POLITECNICO
MILANO 1863

Cognitive Robotics

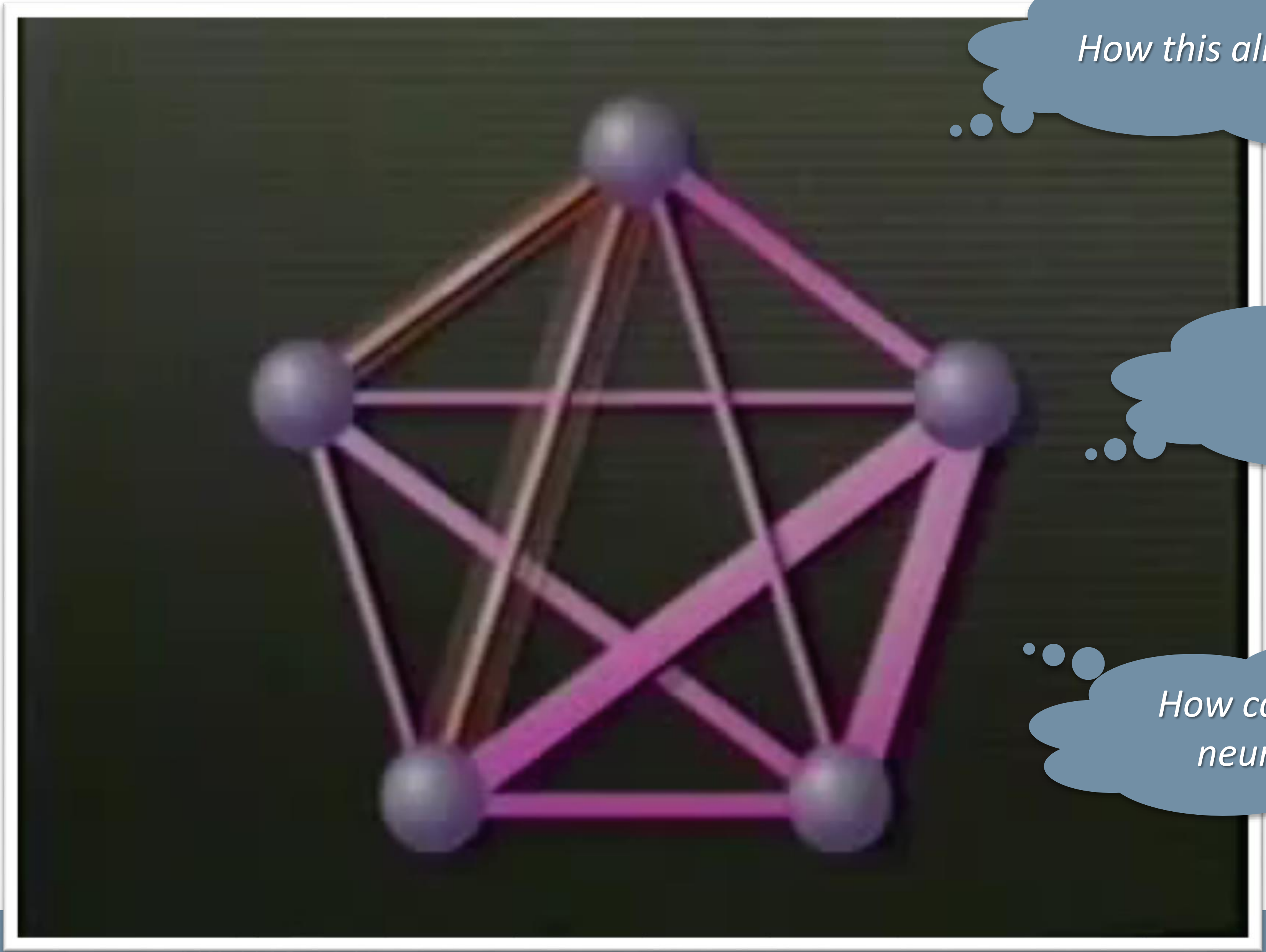
2017/2018

From Perceptrons to Neural Networks

Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

In principle it was the Perceptron ...

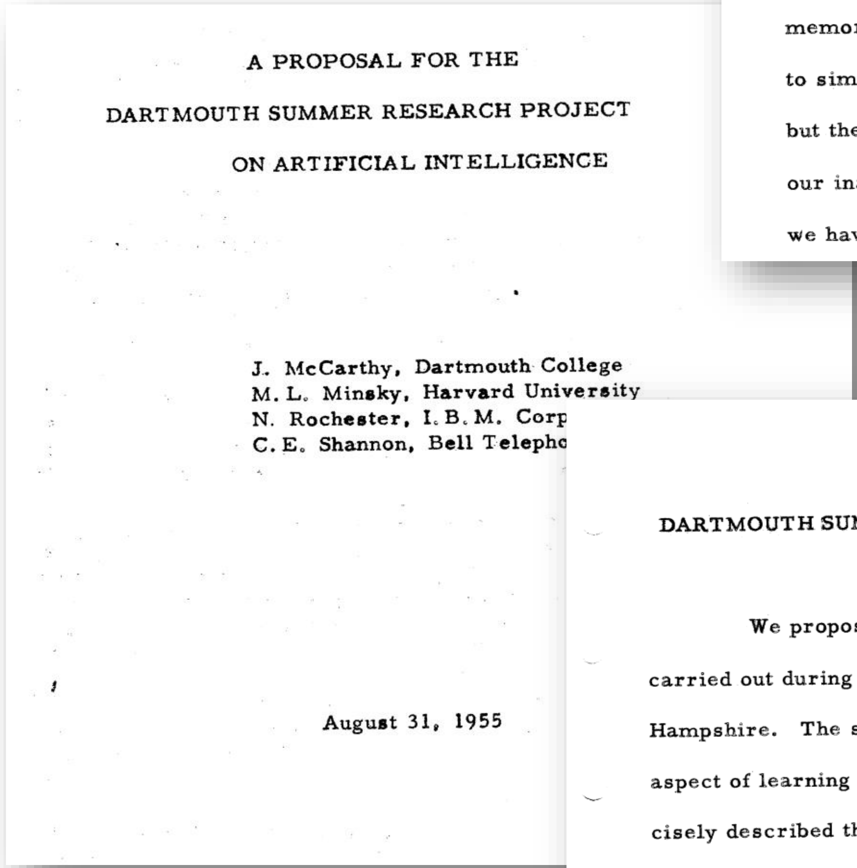


How this all started out?

Why it eventually died out?

How came we still use neural networks?

The inception of AI



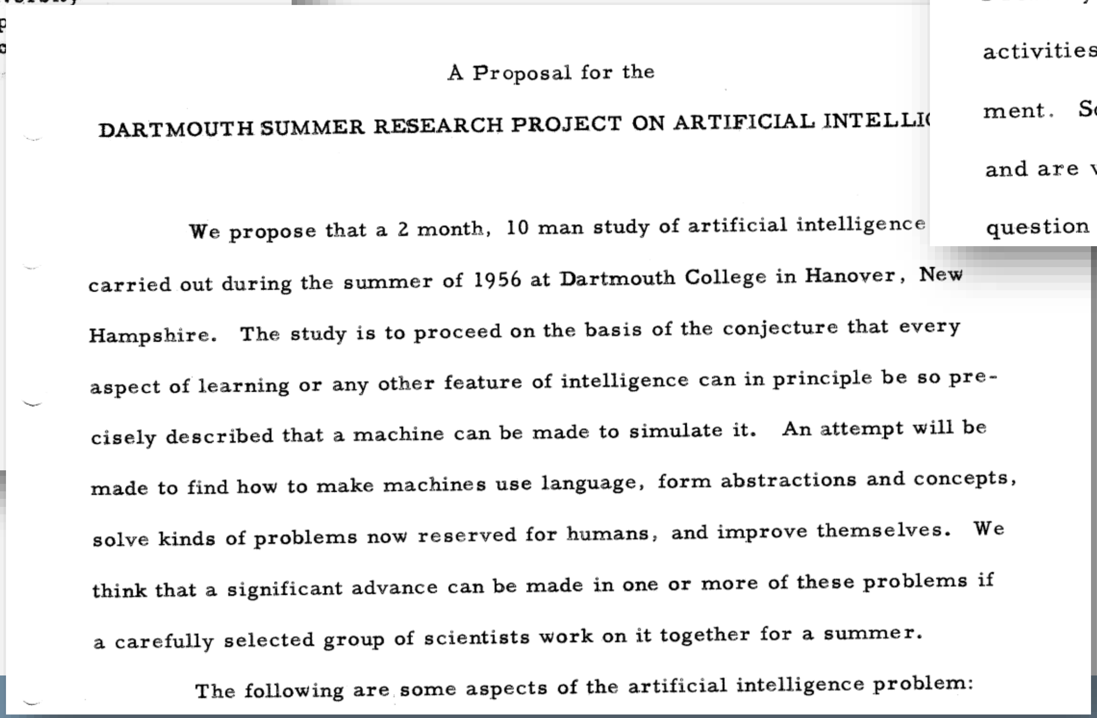
A PROPOSAL FOR THE

DARTMOUTH SUMMER RESEARCH PROJECT

ON ARTIFICIAL INTELLIGENCE

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

August 31, 1955

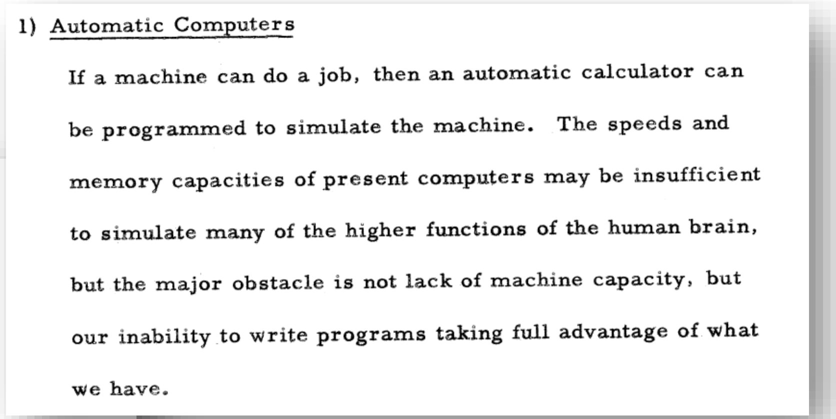


A Proposal for the

DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLI

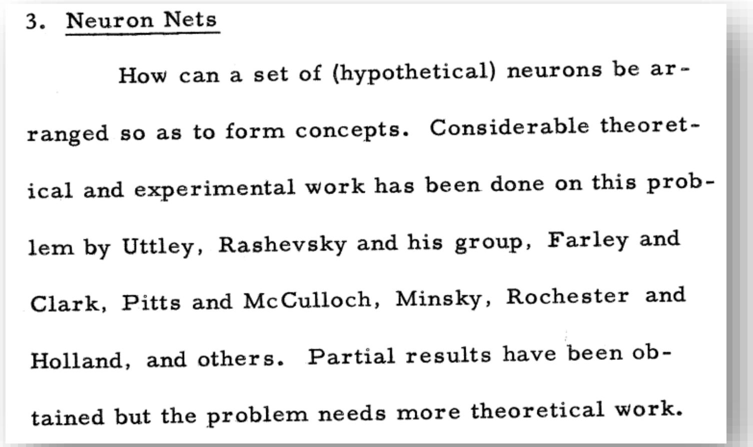
We propose that a 2 month, 10 man study of artificial intelligence carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely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

The following are some aspects of the artificial intelligence problem:



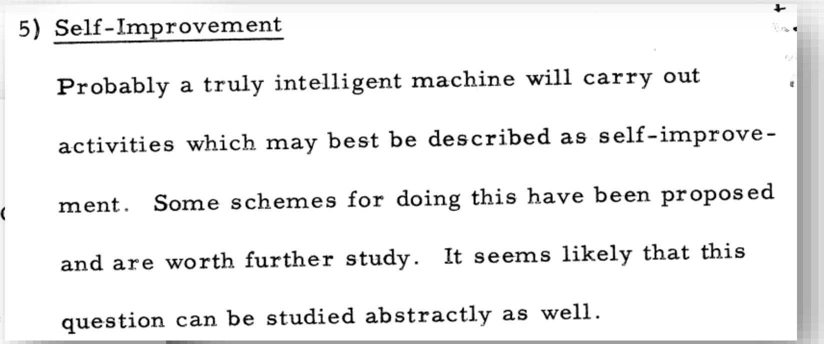
1) Automatic Computers

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



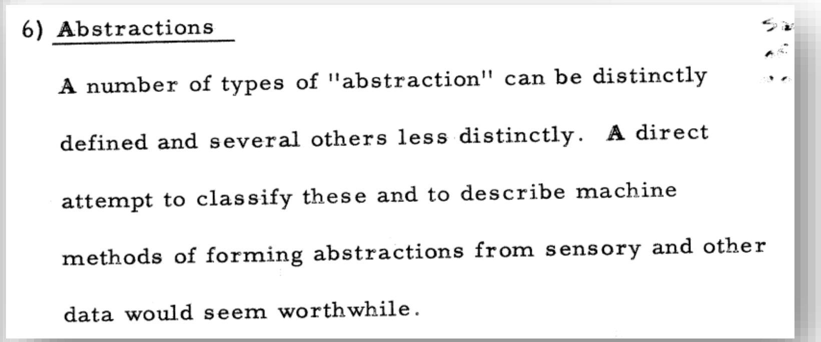
3. Neuron Nets

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



5) Self-Improvement

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



6) Abstractions

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

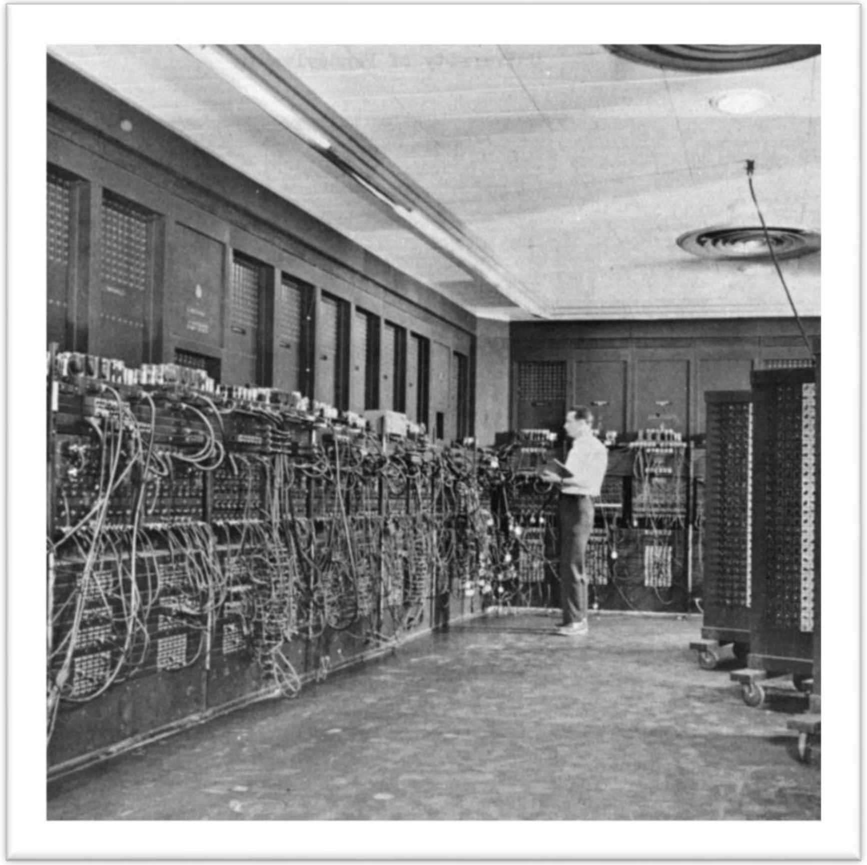
Let’s go back to 1940s ...

Computers were already good at

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

However 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 other than *Von Neumann Machine*!

The Brain Computationa Model

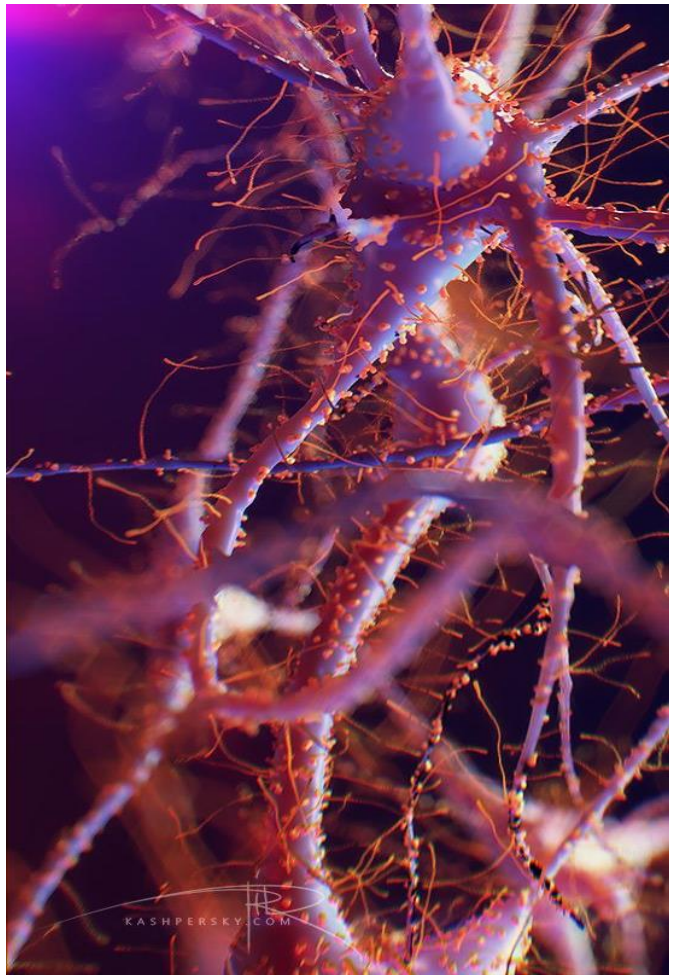
The human brain has a huge number of computing units:

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

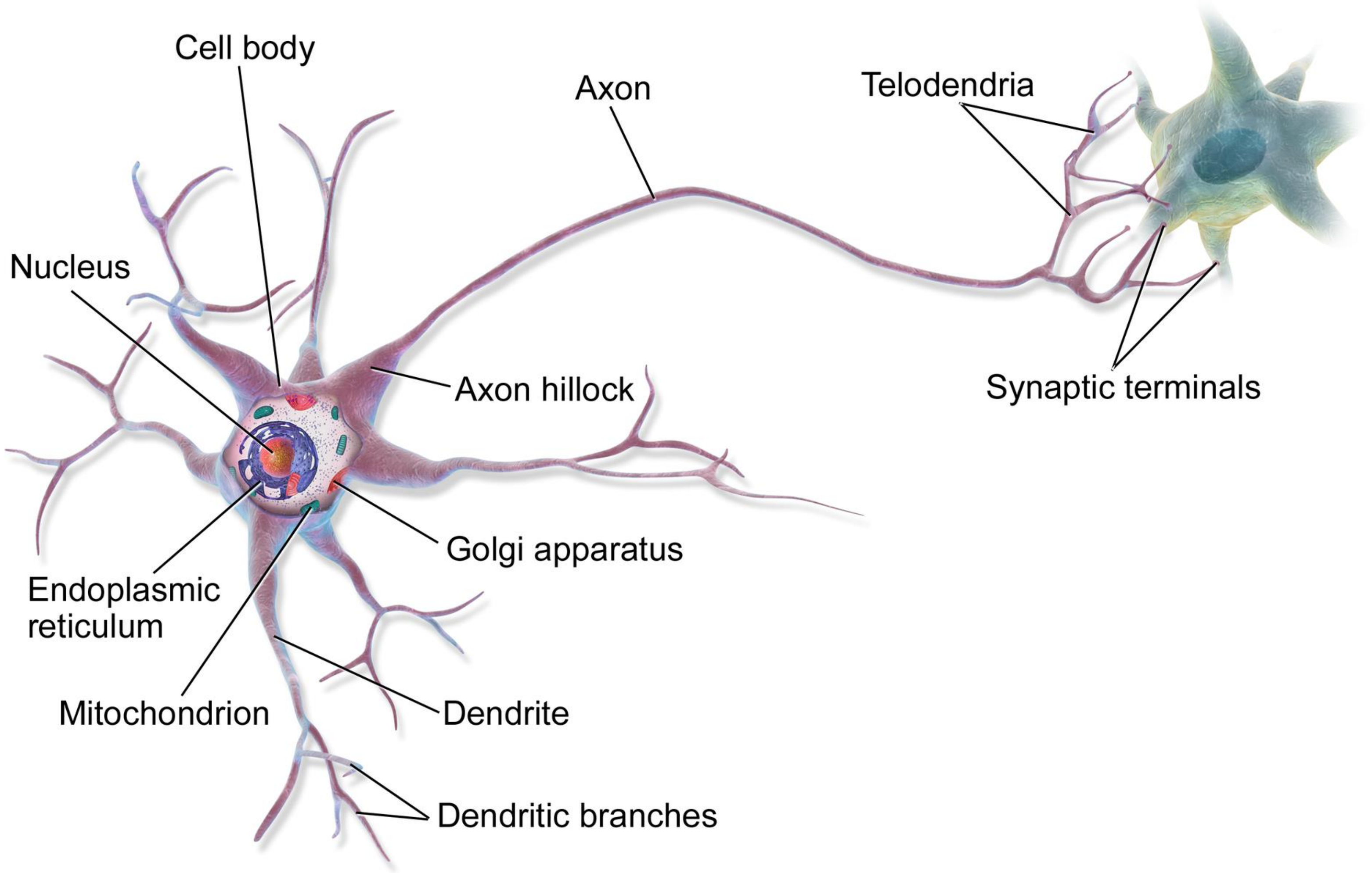
The computational model of the brain is:

