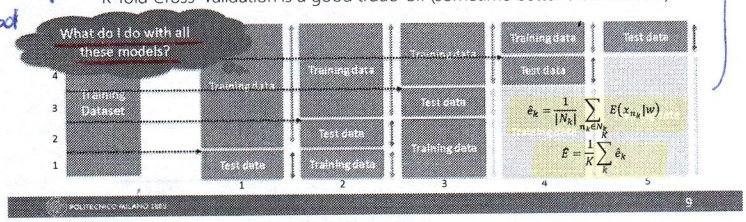
Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data
- When lots of data are available use a Hold Out set and perform validation
- When having few data available use Leave-One-Out Cross-Validation (LOOCV)
- K-fold Cross-Validation is a good trade-off (sometime better than LOOCV)

With this procedure we get **K** errors ($\hat{e}_k$), then we average all the errors obtaining the average error: $\hat{E}$.

Usually this procedure is more accurate of the Hold-Out set and of the Leave-One-Out-Cross-Validation.

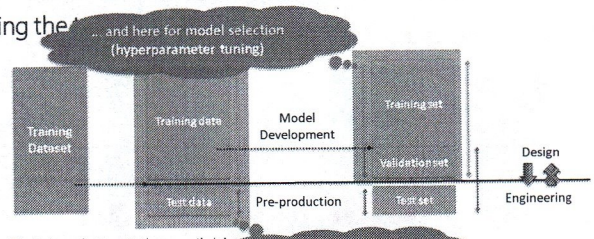
We can, for instance, keep them all and average them. This builds an ENSAMBLE method and it's called BAGGING. Generally, averaging we get a better performance than taking one single model.

Sometimes this is used to check the hyperparameters: at the end of the procedure, if every model is customized, we'll know the optimal number of layers and neurons. Once we know we put all the data together and we train one model (with the optimal hyperparams) from scratch. However this is NOT suggested with Neural Networks.






## Clearing the




- **Training dataset**: the availabl
- Training set: the data use
- Test set: the data used to perfo ssessment
- Validation set: the data used to perform model selection
- Training data: used to train the model (fitting + selection)
- Validation data: used to assess the model quality (selection + assessment)

We can apply K-fold Cross-Validation at this level for model assessment

LIMITATION
* However this method leads us to never use the validation set ⇒ we would like to not waste it

POLITECNICO MILANO 1863

# Artificial Neural Networks and Deep Learning

## - Preventing Neural Networks Overfitting -

= MODEL GENERALIZATION

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

AIRLAB

Methods: (training + prevent overfitting)
- Early stopping
- Weight Decay
- Dropout

discrepancy in performances between the training data and the test data (= gap between training error and test error)

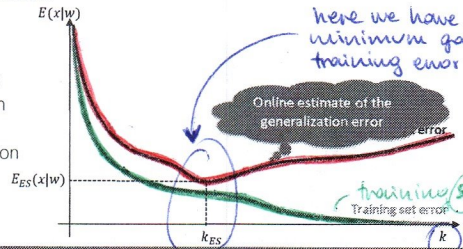
Most simple technique to prevent overfitting: EARLY STOPPING

## 1. Early Stopping: Limiting Overfitting by Cross-validation

= method that stops the training earlier w.r.t. the minimum of the training set. When? → validation set minimum

Overfitting networks show a monotone *training error* trend (on average with SGD) as the number of gradient descent iterations $k$, but they lose generalization at some point ...

- Hold out some data
- Train on the training set
- Perform cross-validation on the hold out set
- Stop train when validation error increases




here we have the minimum gap between training error and validation error

once we find this point we have the model that we test on test set.

validation set error which increase at some point because of the loss of generalizati

training set

there is a point, a moment, when it becomes the overfitting. We need to find that point.

represent # epochs (= # steps of gradient descent)

While training the more iterations we do, the more the error will decrease. At some point it'll be overfitted. How to prevent? Here comes the VALIDATION SET * = data set that we use during the TRAINING procedure to validate what we've doing. This helps the training, the test set does not (it's only to test once everything is over).