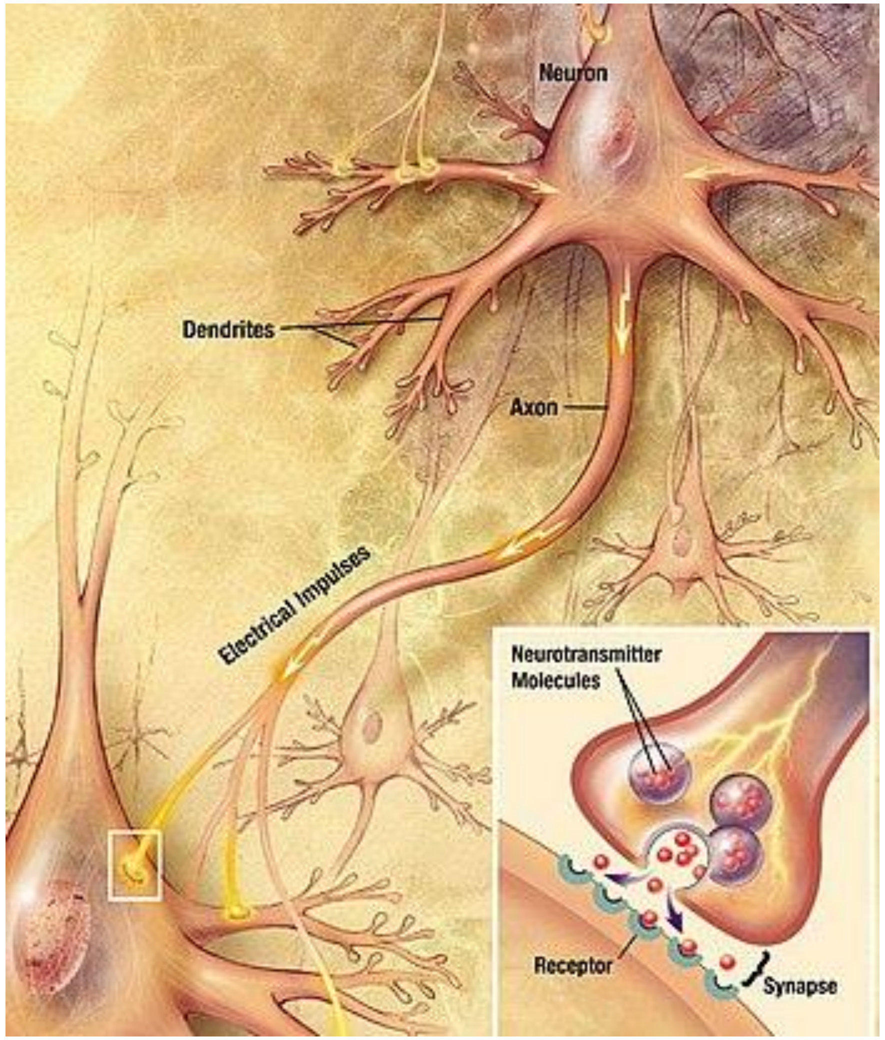
- Distributed among simple non linear units
- Redundant and thus fault tolerant
- Intrinsically parallel

Perceptron: a computational model based on the brain!



Computation in Biological Neurons

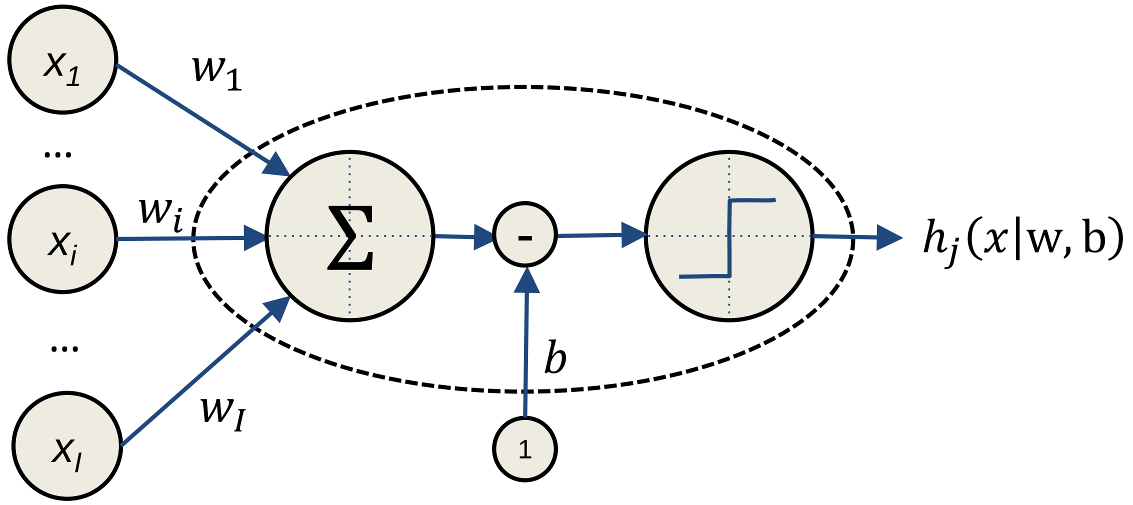




Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

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

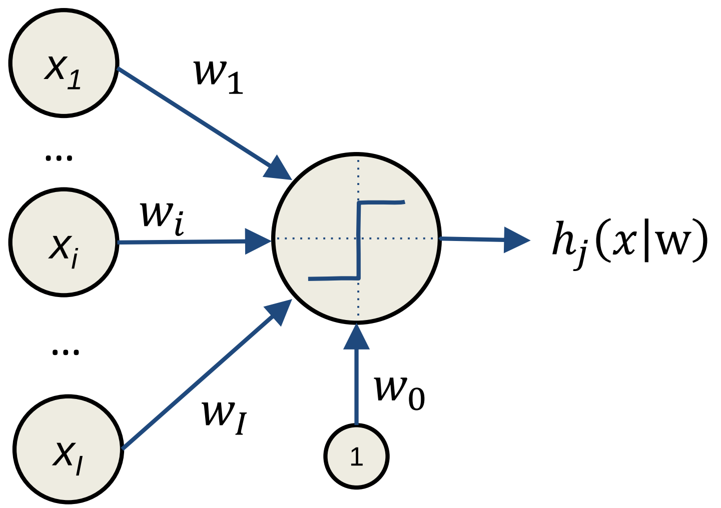


$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

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

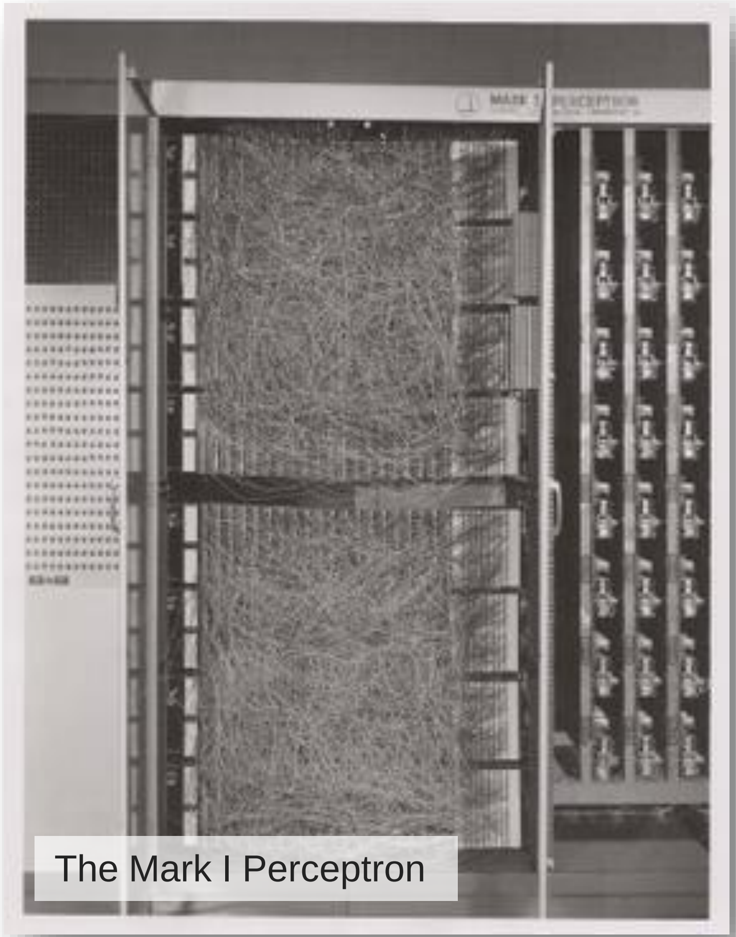


$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

Who did it first?

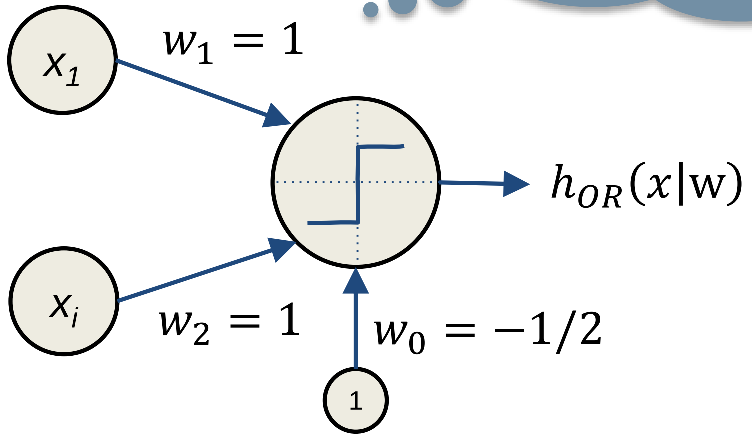
Several researchers were investigating models for the brain

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


The Mark I Perceptron

What can you do with it?

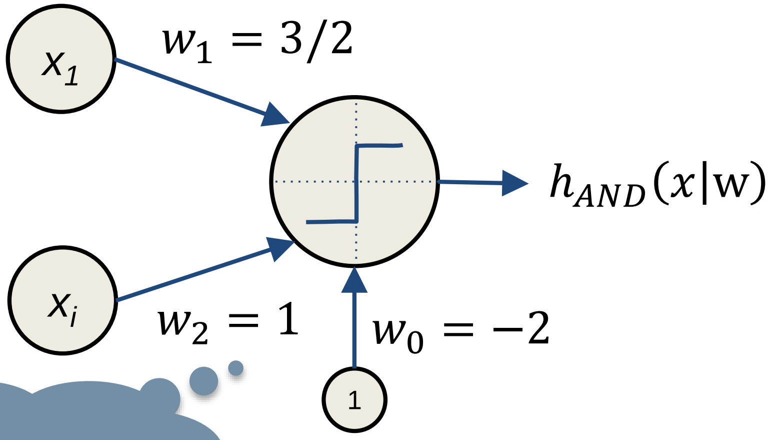
x_0	x_1	x_2	OR
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Perceptron as Logical OR

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

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



Perceptron as Logical AND

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

Perceptron Math

A perceptron computes the value of a weighted sum and returns its Sign (Thresholding)

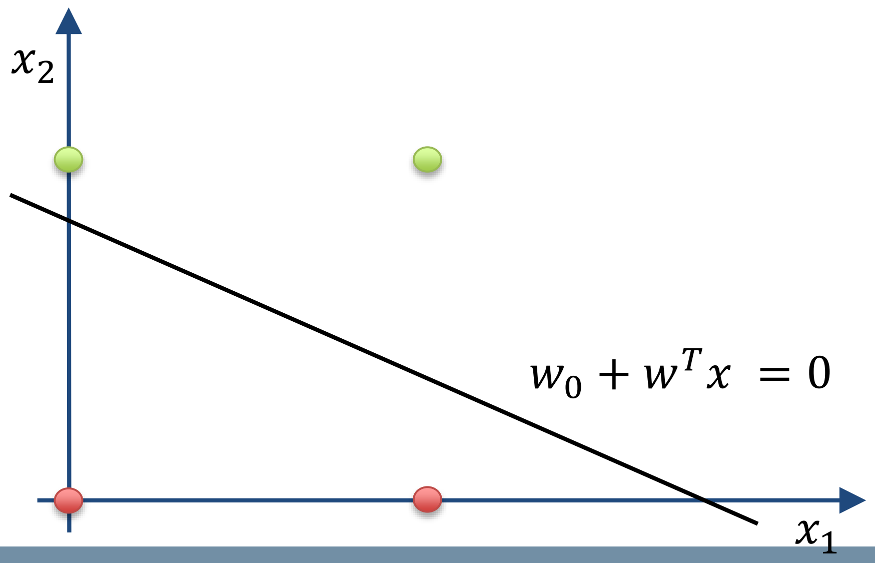
$$h_j(x|\mathrm{w}) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = Sign(w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I)$$

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

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

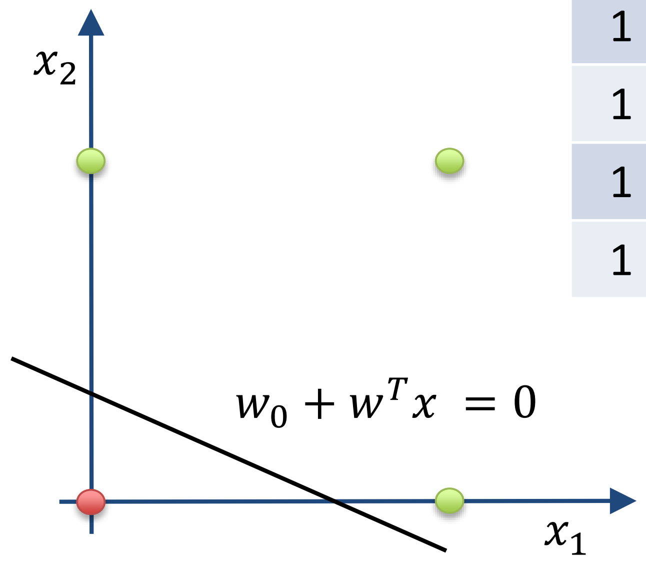
In 2D, this turns into

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

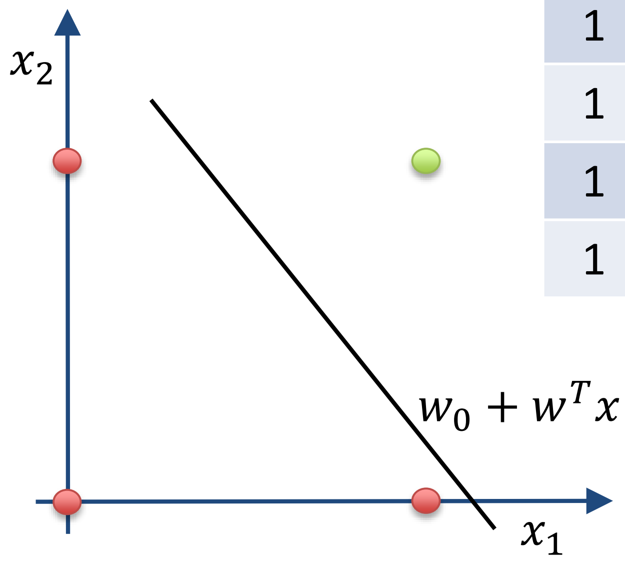


Boolean Operators Linear Boundaries

The previous boundary explains how the Perceptron implements the Boolean operators



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



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

What's about it? We had already Boolean operators

Hebbian Learning

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

(Donald Hebb, The Organization of Behavior, 1949)

Hebbian learning can be summarized by the following rule:

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

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

Start from a random initialization

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

Where we have:

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

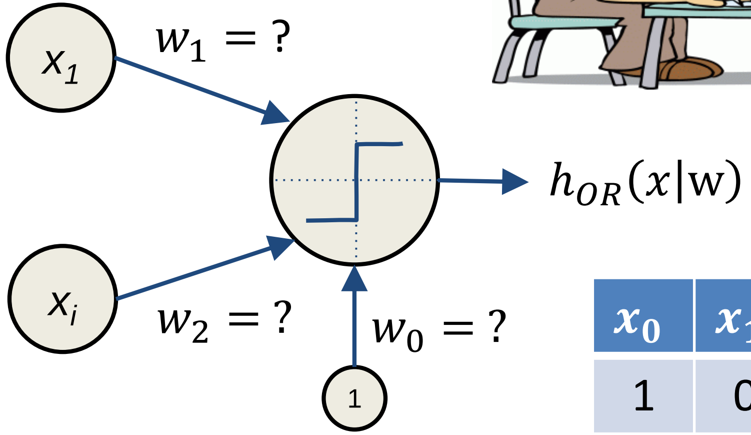
Perceptron Example

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

Does the procedure converge?

Does it always converge to the same sets of weights?

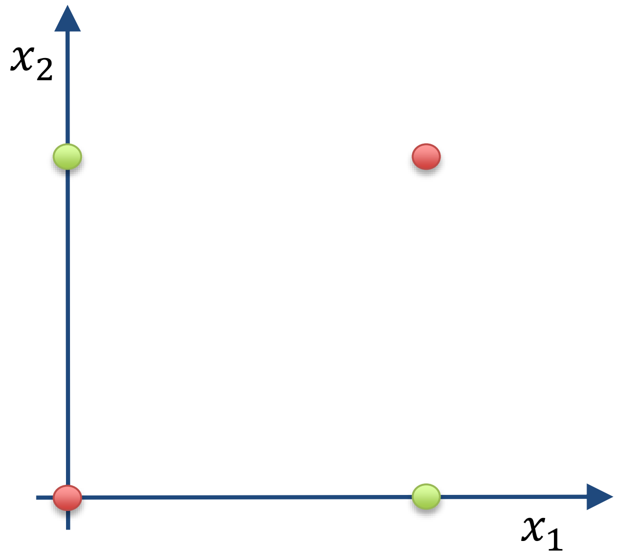


x_0	x_1	x_2	OR
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1

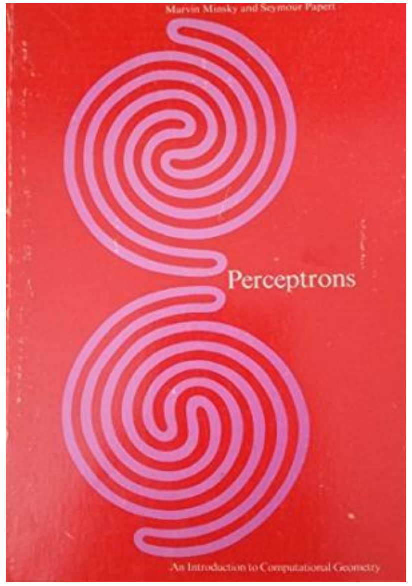
What can't you do with it?