- preparation of the model:
  - training set
  - validation set
- test of the model:
  - test set

we use validation to discover when/where stop our training

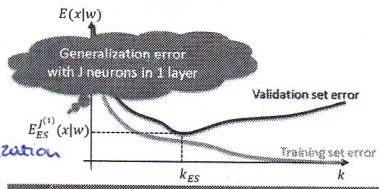
POLITECNICO MILANO 1863 12

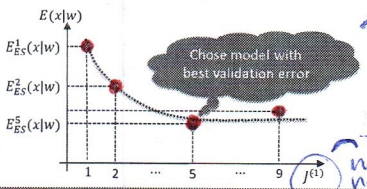
## Cross-validation and Hyperparameters Tuning

Model selection and evaluation happens at different levels:

- Parameters level, i.e, when we learn the weights $w$ for a neural network
- Hyperparameters level, i.e., when we chose the number of layers $L$ or the number of hidden neurons $J^{(l)}$ or a give layer
- ~~Meta-learning, i.e, we learn from data a model to chose hyperparameters~~

hyperparameters e.g.: number of neurons per layer, number of layers, at which epoch we stop, etc.




number of neurons in the hidden layer

we collect all the $E_{ES}$ and we get a curve (←) This curve says that as we increase the complexity of the model (# neurons), the generalization error will decrease. However at some point it's meaningless to add more neurons (the curve gets flat after 5 neurons here).

Then we choose the optimal hyp.s, in this case 1 hidden layer with 5 neurons. Note: we should also try to have more hidden layers.

Suppose to have 1 layer with J neurons. We perform training set error and validation set error. Validation set error tells us where to stop. We get a generalization error $E_{ES}^{J^{(1)}}$. Now we add one neuron to the model and we obtain $E_{ES}^{J+1,(1)}$. We go on and we collect all of these $E_{ES}$. (*)

POLITECNICO MILANO 1863 13

## 2. Weight Decay: Limiting Overfitting by Weights Regularization

Regularization is about constraining the model «freedom», based on a-priori assumption on the model, to reduce overfitting

Many parameters make the model flexible. This can leads to overfitting ⇒ we limit the freedom of the model: we can reduce the number of parameters (for instance) **or** we remove/set some of the weights **or** we can shrink some weights down (imposing that they have to be small). This last approach is called MAXIMUM A-POSTERIORI APPROACH.

So far we have maximized the data likelihood: Maximum Likelihood

maximize the data probability ← $w_{MLE} = argmax_w P(D|w)$

We can reduce model «freedom» by using a Bayesian approach

Make assumption on parameters (a-priori) distribution

Maximum A-Posteriori

$$w_{MAP} = argmax_w P(w|D) = argmax_w P(D|w) \cdot P(w)$$

→ max {(P data given the weights) · (opinion prob. about the weights)}

we used only this in $w_{MLE}$ — PRIOR DISTR.

Small weights observed to improve generalization of neural networks:

$$P(w) \sim N(0, \sigma_w^2)$$

this comes from the fact that famous generalized ANN are good and with small weights ⇒ the probability distribution of w in the networks that generalize better is closer to 0 (and so we can write ↖)

POLITECNICO MILANO 1863 14

## Weight Decay: Limiting Overfitting by Weights Regularization

$$\hat{w} = argmax_w P(w|D) = argmax_w P(D|w) P(w)$$

usual regression likelihood

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

Q = # weights (they're W)

We take the log (Π → Σ) and we change the sign (max → min)

$$= argmin_w \sum_{n=1}^{N} \frac{(t_n - g(x_n|w))^2}{2\sigma^2} + \sum_{q=1}^{Q} \frac{(w_q)^2}{2\sigma_w^2}$$

Here it comes another loss function!!!

this is the idea of RIDGE regression (similar is LASSO: ... + γ Σ |w_q| , which shrink even more)

$$= argmin_w \underbrace{\sum_{n=1}^{N} (t_n - g(x_n|w))^2}_{\text{Fitting}} + \gamma \underbrace{\sum_{q=1}^{Q} (w_q)^2}_{\text{Regularization}}$$

we already had the fitting term, we add the regularization term, which tells us how much we should minimize the weights.

we multiply everything by $\sigma^2$

$$\gamma = \frac{\sigma^2}{\sigma_w^2}$$

trade-off between fitting and regularization

$D_w^2$ = flexibility of the model, $\sigma^2$ = variance of the noise

POLITECNICO MILANO 1863 15

How do we select $\gamma$?
It shouldn't be too high nor too small. The most convenient thing is to set $\gamma$ as a hyperparameter (we have to set $\gamma$ before starting training).

## Recall Cross-validation and Hyperparameters Tuninig

You can use cross-validaton to select the proper $\gamma$:

- Split data in training and validation sets
- Minimize for different values of $\gamma$

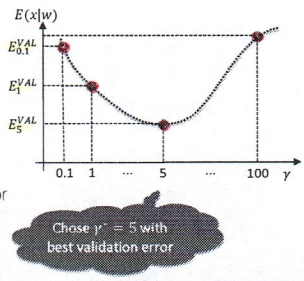
$$w_{MAP}^{train} = E_\gamma^{TRAIN} = \sum_{n=1}^{N_{TRAIN}} (t_n - g(x_n|w))^2 + \gamma \sum_{q=1}^{Q} (w_q)^2$$

- Evaluate the model

$$E_\gamma^{VAL} = \sum_{n=1}^{N_{VAL}} (t_n - g(x_n|w))^2$$

- Chose the $\gamma^*$ with the best validation error
- Put back all data together and minimize

$$E_{\gamma^*} = \sum_{n=1}^{N} (t_n - g(x_n|w))^2 + \gamma^* \sum_{q=1}^{Q} (w_q)^2$$



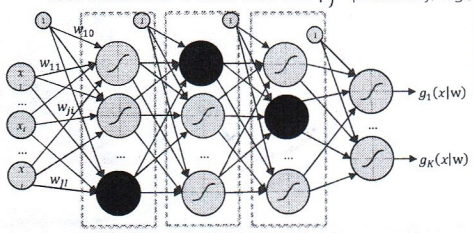

DATA BACK ALL TOGETHER:
suggested to do in WEIGHT DECAY, not suggested with EARLY STOP
[ training + validation (no testing) ]



## 3. Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = 0.3$




$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

$$m_j^{(l)} \sim Be(p_j^{(l)})$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$



Sometimes one neuron has a certain set of weights because of another neuron's. This might prevent a proper learning. We would like the weights to be independent.

each layer we have a "MASK" where every element is from a Bernoulli. Every element in a mask corresponds to a neuron. If the element is 0 then the neuron is switched off. (or otherwise, it depends on the notations). Switching off a neuron means that every connection in and out of that neuron is not used, and so the weights are set to 0.

We generate a new mask at every epoch* (Every time we switch off other neurons).

Once we have all this models (where a model is defined by a set of weight) (we have one model per epoch,* since every time we generate a mask), the final model will be an average of all the models. in this way we obtain that the weights are (more likely) independent

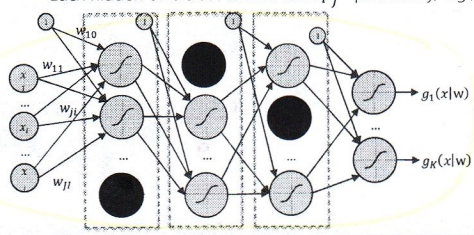
*actually per mini-batch more than per epoch

## Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = 0.3$

mask 1




$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

$$m_j^{(l)} \sim Be(p_j^{(l)})$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

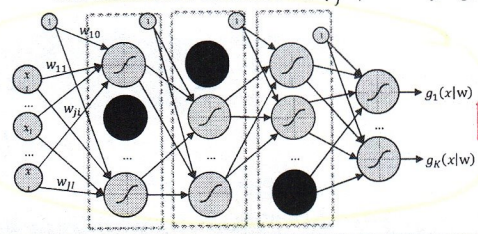


## Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = 0.3$

mask 2




$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

$$m_j^{(l)} \sim Be(p_j^{(l)})$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$



the output of a layer is the weighted sum of the output of the previous layers times the mask.

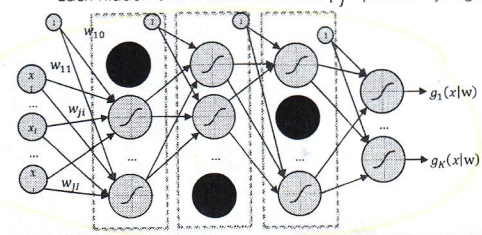
mathematical way to say: "switch off some random neuron"

## Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation):

- Each hidden unit is set to zero with $p_j^{(l)}$ probability, e.g., $p_j^{(l)} = 0.3$

mask 3



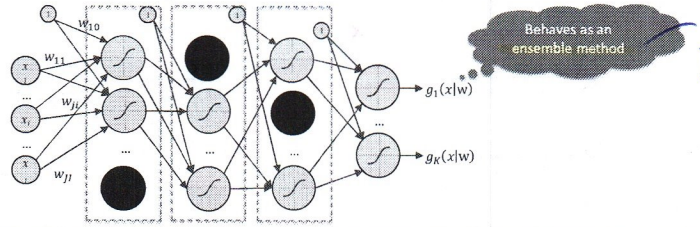

$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

$$m_j^{(l)} \sim Be(p_j^{(l)})$$

$$h^{(l)}(W^{(l)} h^{(l-1)} \odot m^{(l)})$$

## Dropout: Limiting Overfitting by Stochastic Regularization

**Dropout** trains *weaker classifiers*, on different mini- batches and then at test time we implicitly average the responses of all ensemble members.