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

x_0	x_1	x_2	XOR
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	-1



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

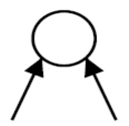
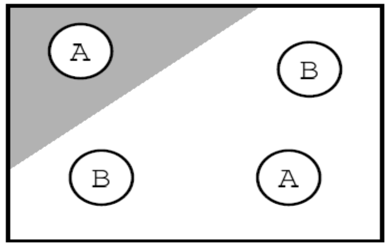
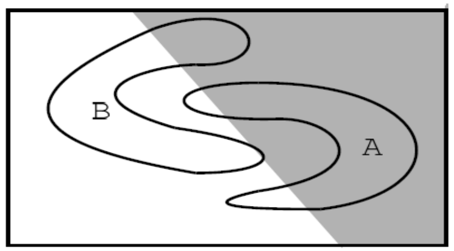
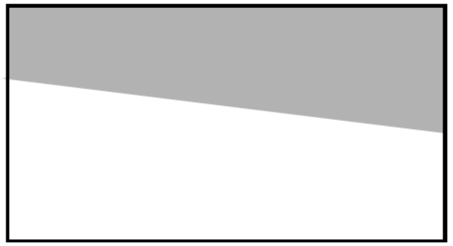
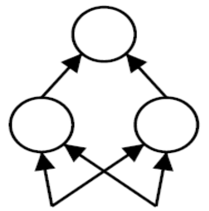
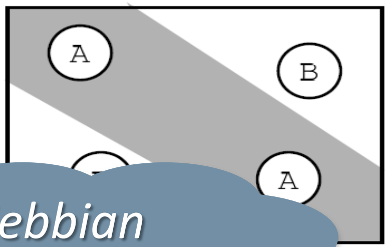
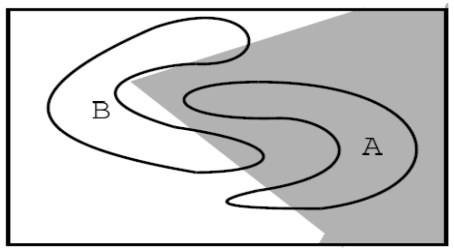
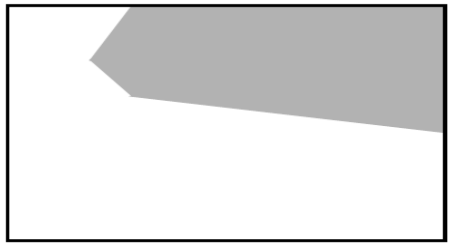
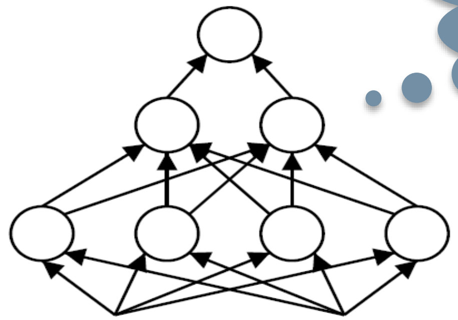
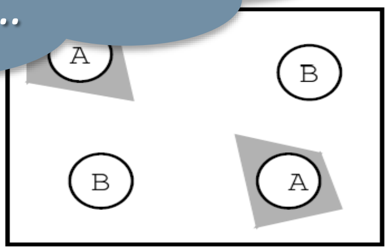
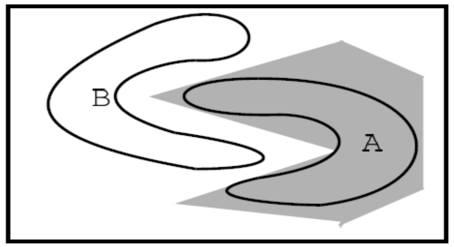
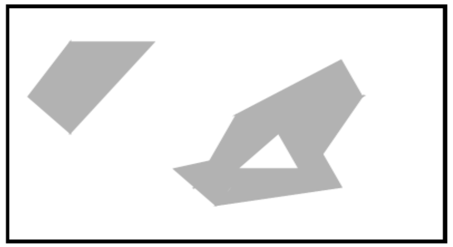

Marvin Minsky and Seymour Papert
Perceptrons
An Introduction to Computational Geometry

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?

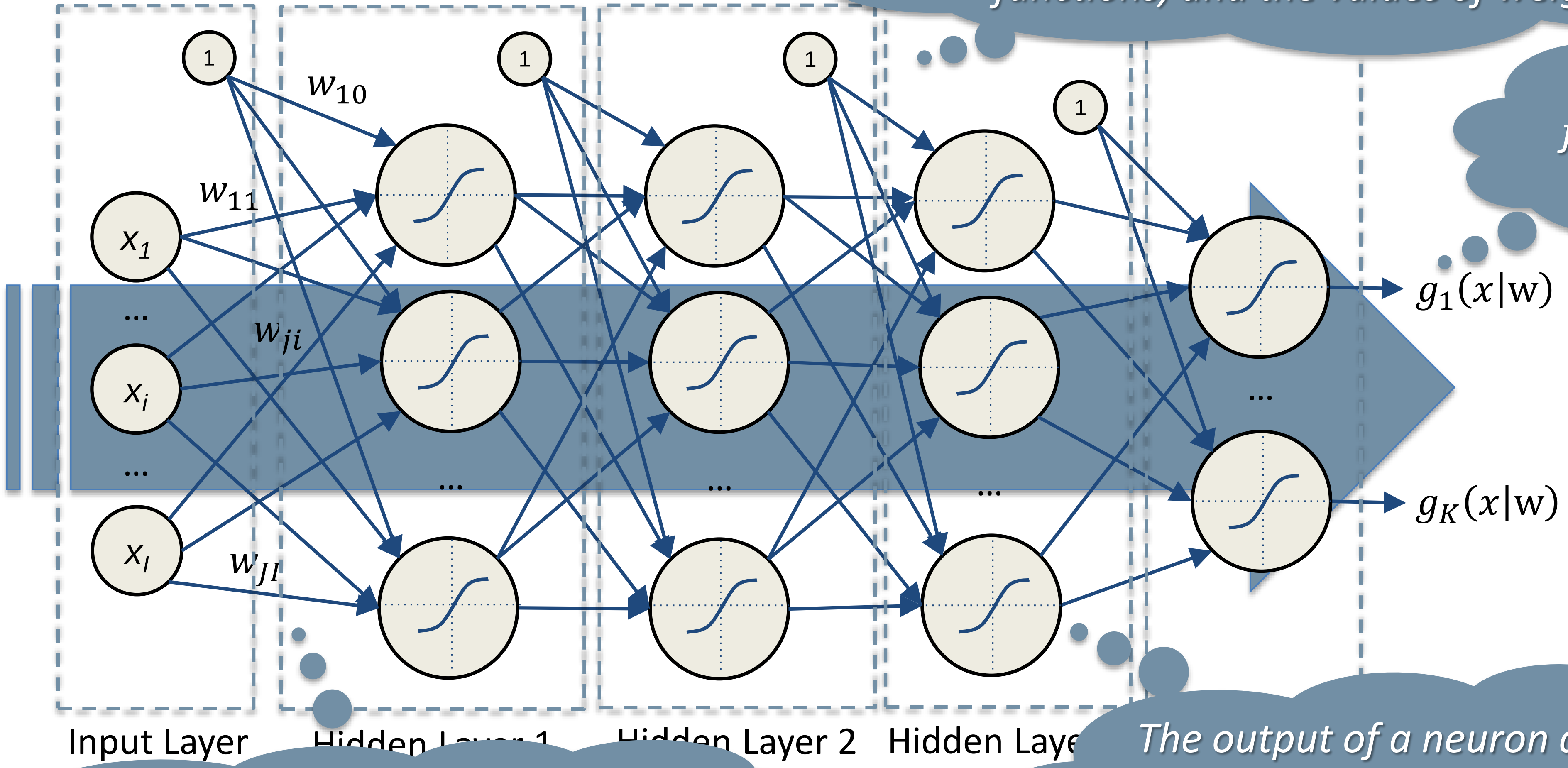
Topology	Type of Decision Region	XOR Problem	Classes with Meshed Regions	Most General Region Shapes
	Half bounded by hyperplanes			
	Convex Open or Clesed Regions			
	Arbitrary Regions (Complexity limited by the number of nodes)			

Unfortunately Hebbian learning does not work any more …

Layer Perceptrons

Feed Forward Neural Networks

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



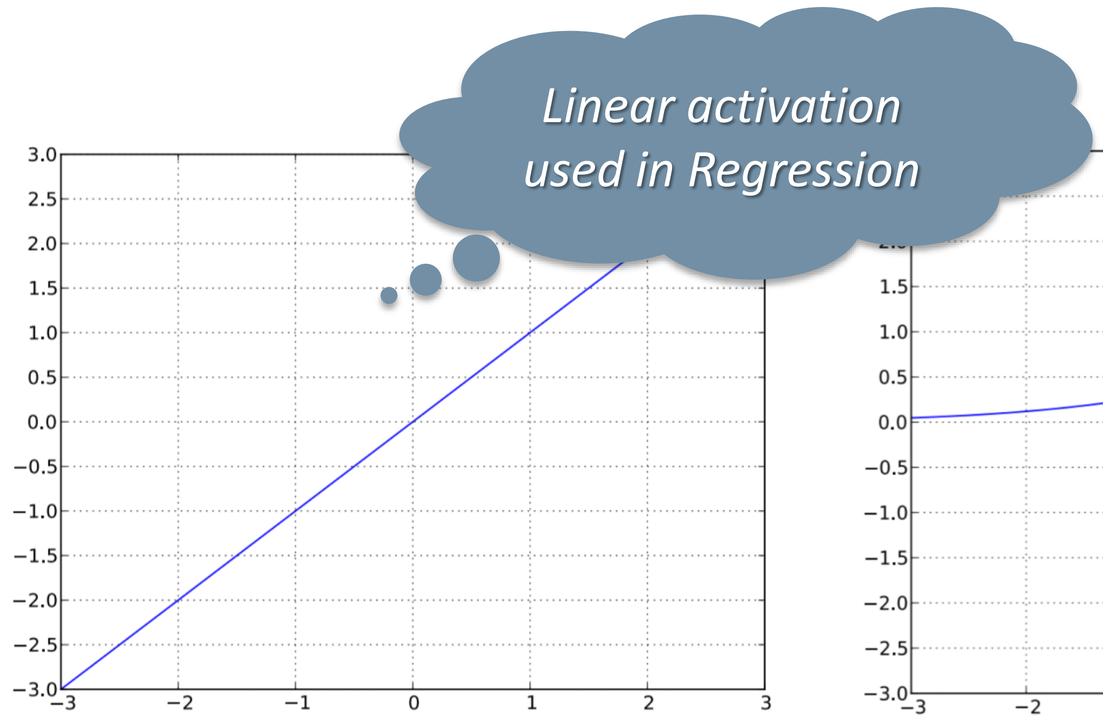
Activation functions must be differentiable

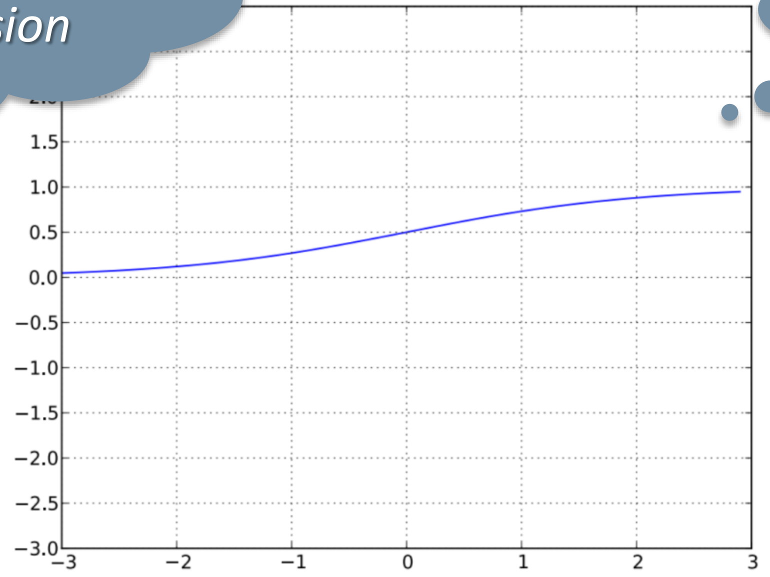
The output of a neuron depends only on the previous layers

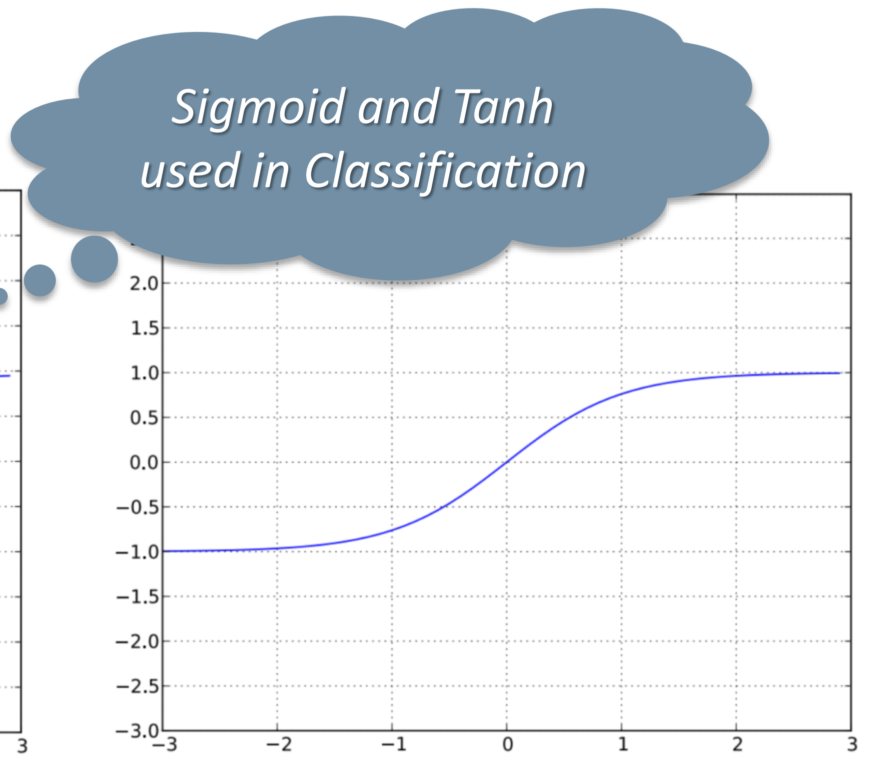
$$h^{(l)} = \left\{h_j^{(l)}\left(h^{(l-1)}, W^{(l)}\right)\right\}$$

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

Which Activation 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

$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$
$$g'(a) = 1 - g(a)^2$$

Output Layer in Regression and Classification

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

- Use a Linear activation function for the output neuron

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

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

«One hot» coding

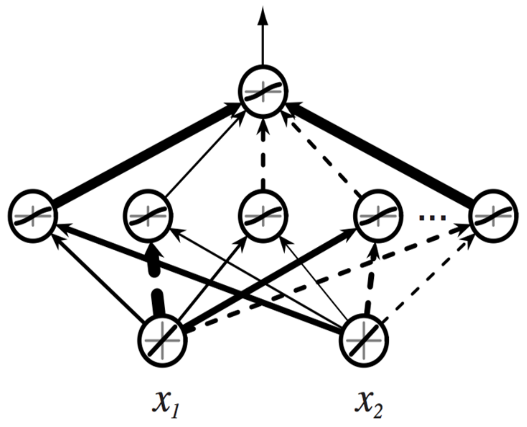
When dealing with multiple classes (K) use as many neuron as classes

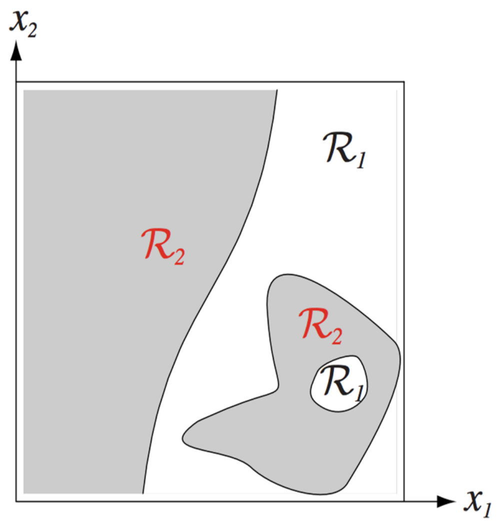
- Classes are coded as $\{\Omega_0 = [0\ 0\ 1], \Omega_1 = [0\ 1\ 0], \Omega_2 = [1\ 0\ 0]\}$
- Output neurons use a softmax unit $\mathrm{y_k} = \dfrac{\exp\left(h_k\left(\sum_i^I w_{ki} \cdot x_i\right)\right)}{\sum_j^J \exp\left(h_j\left(\sum_i^I w_{ji} \cdot x_i\right)\right)}$

Neural Networks are Universal Approximators

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

Universal approximation theorem
(Kurt Hornik, 1991)





Images from Hugo Larochelle's DL Summer School Tutorial

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

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

Classification requires just one extra layer …

Optimization and Learning

Recall about learning a model in regression and classification

- Given a training set

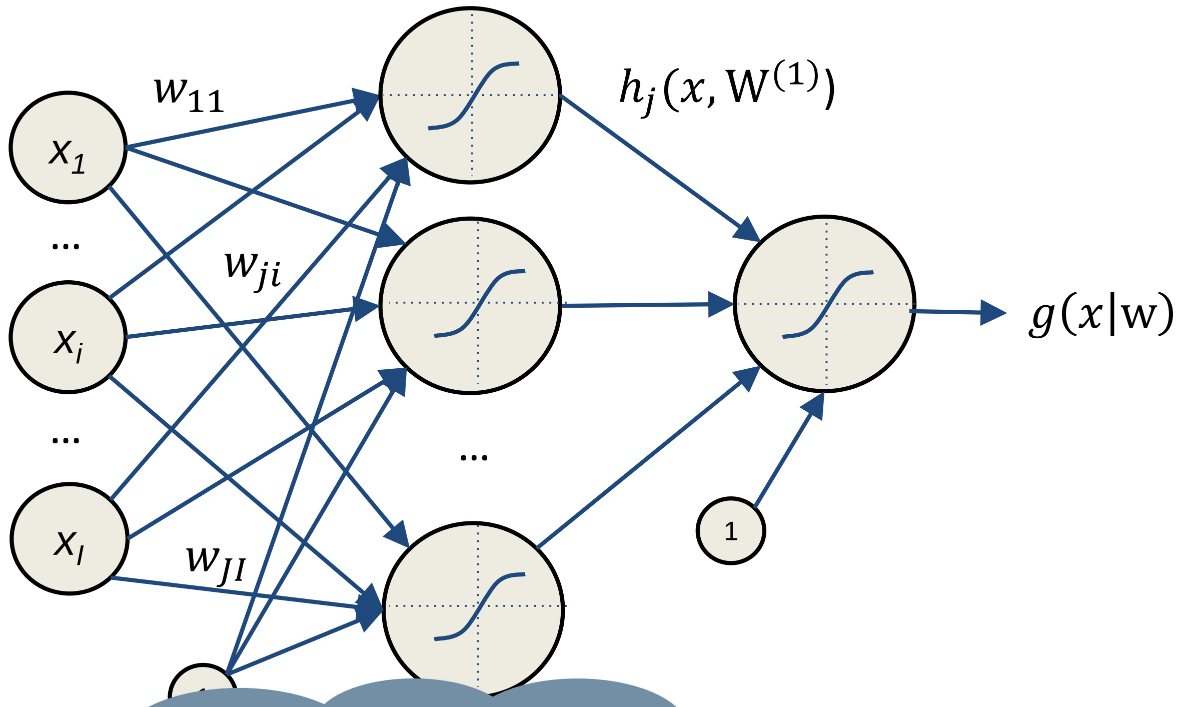
$$D = < x_1, t_1 > \cdots < x_N, t_N >$$