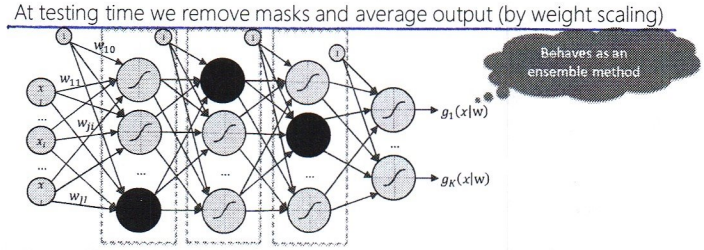



**ENSEMBLE METHOD**: the final model is the mix of a lot of "smaller" models

## Dropout: Limiting Overfitting by Stochastic Regularization

**Dropout** trains *weaker classifiers*, on different mini- batches and then at test time we implicitly average the responses of all ensemble members.

At testing time we remove masks and average output (by weight scaling)






On **which layer** should we apply the mask? Trial and error. This is another hyper-parameter (again cross-validation to select it).

POLITECNICO MILANO 1863

# Artificial Neural Networks and Deep Learning

## - Tips and Tricks in Neural Networks Training -

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

AIRLAB

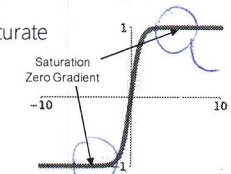
1. Which activation function is better to use?
2. Weights initialization?
3. Training algorithm?

## 1. Better Activation Functions

they're good (because of the universal approximation thm.) but have some issues when we have too big inputs

Activation functions such as Sigmoid or Tanh saturate

- Gradient is close to zero
- Backprop. requires gradient multiplications
- Gradient faraway from the output vanishes
- Learning in deep networks does not happen




if we start from a neuron working in this saturated region we will take forever to go down the gradient and to correct the weights.

Vanishing gradient: effect of multiplying many times numbers $< 1$; we get a gradient which is $\sim 0$ (because the derivatives of the activation functions are small)

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

This is a well known problem in Recurrent Neural Networks, but it affects also deep networks, and it has always hindered neural network training ...

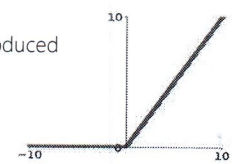
**Conclusion**: if we have a deep network (very "long" or "huge" concatenation of layers) this does not learn. We need an activation function that does not have derivative $< 1 \Rightarrow$ **ReLU**



## Rectified Linear Unit

The ReLU activation function has been introduced

$$g(a) = ReLu(a) = \max(0, a)$$
$$g'(a) = 1_{a>0}$$




It has several advantages: ✓

- Faster SGD Convergence (6x w.r.t sigmoid/tanh) → because the derivative is just an "if"
- Sparse activation (only part of hidden units are activated)
- Efficient gradient propagation (no vanishing or exploding gradient problems), and Efficient computation (just thresholding at zero)
- Scale-invariant: $\max(0, ax) = a\max(0, x)$

! because some of them will have 0 as value of the act. function (we switch off neurons ⇒ this also helps in preventing overfitting)

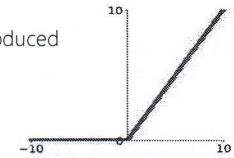
Having gradient $= 1$ is nice since in $\partial E / \partial w_{ji}^{(1)}$ we're not multiplying values $< 1$ but $= 1$. ⇒ with this activation function we can have bigger neural networks (still able to learn)

**Notice**: the universal approx. thm. does not work anymore, but we have analog results for piecewise linear functions

## Rectified Linear Unit

The ReLU activation function has been introduced

$$g(a) = ReLu(a) = \max(0, a)$$
$$g'^{(a)} = 1_{a>0}$$




It has potential disadvantages: ✗

- Non-differentiable at zero: however it is differentiable anywhere else
- Non-zero centered output
- Unbounded: Could potentially blow up
- Dying Neurons: ReLU neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. No gradients flow backward through the neuron, and so the neuron becomes stuck and "dies".

Decreased model capacity, it happens with high learning rates

**Dying Neuron:** A neuron which has a negative input is set to zero, the output is set to zero and also the gradient is zero. If a neuron is working by mistake in $(-\infty, 0)$ and we would like to move the neuron somewhere else this becomes impossible. During training some neurons switch off and we cannot switch them on again.

**Alternatives:**
- Leaky ReLU
- ELU

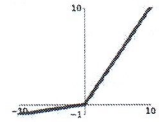
! However it seems that they're not improving impressively → it's okay to use ReLU.



## Rectified Linear Unit (Variants)

**Leaky ReLU**: fix for the "dying ReLU" problem

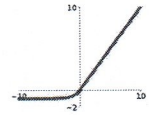
$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01x & otherwise \end{cases}$$




if we work on $(-\infty, 0)$ wrongly we may need some time but we'll be able to come out of the region $(-\infty, 0)$

**ELU**: try to make the mean activations closer to zero which speeds up learning. Alpha is tuned by hand*by hand*

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ \alpha(e^x - 1) & otherwise \end{cases}$$




this is to fix also the non-differentiable point in 0



## 2. Weights Initialization

The final result of gradient descent is affected by weight initialization:

- ✗ Zeros: it does not work! All gradient would be zero, no learning will happen
- ✗ Big Numbers: bad idea, if unlucky might take very long to converge
- ✓ $w \sim N(0, \sigma^2 = 0.01)$: good for small networks, but it might be a problem for deeper neural networks

SMALL NETWORKS →

(vanishing gradient due to the weights)

In deep networks:

- If weights start too small, then gradient shrinks as it passes through each layer
- If the weights in a network start too large, then gradient grows as it passes through each layer until it's too massive to be useful

Some proposal to solve this Xavier initialization or He initialization ...

LARGE NETWORKS ↓



## ⟹ Xavier Initialization

We suppose to have just one hidden neuron →

Suppose we have an input $x$ with $I$ components and a linear neuron with random weights $w$. Its output is

$$h_j = w_{j1}x_1 + \cdots + w_{ji}x_i + \cdots + w_{jI}\,x_I$$

→ output of the hidden neuron: we want it to be not too large and not too small

We can derive that $w_{ji}x_i$ is going to have variance

$$Var(w_{ji}x_i) = E[x_i]^2 Var(w_{ji}) + E[w_{ji}]^2 Var(x_i) + Var(w_{ji})Var(x_i)$$

We try to figure out what should be the weight on average not to have the gradient to shrink to much nor to become too big.

Now if our inputs and weights both have mean 0, that simplifies to

$$Var(w_{ji}x_i) = Var(w_{ji})Var(x_i)$$

If we assume all $w_i$ and $x_i$ are i.i.d. we obtain

I = #inputs →

$$Var(h_j) = Var(w_{j1}x_1 + \cdots + w_{ji}x_i + \cdots + w_{jI}\,x_I) = I * Var(w_i)Var(x_i)$$

← $Var(w_i)$ amplifies the input

Variance of output is the variance of the input, but scaled by $I * Var(w_i)$.

We said that we don't want the weights to amplify too much / shrink too much the input. What is the "magic number"? (1)

We want to initialize the weights s.t. $Var(w_i) \sim 1$.



## Xavier Initialization

If we want the variance of the input and the output to be the same :

$$I * Var(w_j) = 1$$

Linear assumption seem too much, but in practice it works!

For this reason Xavier proposes to initialize $w \sim N\left(0, \frac{1}{n_{in}}\right)$

**Another result (similar):**

Performing similar reasoning for the gradient Glorot & Bengio found

$$n_{out}Var(w_j) = 1$$

To accommodate for this and Xavier propose $w \sim N\left(0, \frac{2}{n_{in}+n_{out}}\right)$

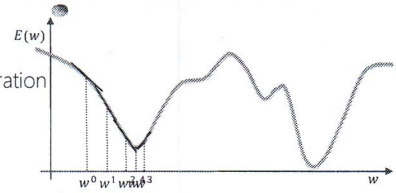
More recently He proposed, for rectified linear units, $w \sim N\left(0, \frac{2}{n_{in}}\right)$