- We want to find the 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

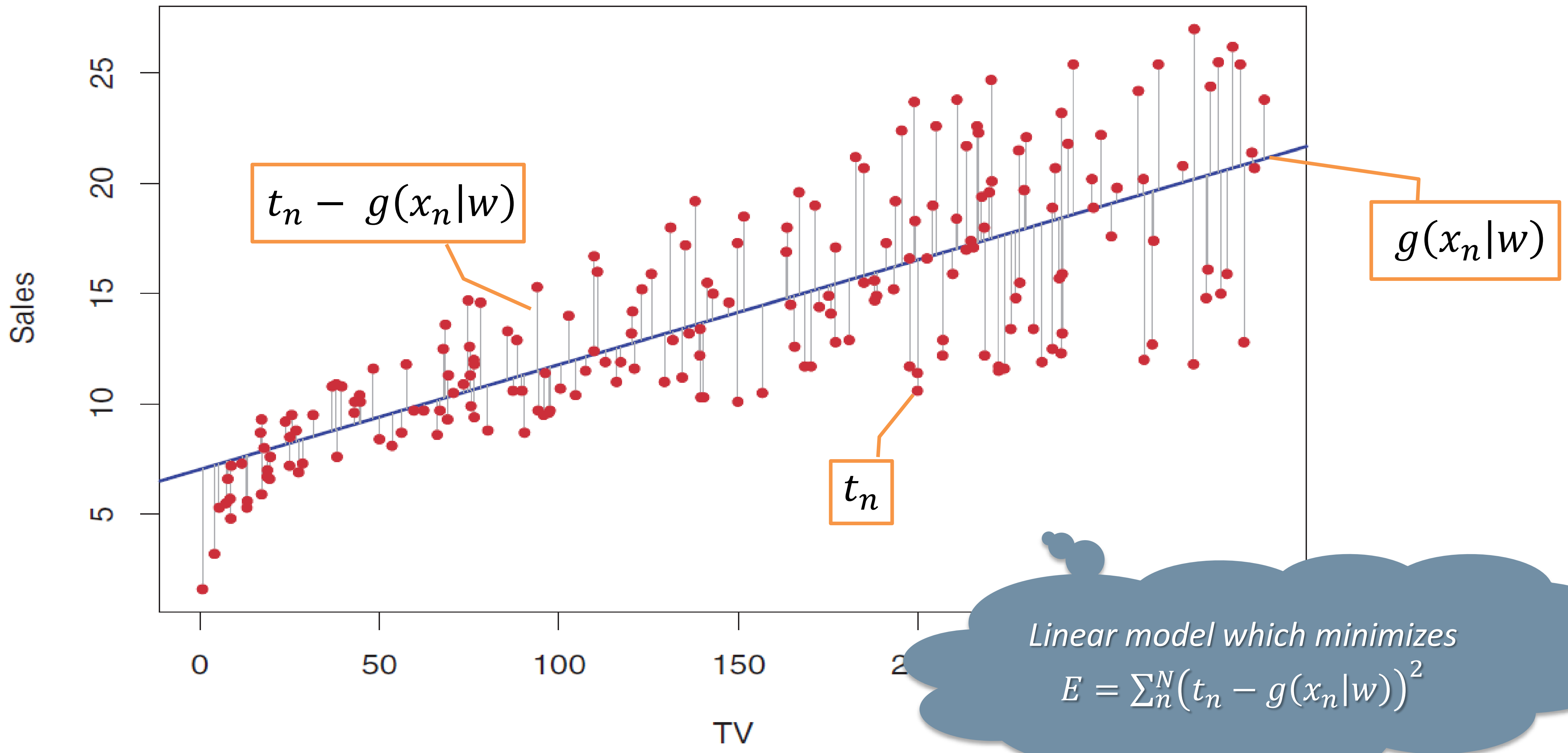
$$g(x_n|w) \sim t_n$$



For this you can minimize

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

Sum of Squared Errors



Linear model which minimizes

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

Non Linear Optimization 101

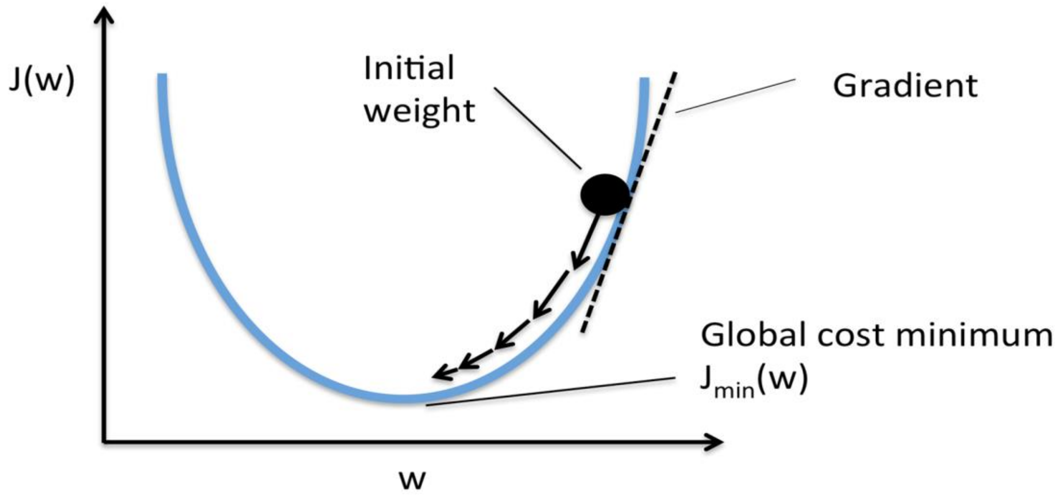
To find the minimum of a generic function, we compute the partial derivatives of the function and set them to zero

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

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

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

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



Gradient descent - Backpropagation

Finding the weighs of a Neural Network is a non linear mi

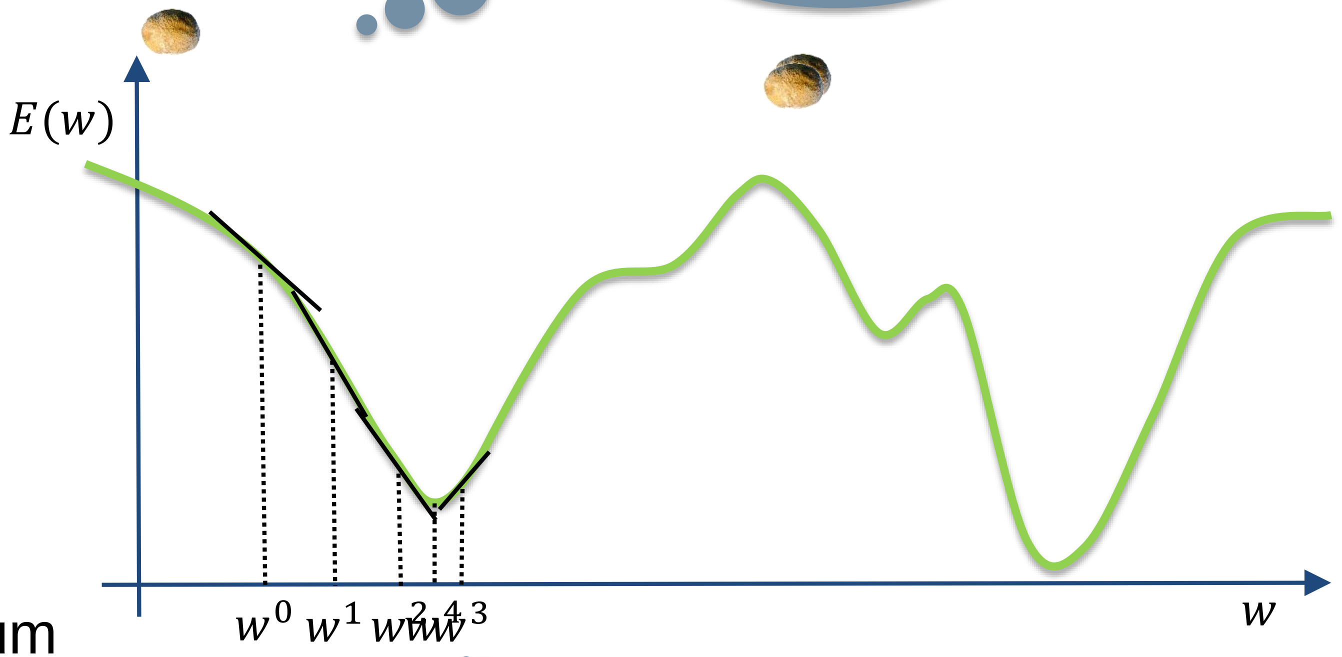
$$argmin_w \ E(w) = \sum_{n=1}^{N} (t_n - g(x_n, w))^2$$

We iterate from a initial configuration

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

To avoid local minima can use momentum

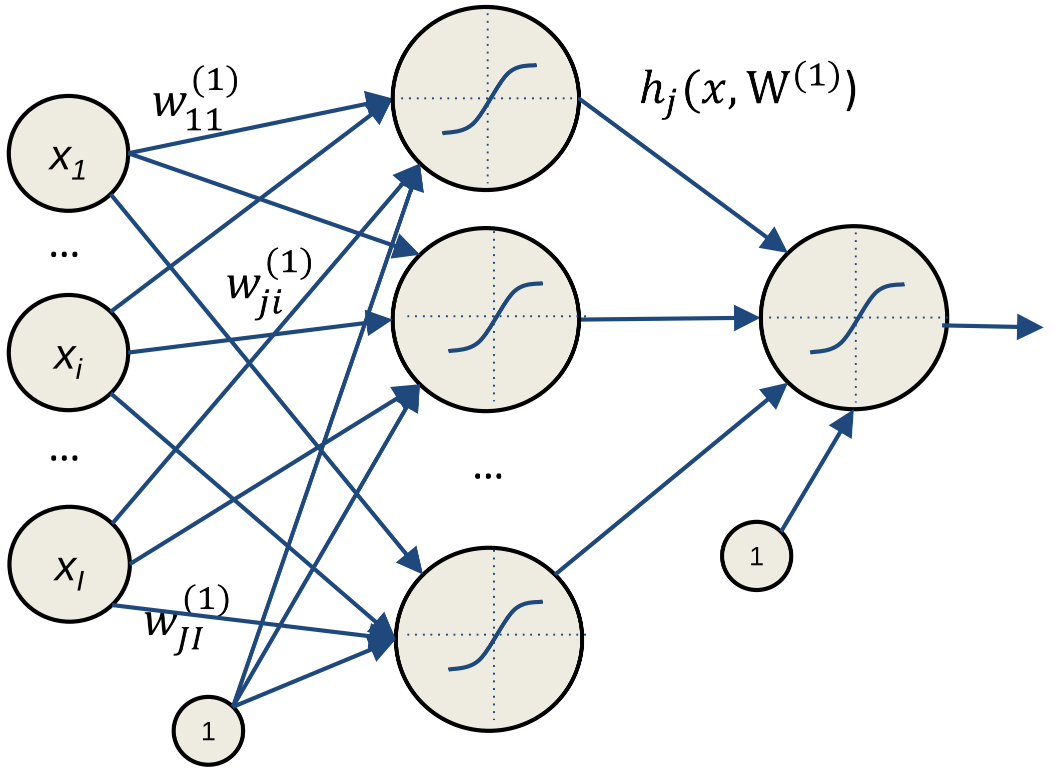
$$w^{k+1} = w^k - \eta \left. \frac{\partial E(w)}{\partial w} \right|_{w^k} - \alpha \left. \frac{\partial E(w)}{\partial w} \right|_{w^{k-1}}$$



Use multiple reastarts to seek for a proper global minimum.

It depends on where we start from

Gradient Descent Example

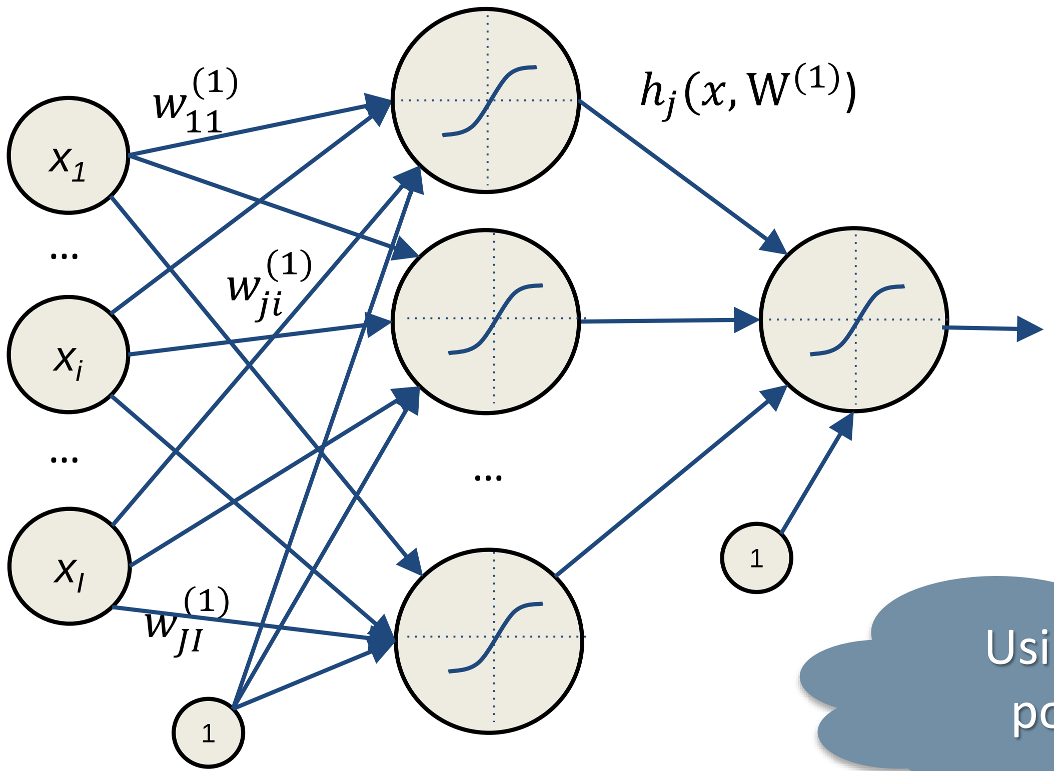


$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

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

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

Gradient Descent Example



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

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

Using all the data points (BATCH)

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

Backpropagation and Chain Rule (1)

Updating the weights can be done in parallel, locally, and it requires just two passes ...

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

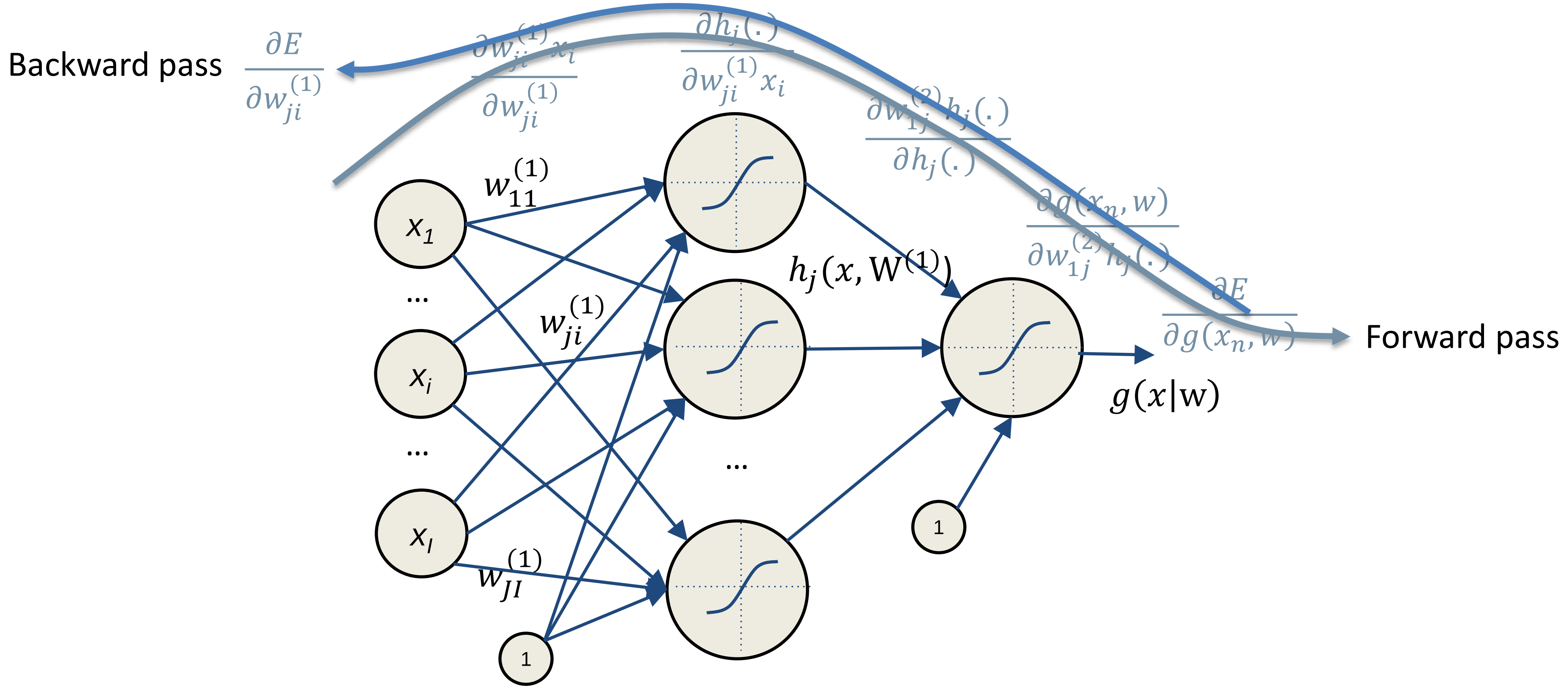
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = f'(y)g'(x) = f'\big(g(x)\big)g'(x)$$

The same holds for backpropagation

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

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

Backpropagation and Chain Rule (2)



Backward pass
$\frac{\partial E}{\partial w_{ji}^{(1)}}$
$\frac{\partial w_{ji}^{(1)} x_i}{\partial w_{ji}^{(1)}}$
$\frac{\partial h_j(.)}{\partial w_{ji}^{(1)} x_i}$
$\frac{\partial w_{1j}^{(2)} h_j(.)}{\partial h_j(.)}$
$\frac{\partial g(x_n, w)}{\partial w_{1j}^{(2)} h_j(.)}$
$\frac{\partial E}{\partial g(x_n, w)}$
Forward pass
x_1
x_i
x_I
...
$w_{11}^{(1)}$
$w_{ji}^{(1)}$
$w_{JI}^{(1)}$
1
$h_j(x, W^{(1)})$
$g(x|w)$

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

Gradient Descent Variations

What Hebbian learning had to do with this?

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

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

Hyperplanes Linear Algebra

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

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

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

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

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

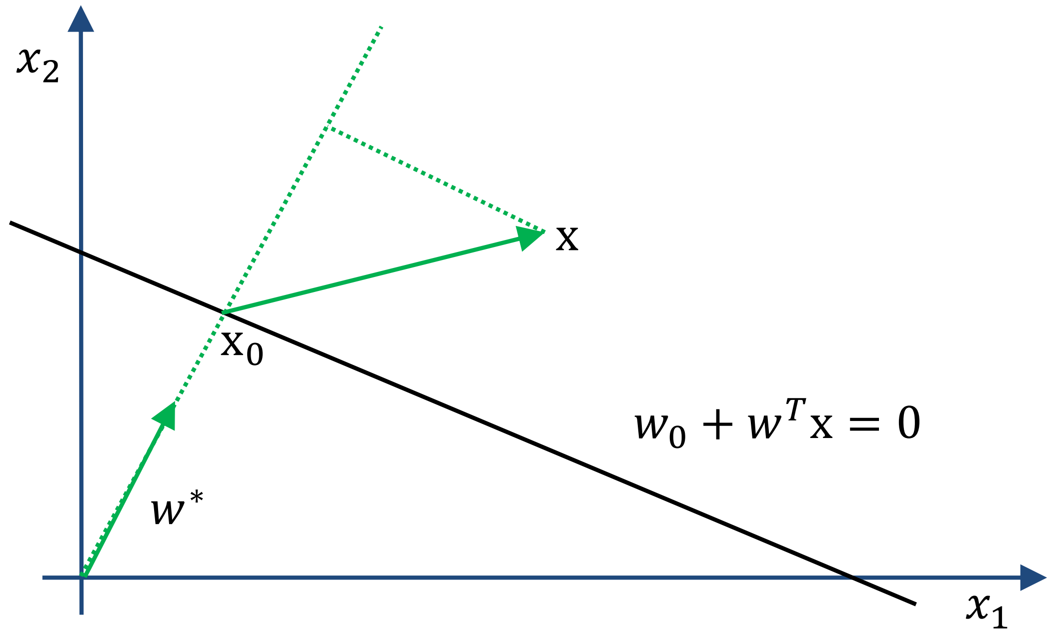
$$w^* = w/\|w\|$$

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

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

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

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



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

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 $\mathrm{w}^{\mathrm{T}}\mathrm{x} + \mathrm{w}_0 < 0$
- For an output with -1 we have the opposite

The goal becomes minimizing

Set of points misclassified

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

This is non negative and proportional to the distance of the misclassified points from

$$\mathrm{w}^{\mathrm{T}}\mathrm{x} + \mathrm{w}_0 = 0$$

Perceptron Learning Algorithm (2/2)

Let’s minimize by stochastic gradient descend the error function

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

The gradients with respect to the model parameters are

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

Stochastic gradient descent applies for each misclassified point

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

Hebbian learning implements Stocastic Gradient Descent

How to Chose the Error Function?

We have observed different error functions so far

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

Sum of Squared Errors

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

Perceptron Classification Error

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

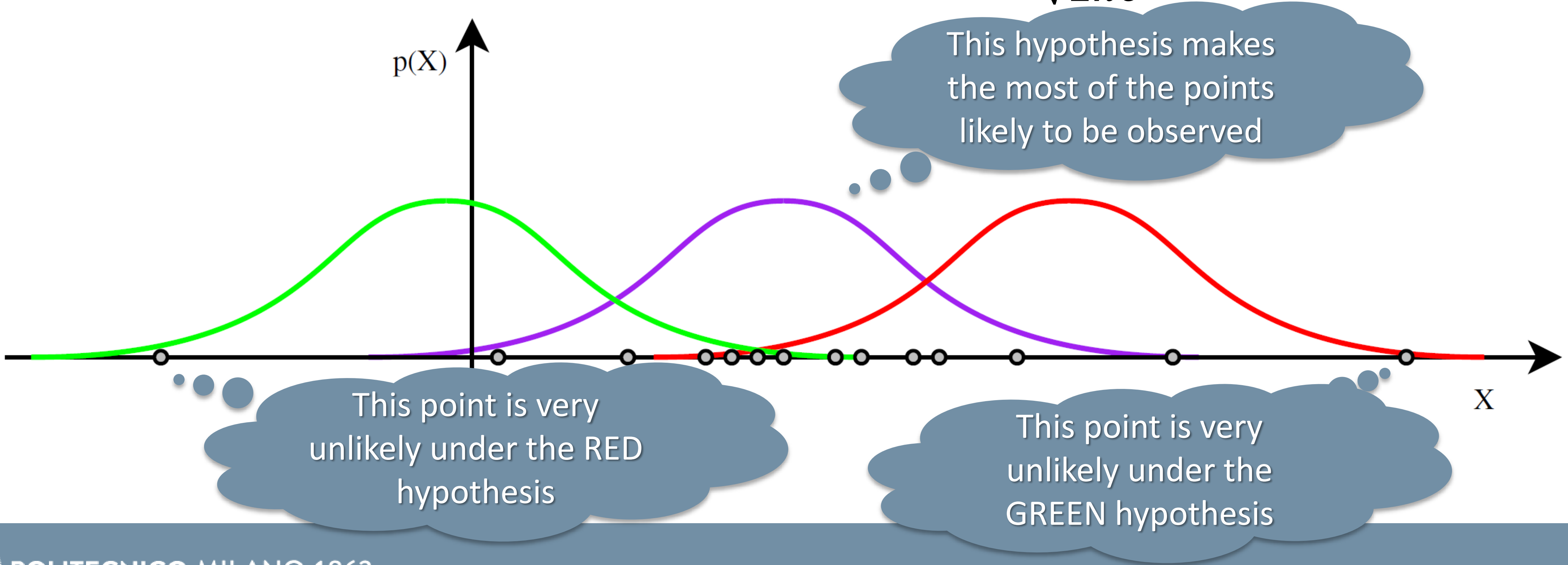
- Exploit background knowledge on the task and the model, e.g., Perceptron
- Use all your knowledge/assumptions about the data distribution
- Use your creativity!

This requires lots of trial and errors ...

A Note on Maximum Likelihood Estimation

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

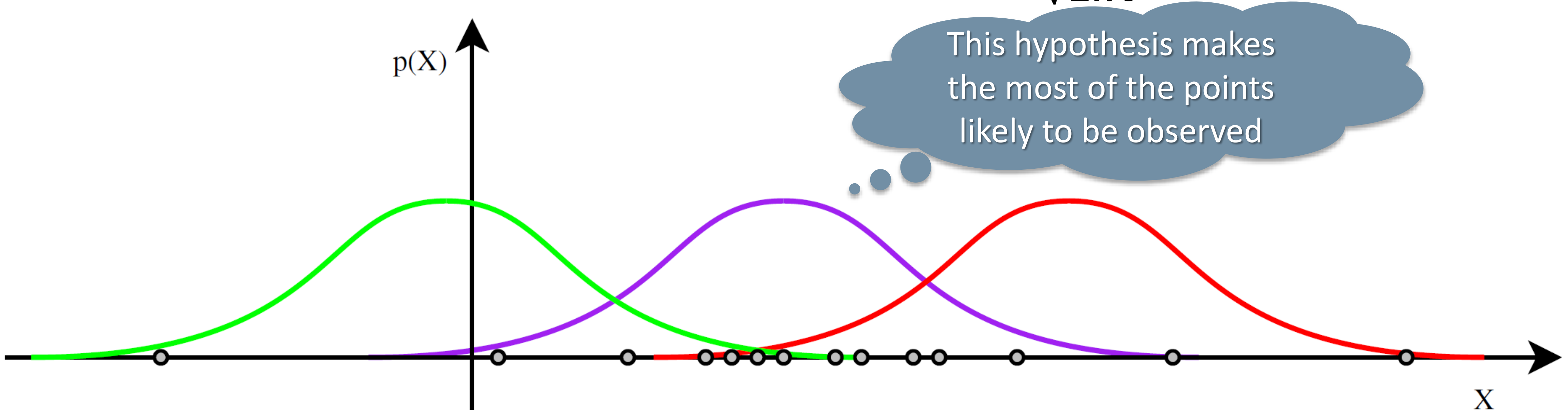
$$x_1, x_2, \ldots, x_N \sim N(\mu, \sigma^2) \qquad p(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



A Note on Maximum Likelihood Estimation

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

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


This hypothesis makes the most of the points likely to be observed
p(X)
X

<u>Maximum Likelihood</u>: Chose the parameters which maximize the data probability

Maximum Likelihood Estimation: The Recipe

Let $\theta = \left(\theta_1, \theta_2, \dots, \theta_p\right)^T$ a vector of parameters, find the MLE for θ by knowing $p(Data|\theta)$:

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

Optional

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

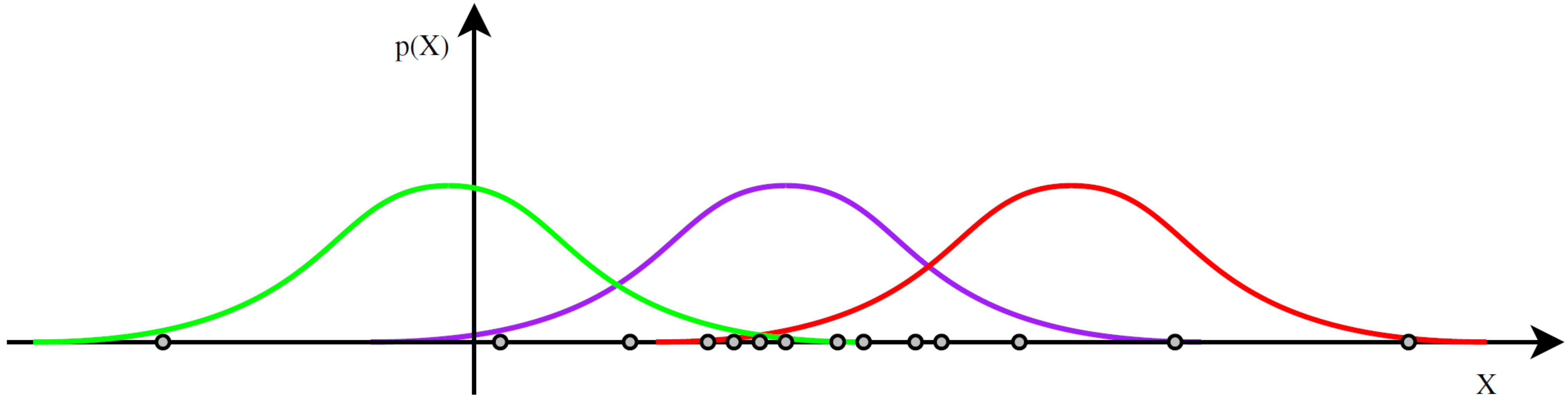
- Analytical Techniques (i.e., solve the equations)
- Optimizaion Techniques (e.g., Lagrange mutipliers)
- Numerical Techniques (e.g., gradient descend)

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

Maximum Likelihood Estimation Example

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

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



Find the Maximum Likelihood Estimator for μ

Maximum Likelihood Estimation Example

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

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

- 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 some i.i.d. samples coming from a Gaussian distribution with known σ^2

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

- 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 some i.i.d. samples coming from a Gaussian distribution with known σ^2

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

- 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 some i.i.d. samples coming from a Gaussian distribution with known σ^2

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

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

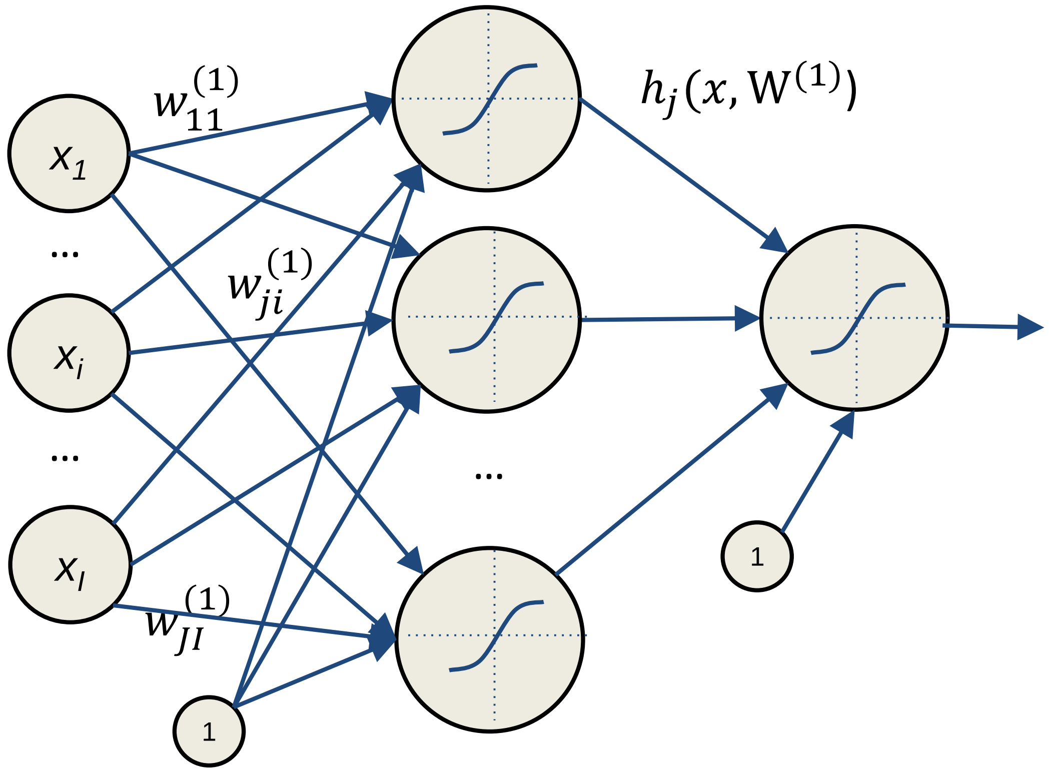
$$-\frac{1}{2\sigma^2}\sum_n^N 2(x_n - \mu) = 0$$

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

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

Let's apply this all to Neural Networks!

Neural Networks for Regression

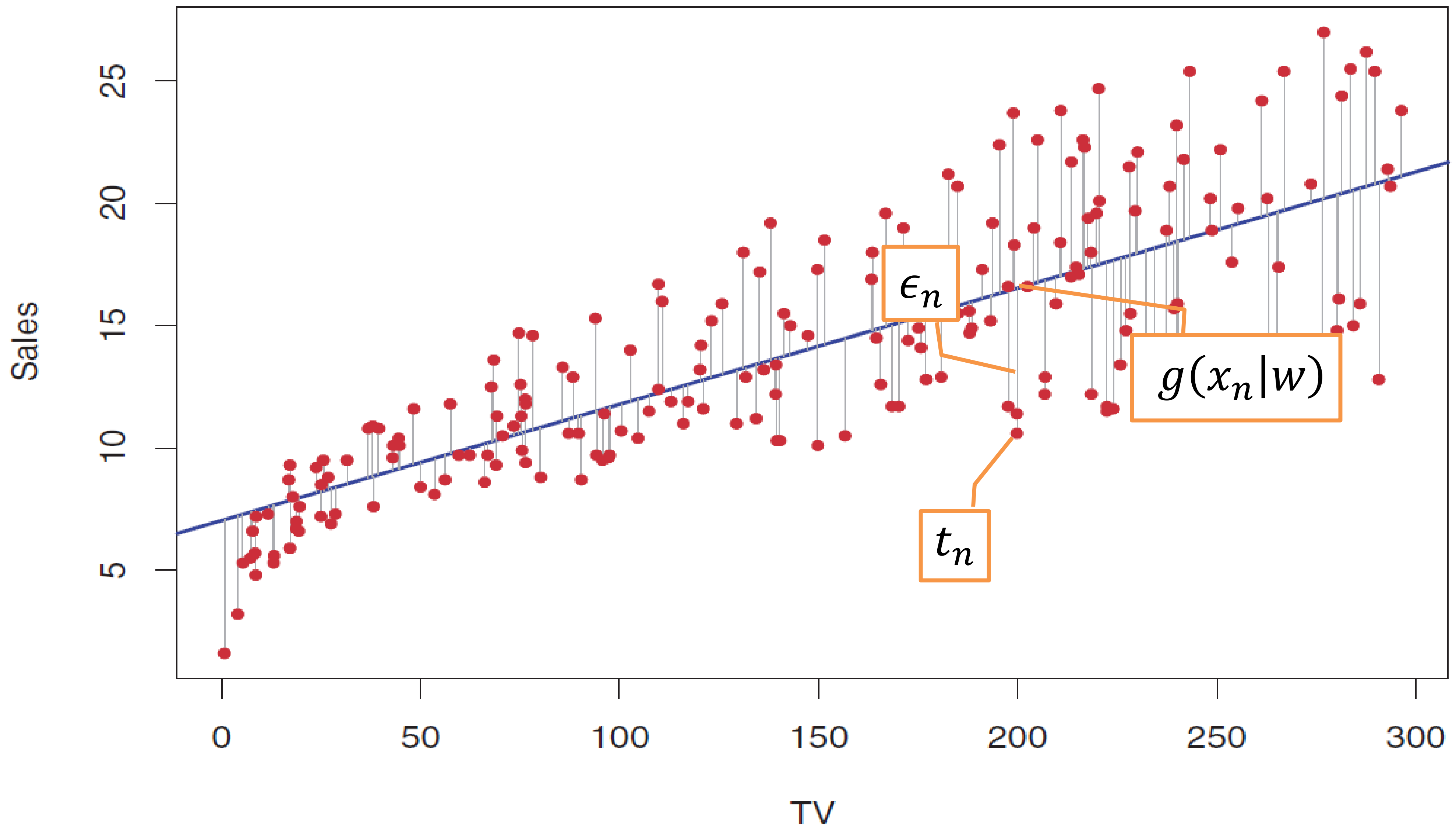


$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

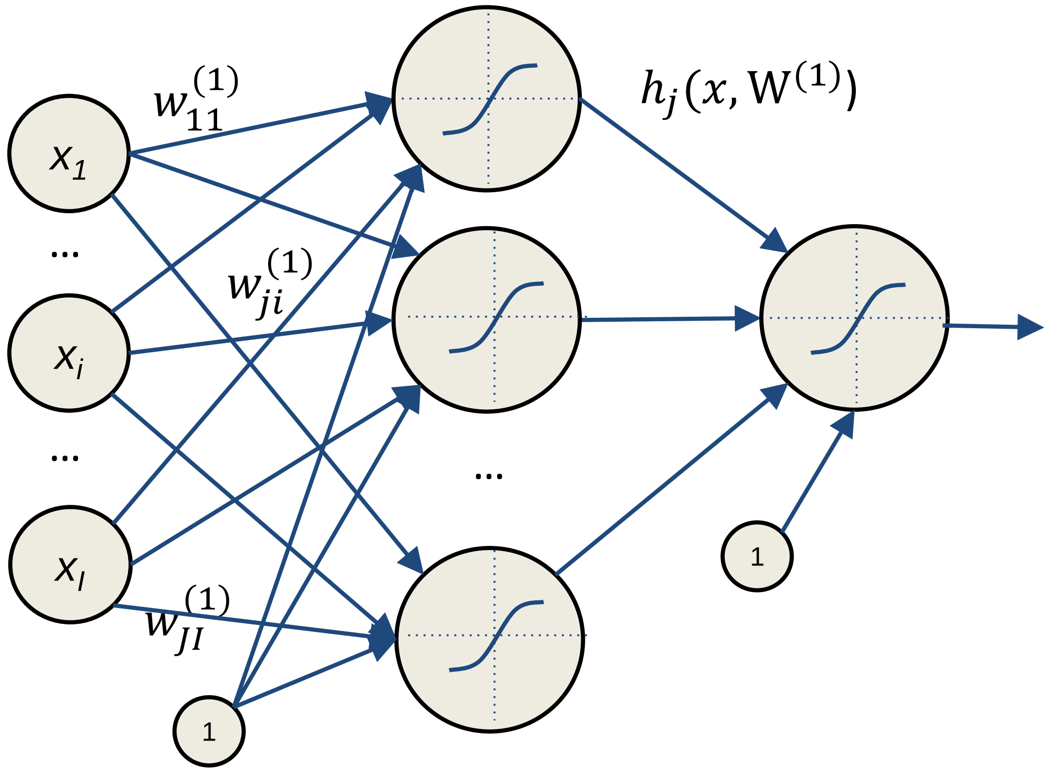
Goal: approximate a target function t having a finite set of N observations

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

Statistical Learnig Framework



Neural Networks for Regression



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

Goal: approximate a target function t having a finite set of N observations

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

Maximum Likelihood Estimation for Regression

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

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

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

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

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

Maximum Likelihood Estimation for Regression

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

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

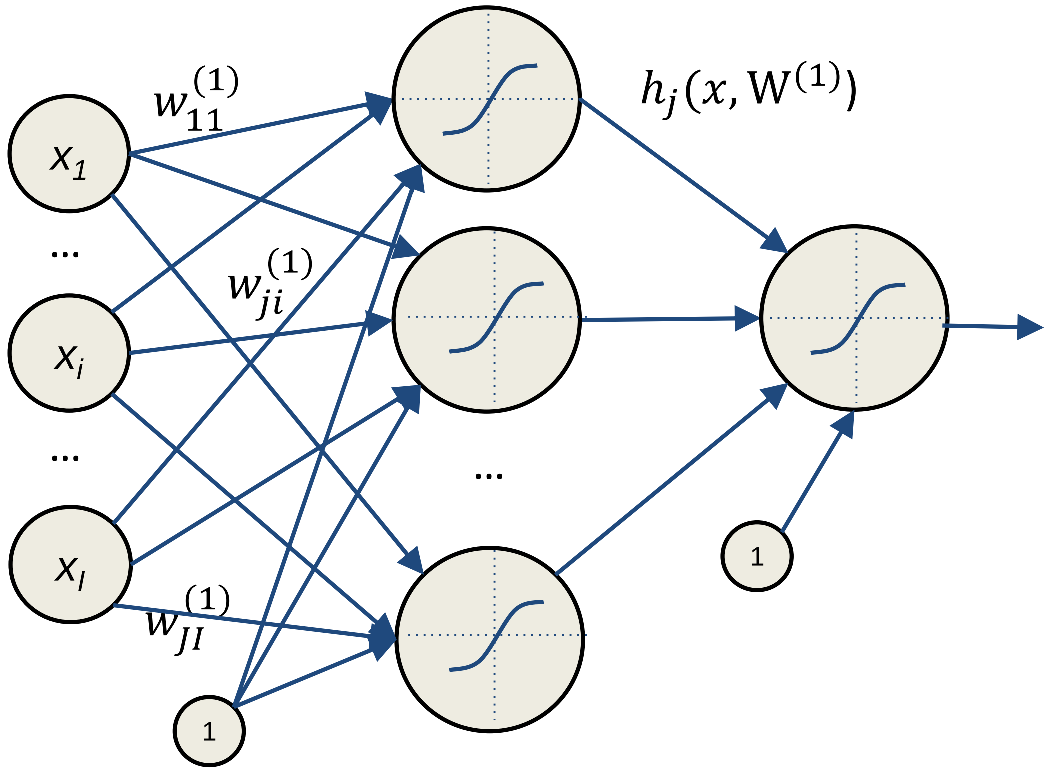
Look for the weights which maximixe the likelihood