# 3. Training algorithm

## Recall about Backpropagation

Finding weighs of a Neural Network is a non linear minimization process

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




We iterate from a initial configuration

gradient descent :

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

To avoid local minima can use momentum

gradient descent with momentum:

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

Several variations exists beside these two



## More about Gradient Descent

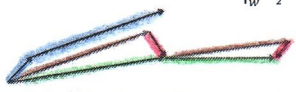
Another variation of the gradient descent:

Nesterov Accelerated gradient: make a jump as momentum, then adjust

— similar to momentum, it converges faster but it does it in 2 steps

$$w^{k+\frac{1}{2}} = w^k - \alpha \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k-1}}$$ ← applies momentum (1.)

$$w^{k+1} = w^k - \eta \left.\frac{\partial E(w)}{\partial w}\right|_{w^{k+\frac{1}{2}}}$$ ← applies gradient descent (2.)



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum



## Adaptive Learning Rates

Neurons in each layer learn differently

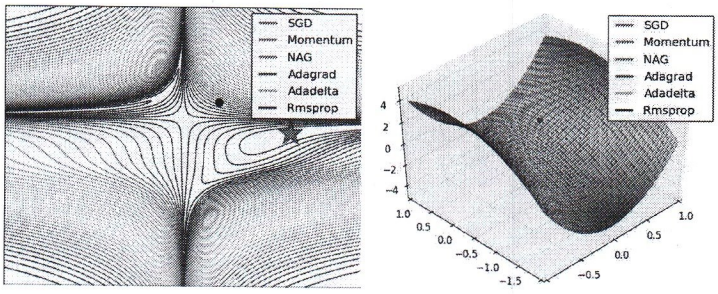
- Gradient magnitudes vary across layers
- Early layers get "vanishing gradients"
- Should ideally use separate adaptive learning rates

Several algoritm proposed:

- Resilient Propagation (Rprop – Riedmiller and Braun 1993)
- Adaptive Gradient (AdaGrad – Duchi et al. 2010)
- RMSprop (SGD + Rprop – Teileman and Hinton 2012)
- AdaDelta (Zeiler et at. 2012)
- Adam (Kingma and Ba, 2012) → keep memory of gradient and try to adapt the learning rate to the gradient
- ...



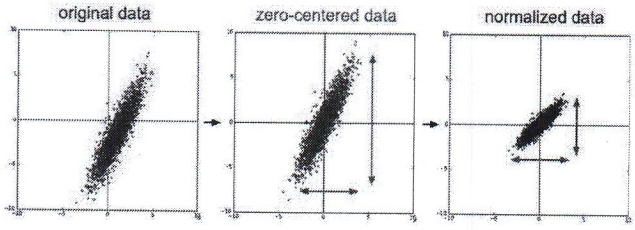
## Learning Rate Matters






## Batch Normalization

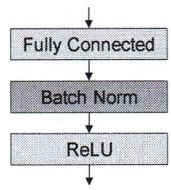
Networks converge faster if inputs have been whitened (zero mean, unit variances) and are uncorrelated to account for *covariate shift*.



## Batch Normalization

Networks converge faster if inputs have been whitened (zero mean, unit variances) and are uncorrelated to account for *covariate shift*.

We can have internal covariate shift; normalization could be useful also at the level of hidden layers.




Batch normalization is a technique to cope with this:

- Forces activations to take values on a unit Gaussian at the beginning of the training
- Adds a BatchNorm layer after fully connected layers (or convolutional layers), and before nonlinearities.
- Can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable way.



## Batch Normalization

In practice

- Each unit's pre-activation is normalized (mean subtraction, stddev division)
- During training, mean and stddev are computed for each minibatch
- Backpropagation takes into account normalization
- At test time, the global mean / stddev are used (global statistics are estimated using training running averages)

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.



## Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

Simple Linear operation!
So it can be back-propagated

Apply a linear transformation, to squash the range, so that the network can decide (learn) how much normalization needs.