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

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

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

Neural Networks for Classification



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

Goal: approximate a posterior probability t having a finite set of N observations

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

Maximum Likelihood Estimation for Classification

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

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

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

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

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

Maximum Likelihood Estimation for Classification

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

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

Look for the weights which maximize the likelihood

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

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

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

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

POLITECNICO
MILANO 1863

Cognitive Robotics
2018/2019

Neural Networks Training

Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

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)

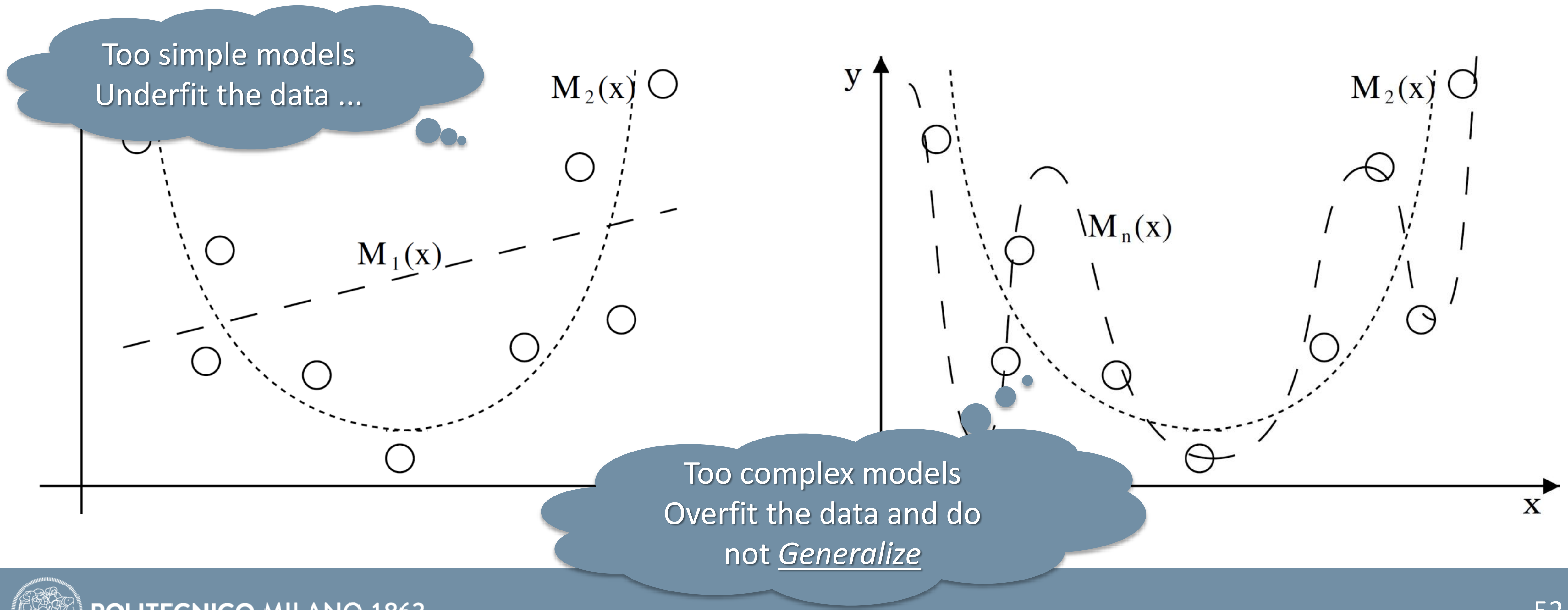
Regardless of what function we are learning, a single layer can do it …

- … 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!

“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



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

- 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

Done for training on small datasets

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

Cross-validation Variations

Cross-validation uses training data itselves to estimate the error on new data

- When enough data available use an hold out set and perform validation

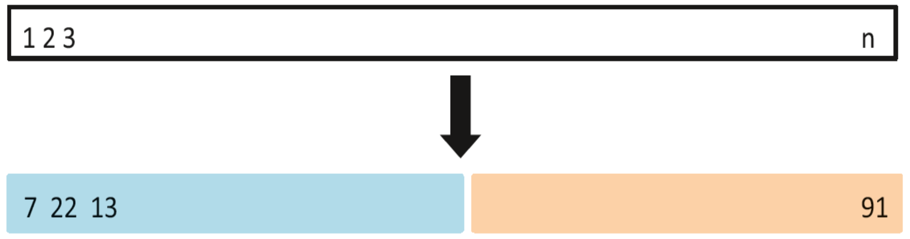


FIGURE 5.1. *A schematic display of the validation set approach. A set of n observations are randomly split into a training set (shown in blue, containing observations 7, 22, and 13, among others) and a validation set (shown in beige, and containing observation 91, among others). The statistical learning method is fit on the training set, and its performance is evaluated on the validation set.*

Cross-validation Variations

Cross-validation uses training data itselves to estimate the error on new data

- When enough data available use an hold out set and perform validation
- When not too many data available use leave-one-out cross-validation (LOOCV)

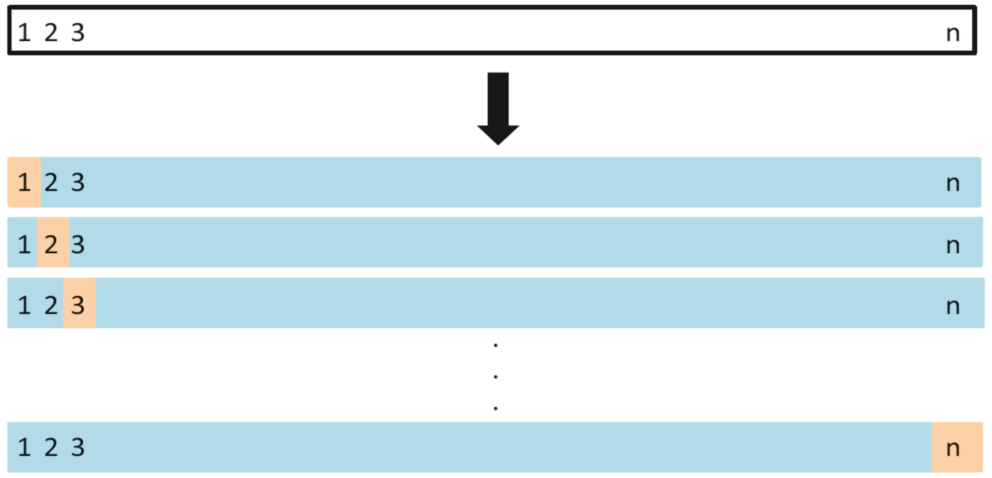


FIGURE 5.3. *A schematic display of LOOCV. A set of n data points is repeatedly split into a training set (shown in blue) containing all but one observation, and a validation set that contains only that observation (shown in beige). The test error is then estimated by averaging the n resulting MSE's. The first training set contains all but observation 1, the second training set contains all but observation 2, and so forth.*

Cross-validation Variations

Cross-validation uses training data itselves to estimate the error on new data

- When enough data available use an hold out set and perform validation
- When not too many data available use leave-one-out cross-validation (LOOCV)
- Use k-fold cross-validation for a good trade-off (sometime better than LOOCV)

$$\hat{e}_k = \frac{1}{N} \sum_{n_k \in N_k} E\left(x_{n_k} | w\right)$$

$$\hat{E} = \frac{1}{K} \sum_{k}^{K} \hat{e}_k$$

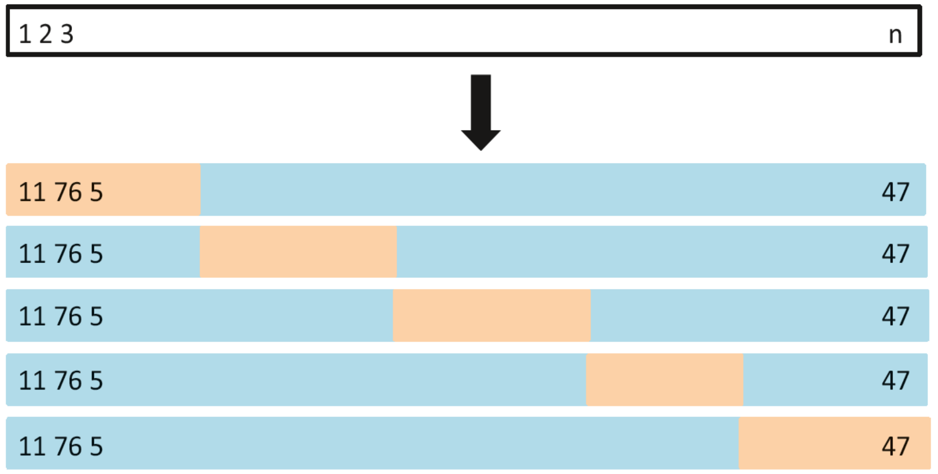


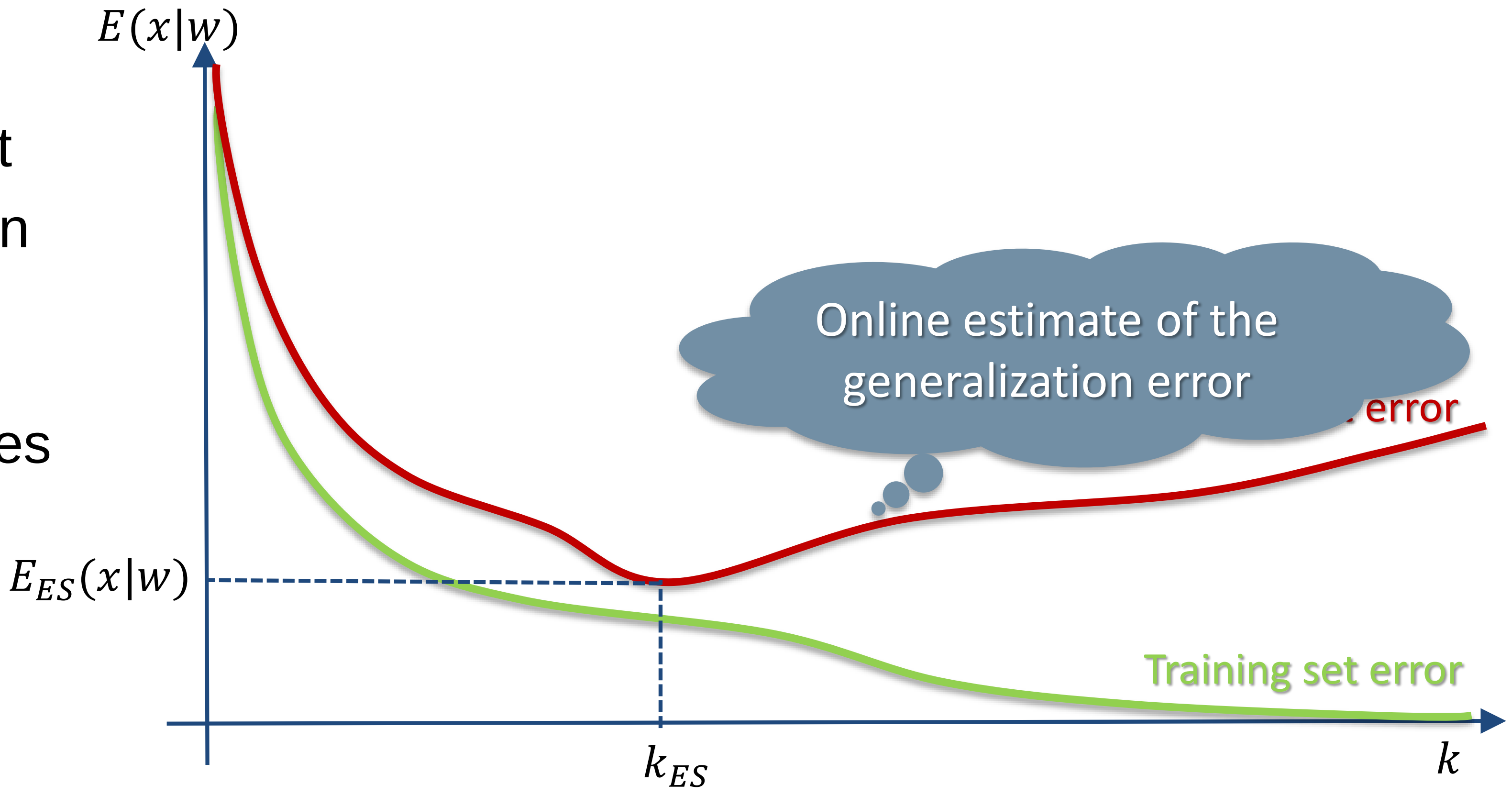
FIGURE 5.5. *A schematic display of 5-fold CV. A set of n observations is randomly split into five non-overlapping groups. Each of these fif validation set (shown in beige), and the remainder as a trainin blue). The test error is estimated by averaging the five resulting M*

What do I do with all these models?

Early Stopping: Limiting Overfitting by Cross-validation

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 ...

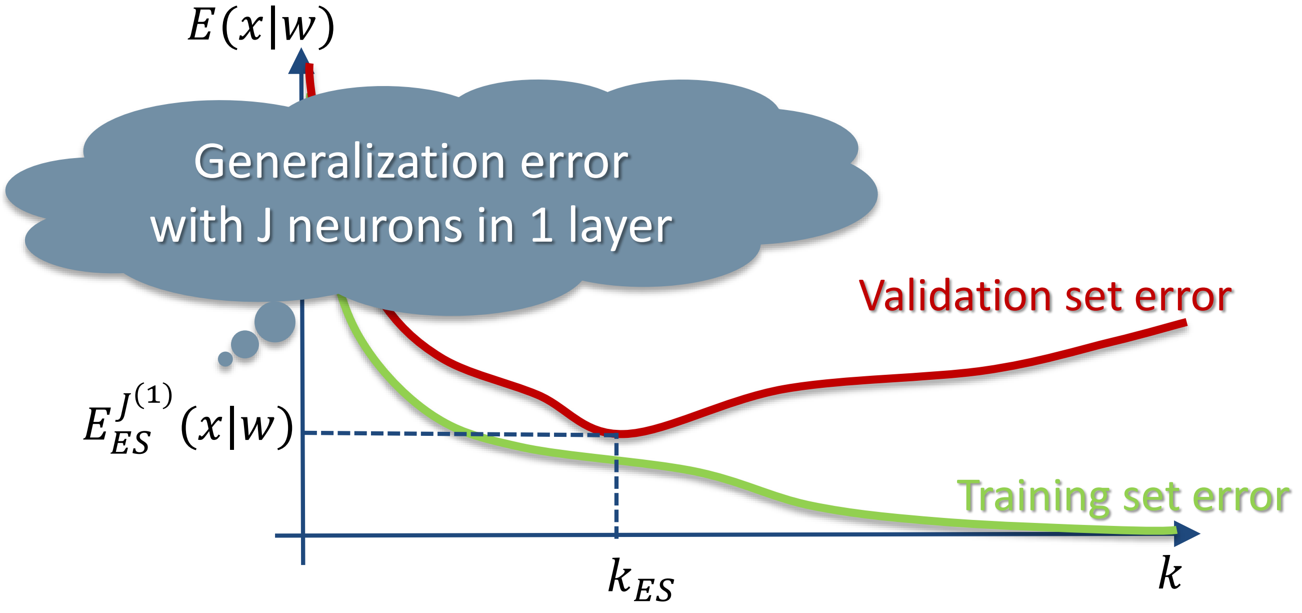
- Hold out some data
- Train on the training set
- Perform cross-validation on the hold out set
- Stop the train when validation error increases

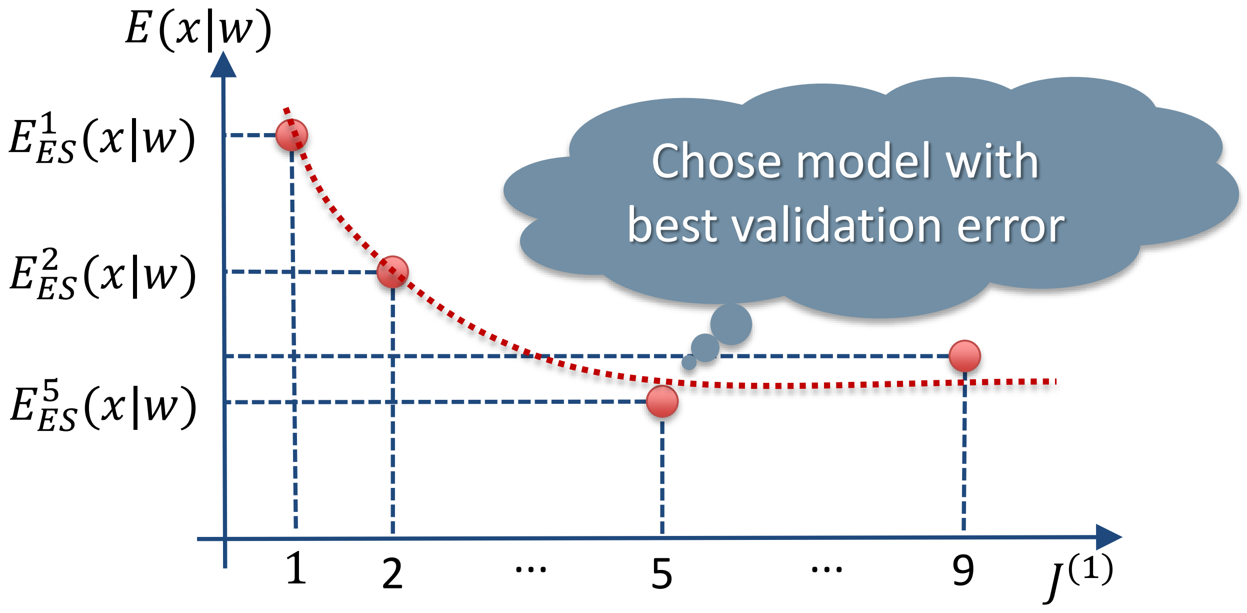


Cross-validation and Hyperparameters Tuninig

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., when we learn from data a model to chose hyperparameters





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.

So far we have maximized the data likelihood:

Maximum Likelihood

$$w_{MLE} = argmax_w \, P(D|w)$$

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

Make assumption on parameters (a-priori) distribution

Maximum A-Posteriori

$$\begin{aligned} w_{MAP} &= argmax_w \, P(w|D) \\ &= argmax_w \, P(D|w) \cdot P(w) \end{aligned}$$

Small weights have been observed to improve generalization of neural networks:

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

Weight Decay: Limiting Overfitting by Weights Regularization

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

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

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

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

Here it comes another loss function!!!

Recall Cross-validation and Hyperparameters Tuninig

You can use cross-validaton to select the proper γ:

- Split data in training and validation sets
- Minimize for different values of γ

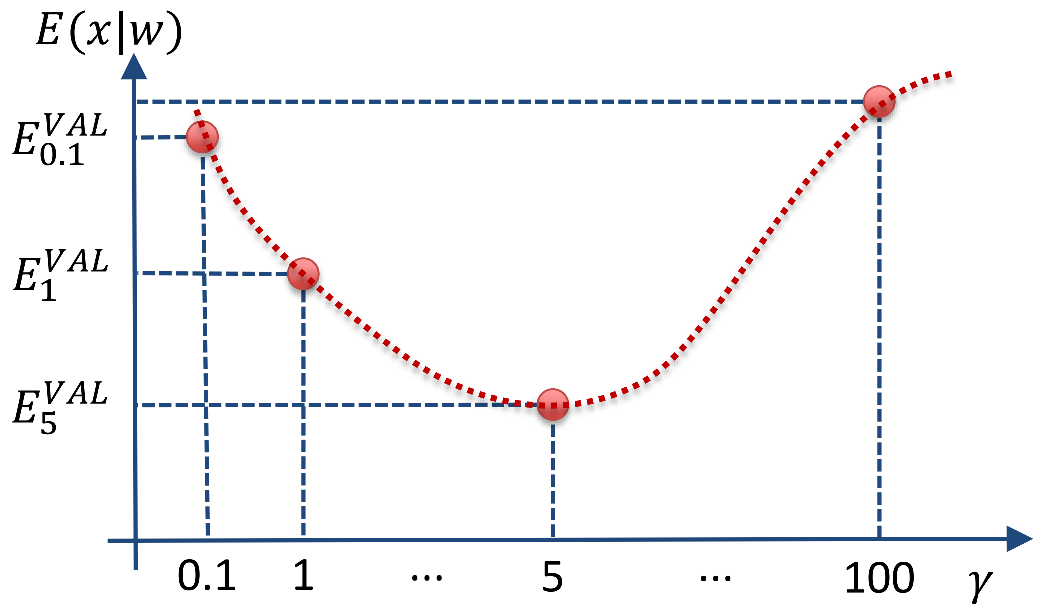
$$E_{\gamma}^{TRAIN} = \sum_{n=1}^{N_{TRAIN}} \left(t_n - g(x_n|w)\right)^2 + \gamma \sum_{q=1}^{Q} \left(w_q\right)^2$$

- Evaluate the model

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

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

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

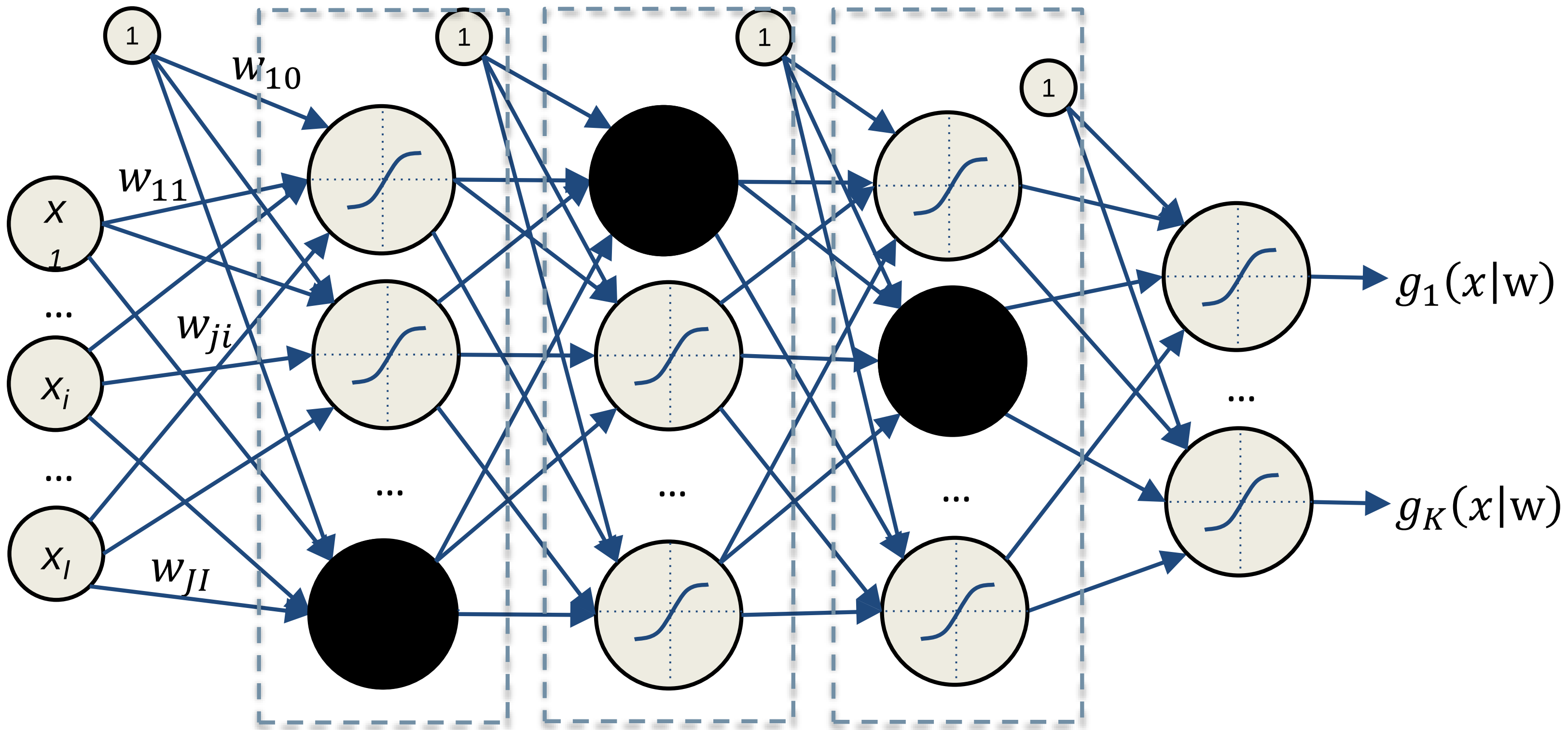


Chose $\gamma^* = 5$ with best validation error

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force them 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., $p_j^{(l)} = 0.3$



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

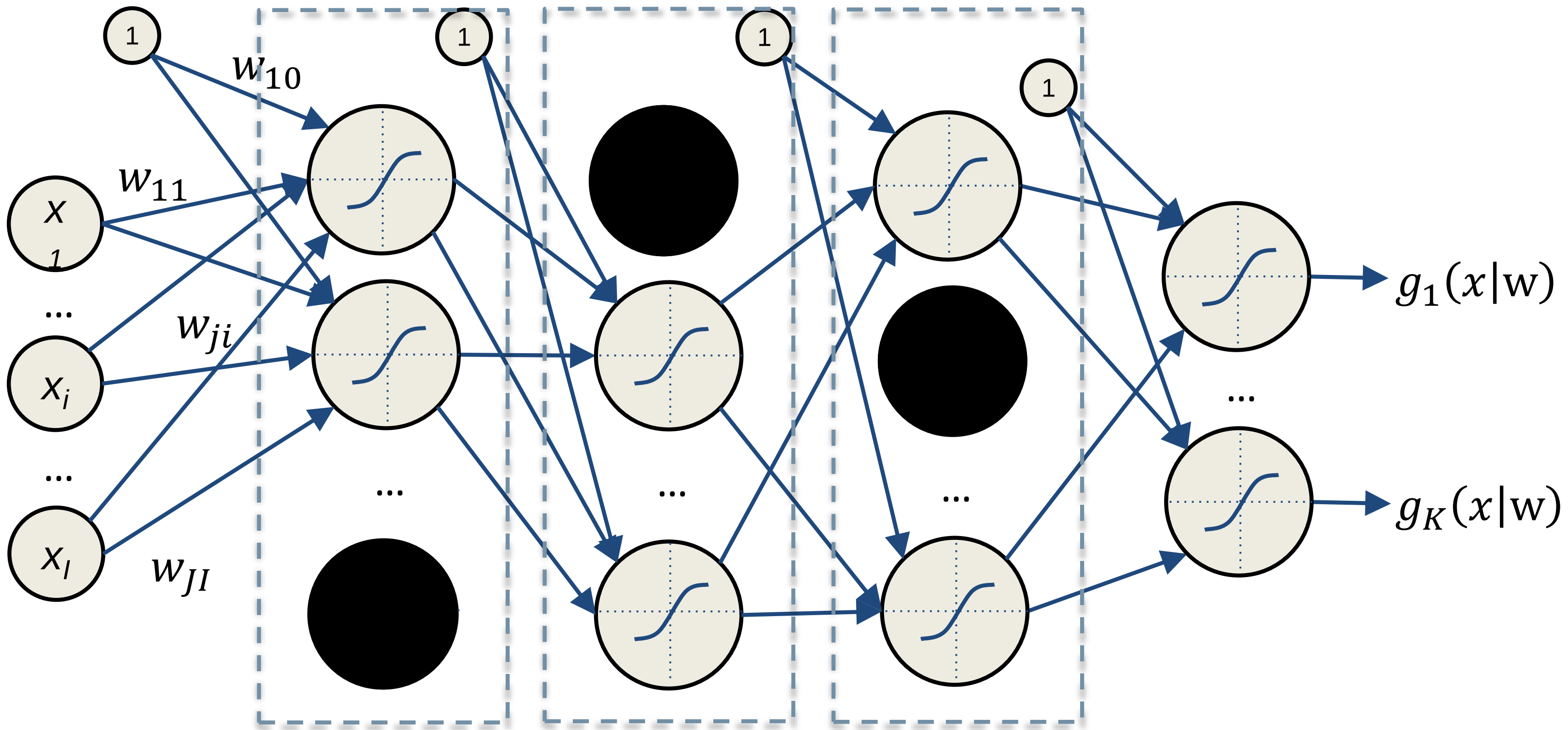
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

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

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force them 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., $p_j^{(l)} = 0.3$



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

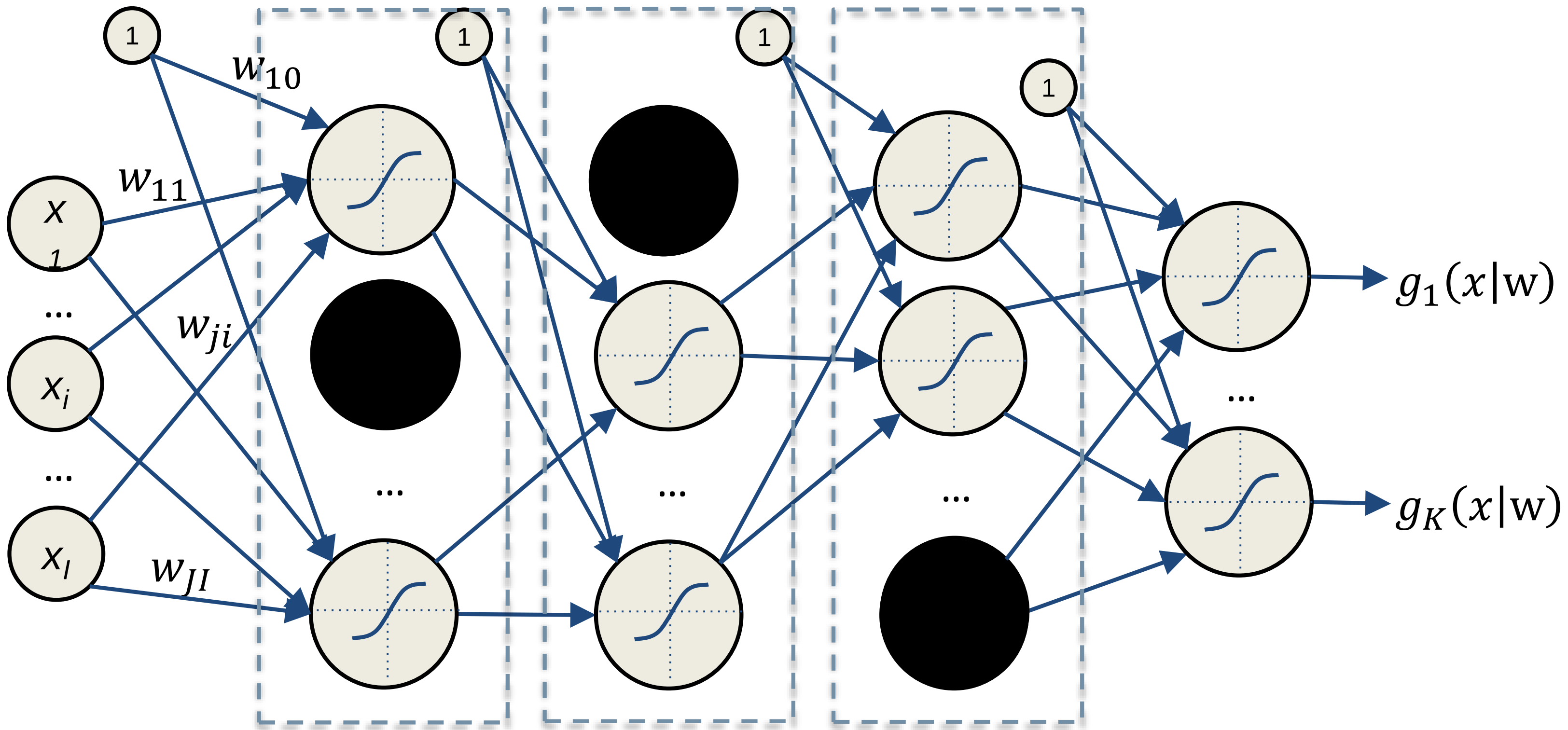
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

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

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force them 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., $p_j^{(l)} = 0.3$



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

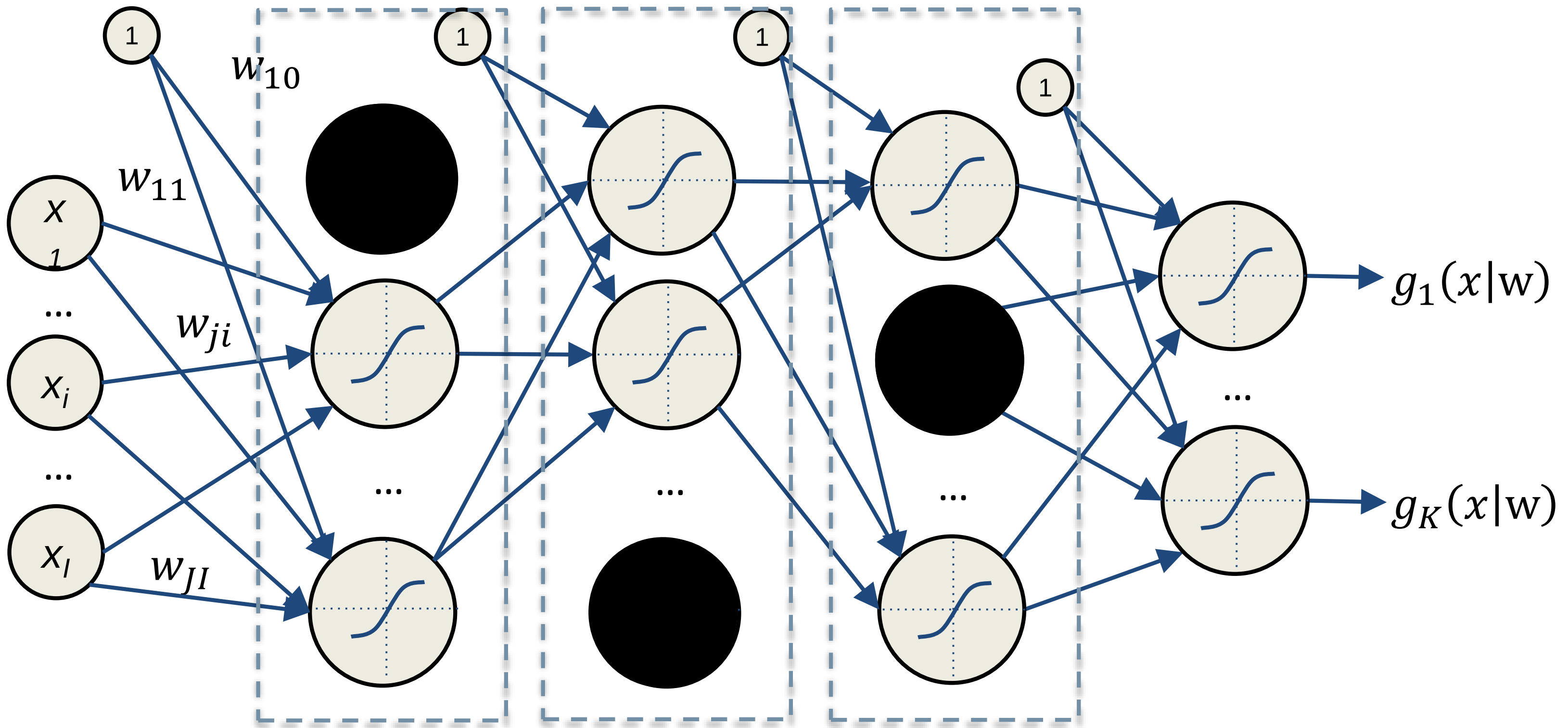
$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

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

Dropout: Limiting Overfitting by Stochastic Regularization

By turning off randomly some neurons we force them 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., $p_j^{(l)} = 0.3$



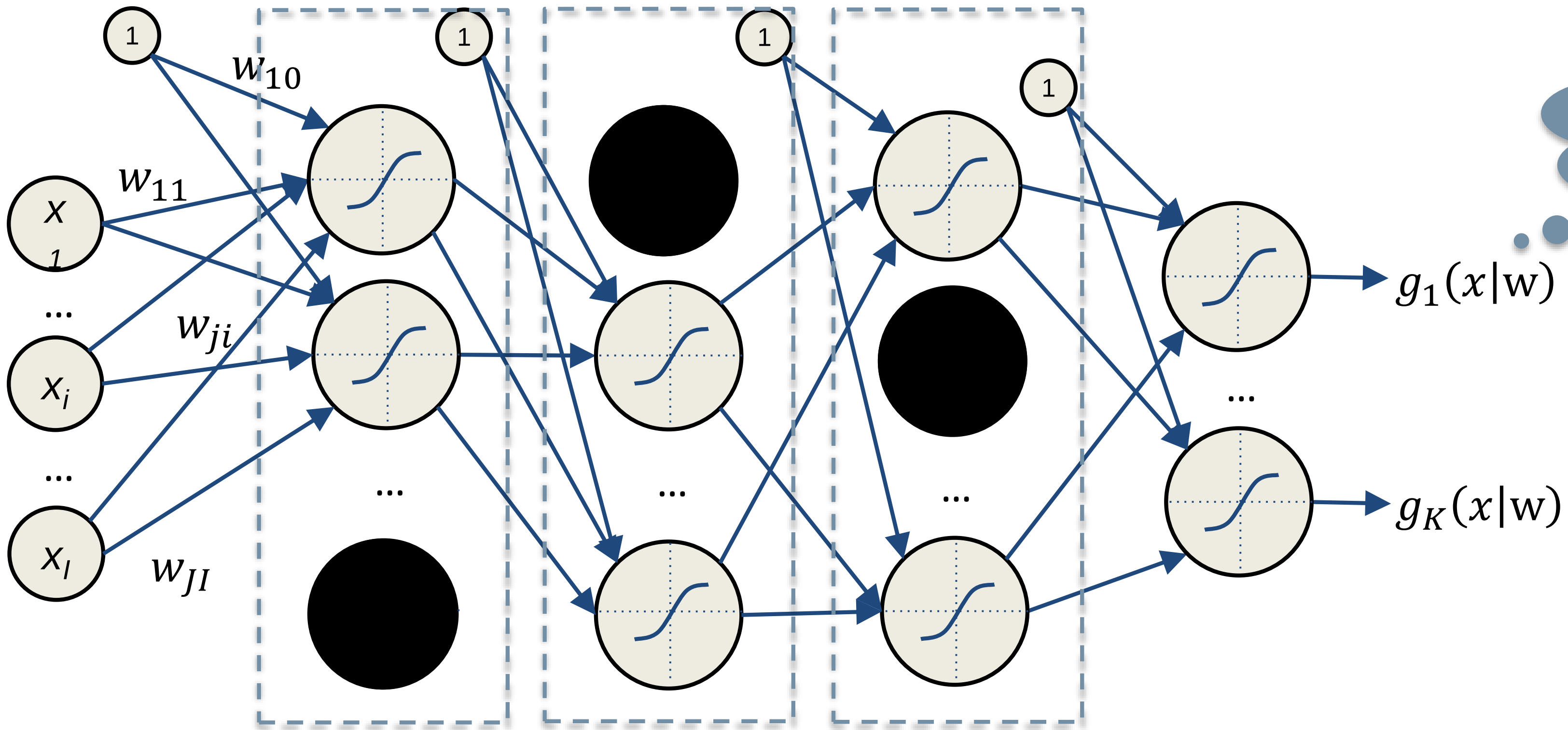
$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

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

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

Dropout: Limiting Overfitting by Stochastic Regularization

In Dropout we train a number of *weaker classifiers*, on the different mini- batch and then at test time we use them by averaging the responses of all ensemble members.