Can also learn $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ to recover the $\beta^{(k)} = \mathrm{E}[x^{(k)}]$
Identity mapping



## Batch Normalization

Has shown to

- Improve gradient flow through the network
- Allows using higher learning rates (faster learning)
- Reduces the strong dependence on weights initialization
- Acts as a form of regularization slightly reducing the need for dropout

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

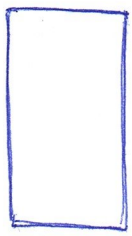
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

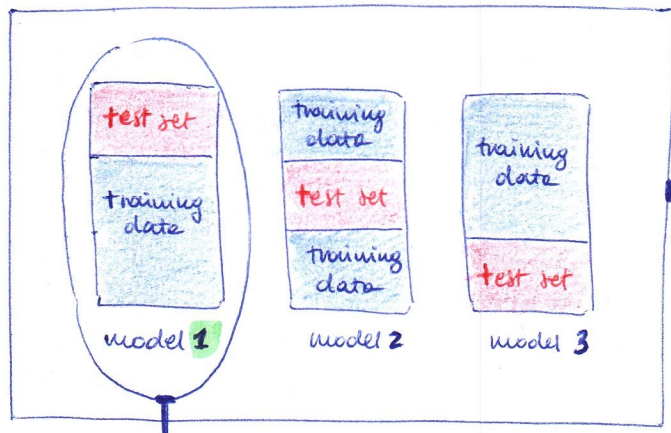
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

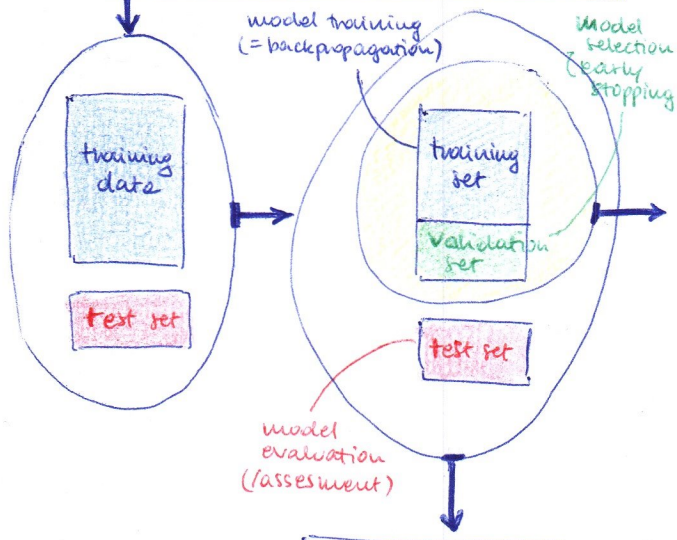
# K-FOLD CROSS VALIDATION WITH EARLY STOPPING



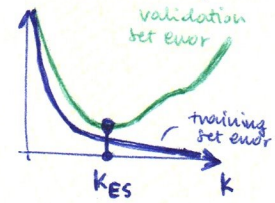

K-fold (k=3)




we average all the $\hat{e}_k$ and we get $\hat{E}$ – says how the model (which is an average of model 1, .., model K) performs on average on new data




**DEVELOPEMENT TEAM:**
model selection and training




$k_{ES}$ = number of steps that we do in the gradient descent (SGD) before we stop = # epochs

We train the model (with backpropagation) with only $k_{ES}$ steps. At the end we have a generalization error of this model.

* We variate the # of neurons / hidden layers and we re-do the work, obtaining another generalization error. At the end we choose the model for which the gen. error is minimal.

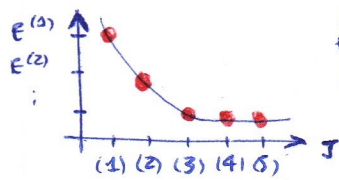
Then we exit this phase with a trained model.

**PRODUCTION TEAM:**
Once we have an optimal model from [ ] we procede in testing it, obtaining an error $\hat{e}_1$

5. We go back at **1.**, we change the model structure and we repeat **2.-4.**

We end the loop when we think we tried enough model structures.

6. Show all the errors found in 4. :




(here J is just the number of neurons in the hidden layer, we varied only # neurons)

We choose the model structure (with the relative weights) that have the min $E^{(j)}$.

→ Now we have the model.

* We have training + validation set :

1. we decide a model structure (#hidden layers, # neurons per layer)
2. We train the model structure with backpropagation : every time we end the epoch we get a set of weights. At the end of every epoch we test the model (with the newest set of weights) on the validation set, obtaining a validation set error. Going on with epochs (k ↑) we obtain the two curves.
3. We select **k** (# epochs) s.t. the validation set error is minimum and we take the set of weights corresponding to that **k** (:= $k_{ES}$)
4. We evaluate the error with the weights of $k_{ES}$ ⇒ $E_{ES}^{(1)}$