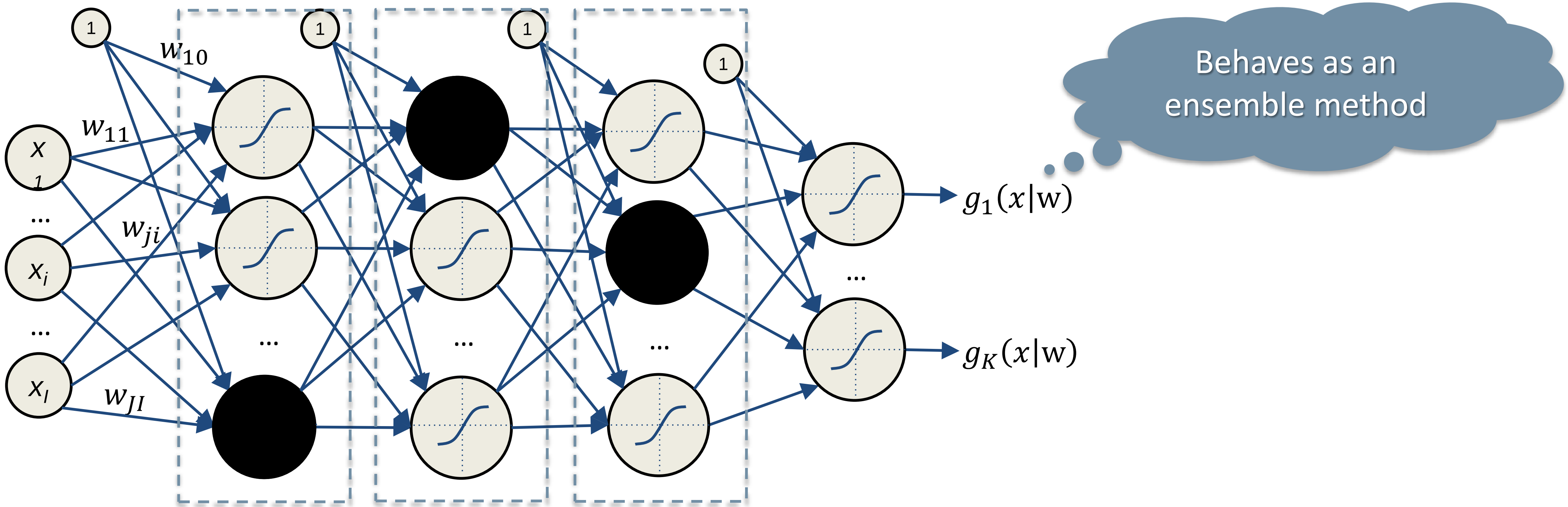


Behaves as an ensemble method

Dropout: Limiting Overfitting by Stochastic Regularization

In Dropout we train a number of *weaker classifiers*, on the different mini- batch and then at test time we use them by averaging the responses of all ensemble members.

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



POLITECNICO
MILANO 1863

Cognitive Robotics

2018/2019

Neural Networks Tips & Tricks

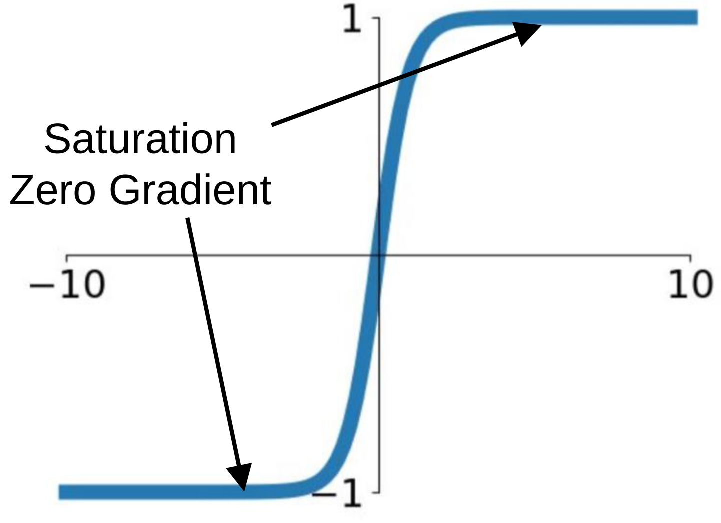
Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

Better Activation Functions

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



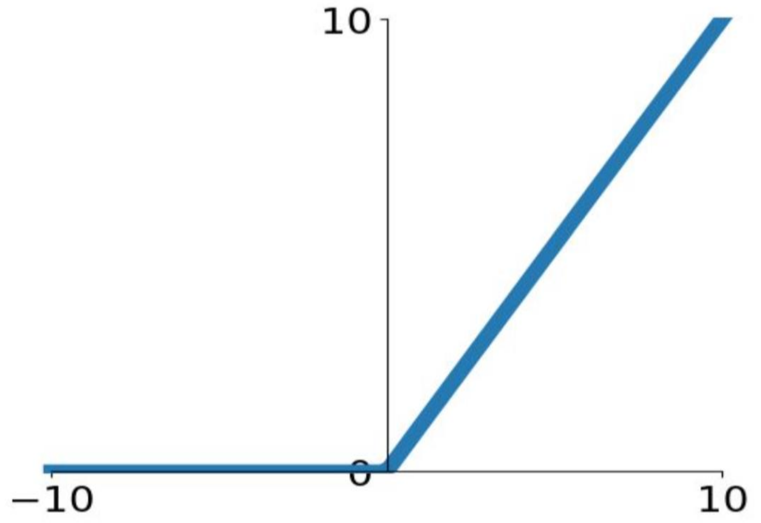
$$\frac{\partial E(w_{ji}^{(1)}))}{\partial w_{ji}^{(1)}} = -2\sum_{n}^{N}\left(t_n - g_1(x_n, w)\right) \cdot g_1'\,(x_n, w) \cdot w_{1j}^{(2)} \cdot h_j'\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right) \cdot x_i$$

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

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:

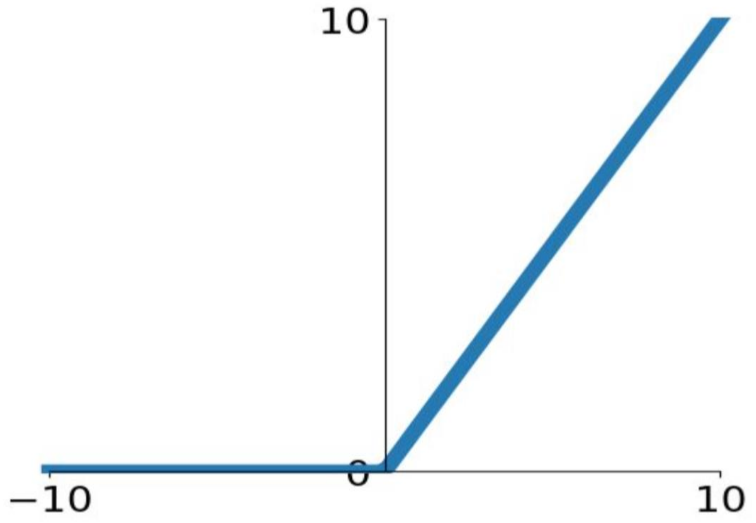
- Faster SGD Convergence (6x w.r.t sigmoid/tanh)
- Sparse activation (only part of hidden units are activated)
- Efficient gradient propagation (no vanishing or exploding gradient problems), and Efficient computation (just thresholding at zero)
- Scale-invariant:

$$\max(0, ax) = a \max(0, x)$$

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 sisadvantages:

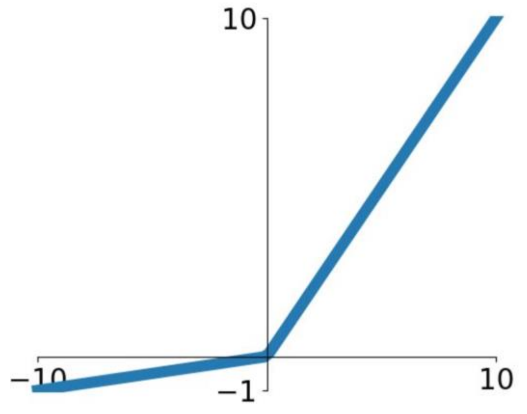
- Non-differentiable at zero: however it is differentiable anywhere else
- Non-zero centered output
- Unbounded: Could potentially blow up
- Dying Neurons: ReLU neurons can sometimes be pushed into st[illegible] which they become inactive for essentially all inputs. No gradients flow backward through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

Decreased model capacity, it happens with high learning rates

Rectified Linear Unit (Variants)

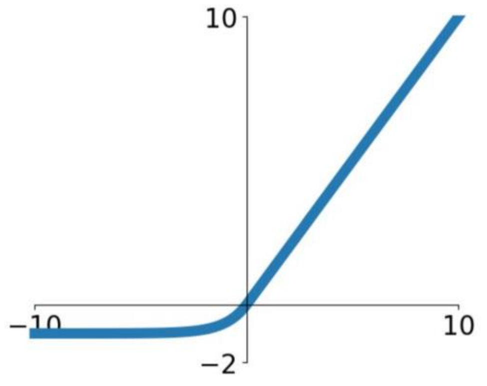
Leaky ReLU: fix for the "dying ReLU" problem

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



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

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



Weights Initialization

The final result of gradient descent is highly 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

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 …

Xavier Initialization

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

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

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

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

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

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

The variance of the output is the variance of the input, but scaled by $nVar(w_i)$.

Xavier Initialization

If we want the variance of the input and the out to be the same

$$nVar(w_j) = 1$$

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

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

Performing similar reasoning for the gradient Glorot & Bengio found

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

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

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

Recall about Backpropagation

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

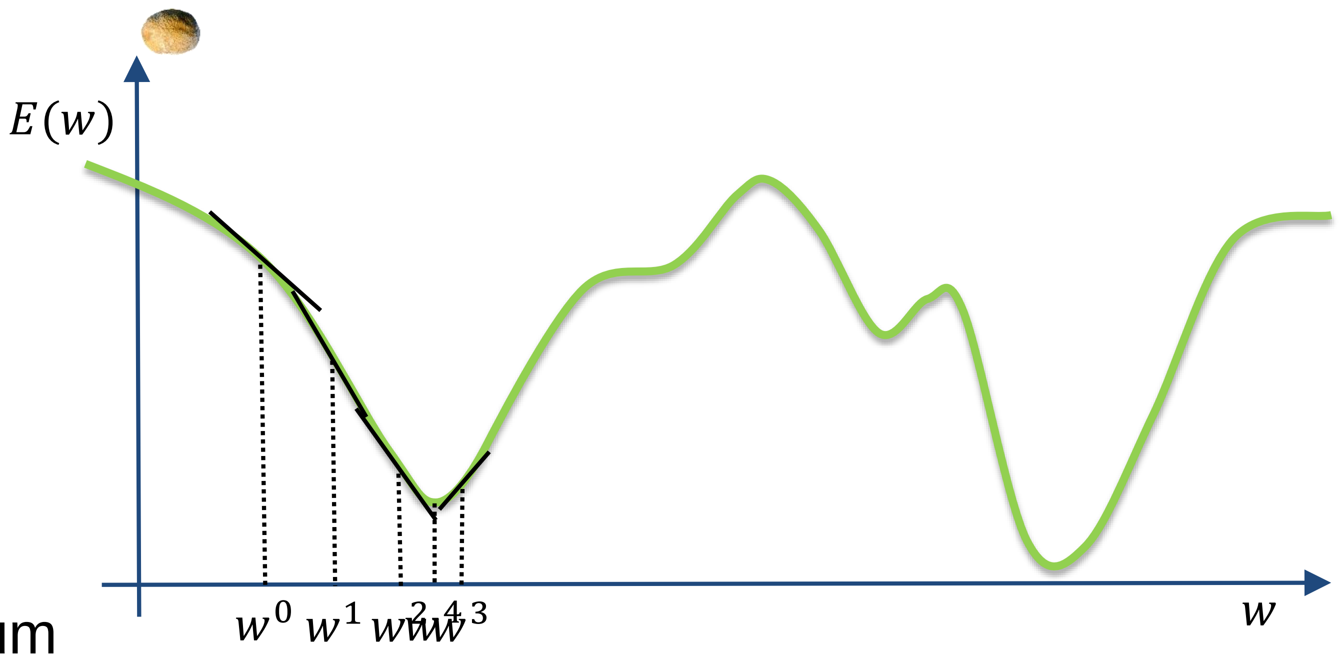
$$argmin_w \; E(w) = \sum_{n=1}^{N} (t_n - g(x_n, w))^2$$

We iterate from a initial configuration

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

To avoid local minima can use momentum

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



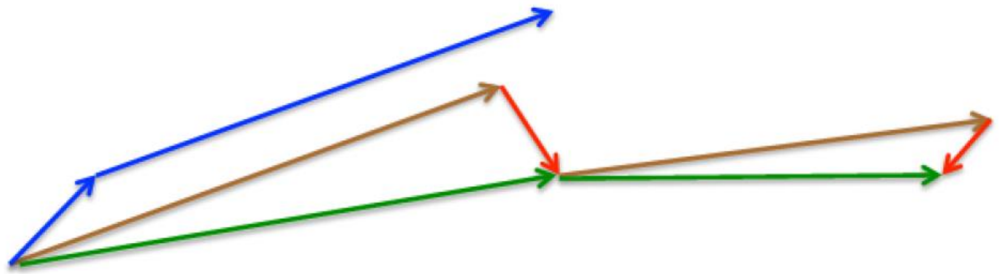
Several variations exists beside these two …

More about Gradient Descent

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

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

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



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

blue vectors = standard momentum

Adaptive Learning Rates

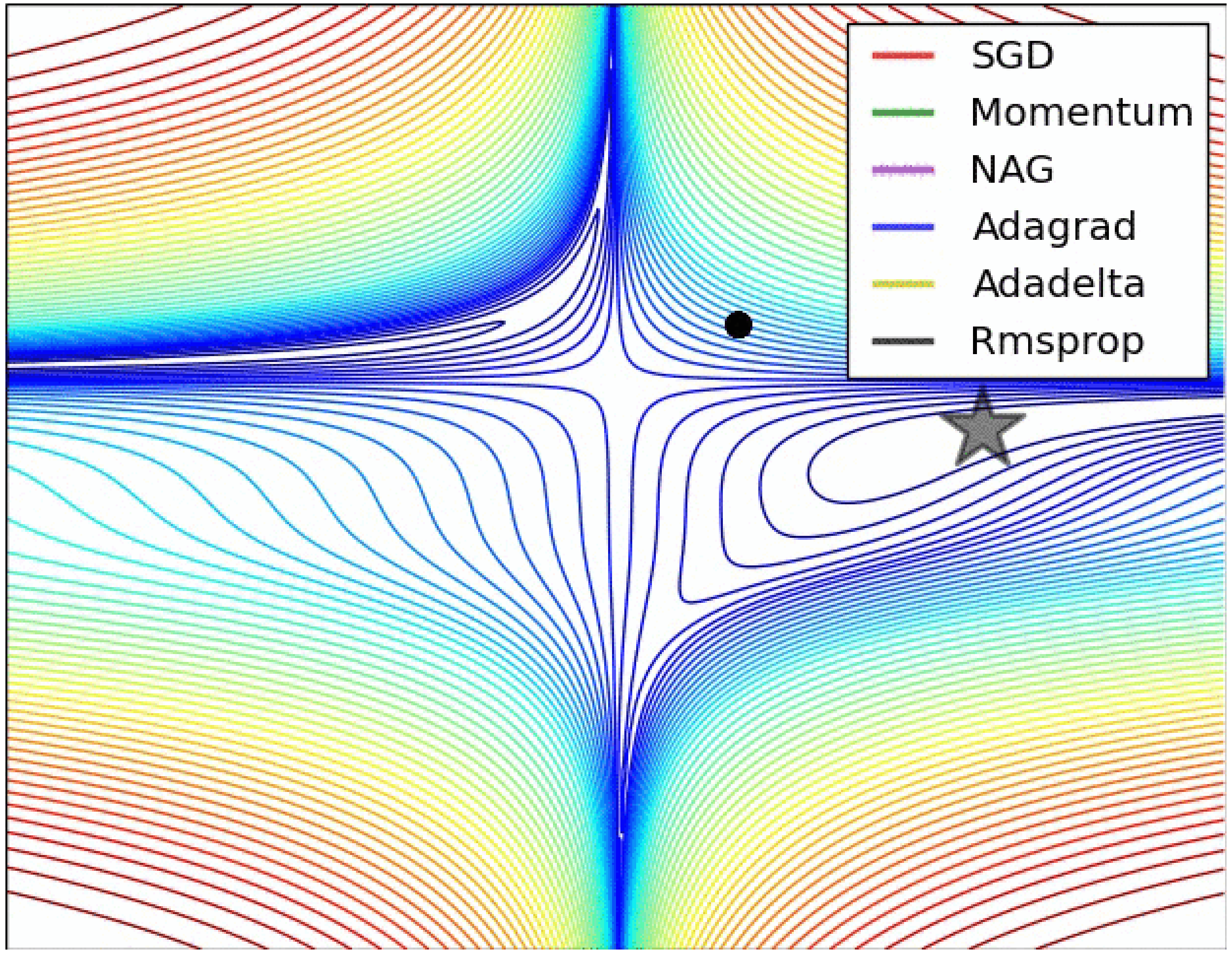
Neurons in each layer learn differently

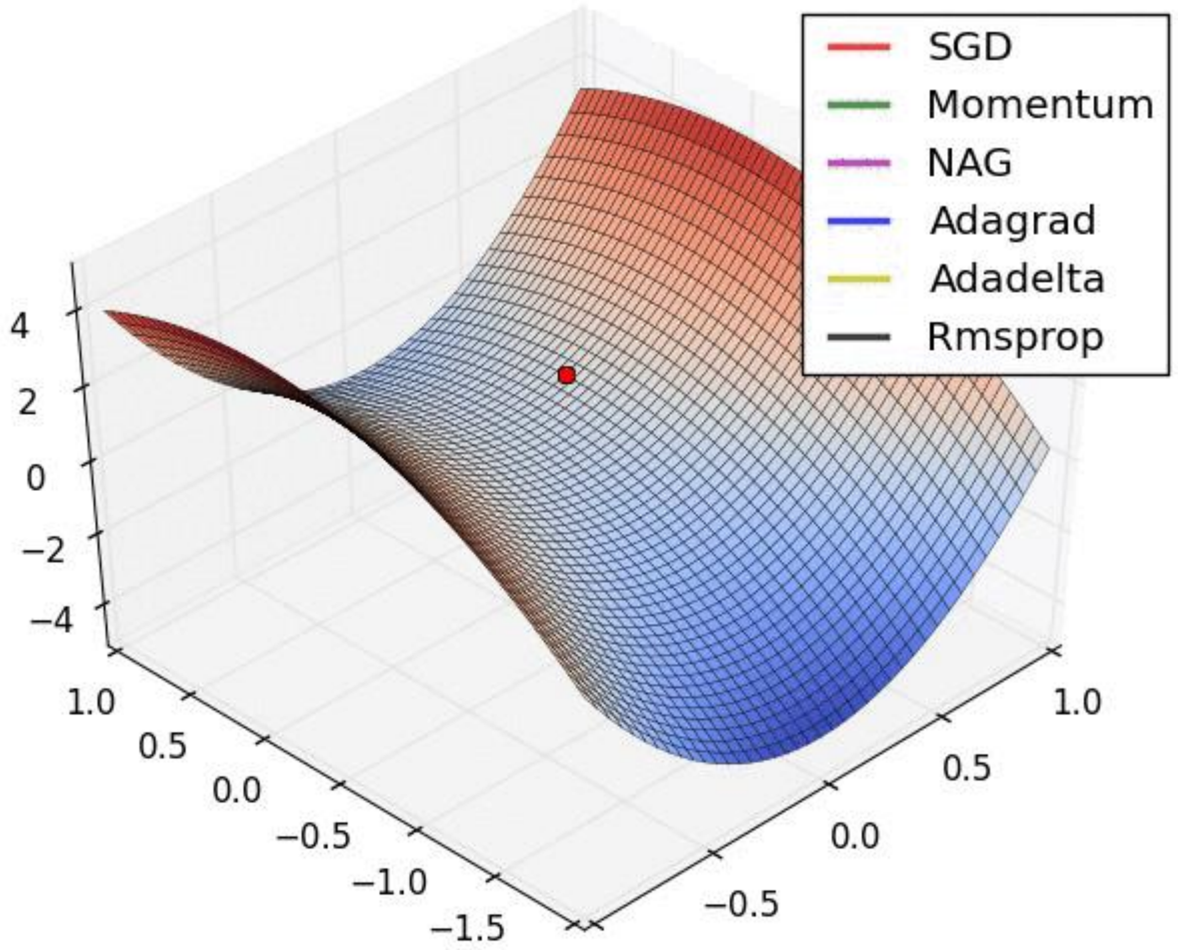
- Gradient magnitudes vary across layers
- Early layers get “vanishing gradients”
- Should ideally use separate adaptive learning rates

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)
- …

Learning Rate Matters


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop